# Flux与Redux学习

## 参考文章

<http://www.ruanyifeng.com/blog/2016/09/react-technology-stack.html>

<http://www.redux.org.cn/>

## 不使用Flux架构

### 数据流动方式

用户操作

View层

组件

Component

State

状态

### 分析过程

* View视图层

用户访问页面，执行交互行为（例如：点击）。

* Component组件层

触发组件内部定义的click事件

* Component组件层

逻辑处理，修改组件状态（setState方法）。

React自动执行render方法，生成虚拟DOM，执行diff算法。

* View视图层

更新视图。

## Flux

参考文章：<http://www.ruanyifeng.com/blog/2016/01/flux.html>

Git地址：<https://github.com/nmwei/flux/tree/master/extremely-simple-flux-demo>

### Flux介绍

Flux是Facebook提供的一种设计架构思想，但是facebook只提供了Flux的设计思路和原理，Flux存在多种实现方式。

### 基本概念

1. Flux将一个应用分成四个部分:

(1)View：视图层

(2)Action（动作）：用户行为的抽象。

(3)Dispatcher（派发器）：用来接收Actions、执行回调函数

(4)Store（数据层）：用来存放应用的状态，一旦发生变动，就提醒Views要更新页面

1. Flux将组件分为两类:

(1)纯组件：纯粹的UI层，组件内没有状态(State)，根据外层组件传递的props生成对应视图。

(2)容器组件：处理纯组件的数据和事件函数。

### 数据流动方式

委托

用户操作

View层

展示组件

Component

容器组件

Component

动作

Action

派发器

Dispatcher

数据

Store

### 分析过程

* View视图层

用户访问页面，执行交互行为（例如：点击）。

* Component(纯组件)

触发纯组件内部定义的click事件。

* Component(容器组件)

子组件（纯组件）委托父组件（容器组件）处理点击事件的逻辑。

* Action层

将交互行为（点击）抽象为动作。

* Dispatcher层

派发器接收动作，执行该动作对应的回调函数。

* Store层

修改数据，并发出一个"change"事件,调用该事件注册的所有回调函数。

* Component(容器组件)

容器组件监听"change"事件，更新组件状态，并将状态传递到纯组件中。

* Component(纯组件)

React自动执行render方法，生成虚拟DOM，执行diff算法。

* View视图层

更新视图。

## Redux

参考文章:

<http://www.ruanyifeng.com/blog/2016/09/redux_tutorial_part_one_basic_usages.html>

<https://github.com/kenberkeley/redux-simple-tutorial>

**<http://cn.redux.js.org/index.html>**

Git地址:

<https://github.com/nmwei/redux/tree/master/Redux/counter>

### Redux介绍

Redux是Flux架构思想的一种实现方式。

### 安装

1. 安装稳定版

npm install --save redux

1. 安装附加包

npm install --save react-redux

npm install --save-dev redux-devtools

### 核心概念

1. Store (数据容器)

Store 是保存数据的容器。一个应用只能有一个 Store。

1. State(所有数据)

State与View一一对应。只要State 相同，View 就相同。

1. Action(动作)

Action 是一个对象，是对用户行为的抽象。

1. 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。

注意: 当 state 变化时需要返回全新的对象，而不是修改传入的参数。

### 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。

1. State 是只读的

惟一改变 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。

1. 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。Reducer 只是一些纯函数，它接收先前的 state 和 action，并返回新的 state。刚开始你可以只有一个 reducer，随着应用变大，你可以把它拆成多个小的 reducers，分别独立地操作 state tree 的不同部分。

### Store数据容器

1. Store数据容器的创建

import { createStore } from 'redux';

//传入reducer函数，创建Store数据容器

const store = createStore(reducer);

1. Store数据容器的三个方法

store.getState(); //得到state数据

store.dispatch(action); //更新数据

store.SubScribe(fn); //监听数据变化

注意: **Redux 应用只有一个单一的 store。当需要拆分数据处理逻辑时，你应该使用 reducer 组合 而不是创建多个 store。**

### 数据流动方式

委托

reducer(action, state)

Store.subscribe(linternsr)

Store.dispatch(action)

用户操作

View层

展示组件

Component

容器组件

Component

动作

Action

数据容器

Store

更新状态函数

Reducer

### 分析过程

* View视图层

用户访问页面，执行交互行为（例如：点击）。

* Component(纯组件)

触发纯组件内部定义的click事件。

* Component(容器组件)

子组件（纯组件）委托父组件（容器组件）处理点击事件的逻辑。

* Action层

将交互行为（点击）抽象为动作。

-----------------------以上步骤与Flux部分相同-------------------

* Dispatch层

数据容器执行store.dispatch(action)方法

* Reducer层

Reducer根据旧状态和接收到的action生成新状态。

* Store层

state数据变化，自动执行store.SubScribe(fn);

* Component(容器组件)

设置的监听函数fn更新组件状态，并将状态传递到纯组件中。

* Component(纯组件)

React自动执行render方法，生成虚拟DOM，执行diff算法。

* View视图层

更新视图。

### 要点

1. 应用中所有state都以一个对象树的形式存储在一个单一的store中， 惟一改变 state 的办法是触发 action(一个描述发生什么的对象)。
2. store 由 Redux 的 createStore(reducer) 生成。
3. state 通过 store.getState() 获取，本质上一般是一个存储着整个应用状态的对象。
4. action 本质上是一个包含 type 属性的普通对象，由 Action Creator (函数) 产生。
5. 改变 state 必须 dispatch 一个 action。
6. reducer 本质上是根据 action.type 来更新 state 并返回 nextState 的函数(描述 action 如何改变 state 树)。
7. reducer 必须返回值，否则 nextState 即为 undefined
8. 实际上，state 就是所有 reducer 返回值的汇总（本教程只有一个 reducer，主要是应用场景比较简单）。
9. Redux与Flux对比有一个重要的区别。

Redux 没有 Dispatcher 且不支持多个 store。相反，只有一个单一的 store 和一个根级的 reduce 函数（reducer）。随着应用不断变大，你应该把根级的 reducer 拆成多个小的 reducers，分别独立地操作 state 树的不同部分，而不是添加新的 stores。这就像一个 React 应用只有一个根级的组件，这个根组件又由很多小组件构成。

## React-Redux

参考文章:

<http://www.ruanyifeng.com/blog/2016/09/redux_tutorial_part_three_react-redux.html>

<http://cn.redