Redux 入门

随着 JavaScript 单页应用开发日趋复杂，JavaScript 需要管理比任何时候都要多的 state （状态）。 这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括 UI 状态，如激活的路由，被选中的标签，是否显示加载动效或者分页器等等。Redux框架是用来进行数据流状态管理的，***并不仅限于react***，但是本文仅介绍如何在react中使用redux。

初次使用redux会比较痛苦，因为redux涉及到很多新的概念，但是只要掌握住这些基本概念，接下来在react代码开发中速度将会快的飞起，***因为大部分都是模板代码***

只有当交互复杂state较多的时候，引入redux管理state才有意义，如果只有一个页面展示，可能直接开发不使用数据管理框架会比较快速

三大原则

1. 单一数据源 : 整个应用的 state 被储存在一棵 object tree 中，并且这个 object tree 只存在于唯一一个 store 中
2. State 是只读的 : 唯一改变 state 的方法就是触发 action，action 是一个用于描述已发生事件的普通对象。
3. 使用纯函数来执行修改 : 为了描述 action 如何改变 state tree ，你需要编写 reducers。

什么是store，什么是纯函数，什么是 reducers，这些概念后面会介绍，先往后看

基础概念

> *Action*

Action 是把数据从应用（这里之所以不叫 view 是因为这些数据有可能是服务器响应，用户输入或其它非 view 的数据 ）传到 store 的有效载荷。它是 store 数据的唯一来源。一般来说你会通过 store.dispatch() 将 action 传到 store。

例如添加新 todo 任务的 action 是这样的

```
const ADD_TODO = 'ADD_TODO'
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

Action 本质上是 JavaScript 普通对象。我们约定，***action 内必须使用一个字符串类型的 type 字段来表示将要执行的动作***。多数情况下，type 会被定义成字符串常量。当应用规模越来越大时，建议使用单独的模块或文件来存放 action。

这里redux规定action必须要一个type的字符串常量，因为在reducers中我们根据action来修改state就是根据这个唯一的type来判断该如何修改state。

> *action creator*

Action creator 就是生成 action 的方法。“action” 和 “action creator” 这两个概念很容易混在一起，使用时最好注意区分。

在 Redux 中的 action 创建函数只是简单的返回一个 action:

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

Redux 中只需把 action 创建函数的结果传给 dispatch() 方法即可发起一次 dispatch 过程。

```
dispatch(addTodo(text))
```

> *reducer*

reducer 就是一个纯函数，接收旧的 state 和 action，返回新的 state。

```
(previousState, action) => newState
```

Redux 中只需把 action 创建函数的结果传给 dispatch() 方法即可发起一次 dispatch 过程。

```
dispatch(addTodo(text))
```

注意：

1. ***不要修改 state***。 使用 Object.assign() 新建了一个副本。不能这样使用 Object.assign(state, { visibilityFilter: action.filter })，因为它会改变第一个参数的值。你必须把第一个参数设置为空对象。
2. ***在 default 情况下返回旧的 state***。遇到未知的 action 时，一定要返回旧的 state。

下面是一个处理多个action的例子：

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    default:
      return state
  }
}
```

> *拆分 Reducer*

当reducer很长的时候，会很难看很难懂，需要根据业务不同拆分开来，这就是所谓的 reducer 合成，它是开发 Redux 应用最基础的模式。

例如：

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

Redux 提供了 `combineReducers()` 工具类来做上面 todoApp 做的事情，这样就能消灭一些样板代码了。有了它，可以这样重构 todoApp：

```
import { combineReducers } from 'redux';

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp;
```

> *Store*

Redux 应用只有一个单一的 store，Store 有以下职责：

- 维持应用的 state；
- 提供 getState() 方法获取 state；
- 提供 dispatch(action) 方法更新 state；
- 通过 subscribe(listener) 注册监听器；
- 通过 subscribe(listener) 返回的函数注销监听器。

根据已有的 reducer 来创建 store 是非常容易的。我们使用 combineReducers() 将多个 reducer 合并成为一个。现在我们将其导入，并传递 createStore()。

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp)
```

> *发起 Actions*

我们已经创建好了 store，发起action只需要从store中获取dispatch，然后dispatch action

```
store.dispatch(addTodo('Learn about actions'))
```

> *react-redux*

Redux 默认并不包含 React 绑定库，需要单独安装 `react-redux`

使用react-redux中的 `Provider` 组件把redux store注入进去：

```
import React from 'react'
import { render } from 'react-dom'
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')
)
```

使用react-redux中的 `connect` 方法把redux 与react连接起来：

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

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

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onClick: () => {
      dispatch(setVisibilityFilter(ownProps.filter))
    }
  }
}

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

其中`mapStateToProps`把store 中的 state 映射到组件中的props直接使用，`mapDispatchToProps`把store 中的 dispatch 映射到组件中的props直接使用。

TodoList

下面以一个经典todolist来介绍redux基本使用。

> *安装环境：*

1. 安装 node
2. npm install -g create-react-app
3. create-react-app redux-app
4. cd redux-app/
5. npm start

接着浏览器会默认打开 `http://localhost:3000/`，这样环境就配置成功了

> *运行效果*

> *设计组件层次结构*

- TodoList 用于显示 todos 列表。
  - todos: Array 以 { text, completed } 形式显示的 todo 项数组。
  - onTodoClick(index: number) 当 todo 项被点击时调用的回调函数。
- Todo 一个 todo 项。
  - text: string 显示的文本内容。
  - completed: boolean todo 项是否显示删除线。
  - onClick() 当 todo 项被点击时调用的回调函数。
- AddTodo 增加一个 todo 项。
  - onAddClick: 增加todo方法。
- Footer 一个允许用户改变可见 todo 过滤器的组件。
  - onFilterChange: 改变filter的方法
  - filter： filter类型常量
- App 根组件，渲染余下的所有内容。

编写代码

> *编写 Todo组件*

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';

export default class Todo extends Component {
  render() {
    return (
      <li
        onClick={this.props.onClick}
        style={{
          textDecoration: this.props.completed ? 'line-through' : 'none',
          cursor: this.props.completed ? 'default' : 'pointer'
        }}>
        {this.props.text}
      </li>
    )
  }
}

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  text: PropTypes.string.isRequired,
  completed: PropTypes.bool.isRequired
}
```

“

编写 TodoList组件

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';
import Todo from './Todo'

export default class TodoList extends Component {
    render() {
        return (
            <ul>
                {this.props.todos.map((todo, index) =>
                    <Todo {...todo}
                        key={index}
                        onClick={() => this.props.onTodoClick(index)} />
                )}
            </ul>
        )
    }
}

TodoList.propTypes = {
    onTodoClick: PropTypes.func.isRequired,
    todos: PropTypes.arrayOf(PropTypes.shape({
        text: PropTypes.string.isRequired,
        completed: PropTypes.bool.isRequired
    }).isRequired).isRequired
}
```

“

编写 Footer组件

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';

export default class Footer extends Component {
    renderFilter(filter, name) {
        if (filter === this.props.filter) {
            return name
        }

        return (
            <a href='#' onClick={e => {
                e.preventDefault()
                this.props.onFilterChange(filter)
            }}>
                {name}
            </a>
        )
    }

    render() {
        return (
            <p>
                Show:
        {' '}
                {this.renderFilter('SHOW_ALL', 'All')}
                {', '}
                {this.renderFilter('SHOW_COMPLETED', 'Completed')}
                {', '}
                {this.renderFilter('SHOW_ACTIVE', 'Active')}
                .
      </p>
        )
    }
}

Footer.propTypes = {
    onFilterChange: PropTypes.func.isRequired,
    filter: PropTypes.oneOf([
        'SHOW_ALL',
        'SHOW_COMPLETED',
        'SHOW_ACTIVE'
    ]).isRequired
}
```

“

编写 AddTodo组件

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';

export default class AddTodo extends Component {
    render() {
        return (
            <div>
                <input type='text' ref='input' />
                <button onClick={(e) => this.handleClick(e)}>
                    Add
                </button>
            </div>
        )
    }

    handleClick(e) {
        const node = this.refs.input
        const text = node.value.trim()
        this.props.onAddClick(text)
        node.value = ''
    }
}

AddTodo.propTypes = {
    onAddClick: PropTypes.func.isRequired
}
```

“

Action 创建函数和常量

```
/*
 * action 类型
 */

export const ADD_TODO = 'ADD_TODO';
export const COMPLETE_TODO = 'COMPLETE_TODO';
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'

/*
 * 其它的常量
 */

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
};

/*
 * action 创建函数
 */

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

export function completeTodo(index) {
  return { type: COMPLETE_TODO, index }
}

export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter }
}
```

这里导出了action type常量和action creator

> *Reducers*

```
import { combineReducers } from 'redux'
import { ADD_TODO, COMPLETE_TODO, SET_VISIBILITY_FILTER, VisibilityFilters } from './actions'
const { SHOW_ALL } = VisibilityFilters

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case COMPLETE_TODO:
      return [
        ...state.slice(0, action.index),
        Object.assign({}, state[action.index], {
          completed: true
        }),
        ...state.slice(action.index + 1)
      ]
    default:
      return state
  }
}

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

注意点：

1. reducer需要是一个纯函数，不能直接操作state，如果state发生变化，一定要返回一个新的state
2. 每一个reducer都需要返回一个state的默认值

> *容器组件*

```
import React, { Component } from 'react'
import PropTypes from 'prop-types';
```

```
import { connect } from 'react-redux'
import { addTodo, completeTodo, setVisibilityFilter, VisibilityFilters } from './actions'
import AddTodo from './AddTodo'
import TodoList from './TodoList'
import Footer from './Footer'

class App extends Component {
  render() {
    // Injected by connect() call:
    const { dispatch, visibleTodos, visibilityFilter } = this.props
    return (
      <div>
        <AddTodo
          onAddClick={text =>
            dispatch(addTodo(text))
          } />
        <TodoList
          todos={visibleTodos}
          onTodoClick={index =>
            dispatch(completeTodo(index))
          } />
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter =>
            dispatch(setVisibilityFilter(nextFilter))
          } />
      </div>
    )
  }
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  }).isRequired).isRequired,
  visibilityFilter: PropTypes.oneOf([
    'SHOW_ALL',
    'SHOW_COMPLETED',
    'SHOW_ACTIVE'
  ]).isRequired
}

function selectTodos(todos, filter) {
  switch (filter) {
    case VisibilityFilters.SHOW_ALL:
      return todos
    case VisibilityFilters.SHOW_COMPLETED:
      return todos.filter(todo => todo.completed)
    case VisibilityFilters.SHOW_ACTIVE:
      return todos.filter(todo => !todo.completed)
  }
}

// Which props do we want to inject, given the global state?
// Note: use https://github.com/faassen/reselect for better performance.
function select(state) {
```

```
  return {
    visibleTodos: selectTodos(state.todos, state.visibilityFilter),
    visibilityFilter: state.visibilityFilter
  }
}

// 包装 component ，注入 dispatch 和 state 到其默认的 connect(select)(App) 中;
export default connect(select)(App)
```

注意点：通过react-redux的connect方法连接reduc与react，被包装了之后的容器组件可以在组件内部使用store中的state与dispatch

> *注入store*

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './component'
import todoApp from './component/reducers'
let store = createStore(todoApp)

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

调用redux的createStore方法包装reducer得到store，再把store注入到子组件中

总结

整个demo比较简单，如果跟着代码把TodoApp写一遍，基本可以入门，但是中间件的使用还未涉及，可以参照官方文档学习http://cn.redux.js.org/docs/introduction/index.html

demo源码地址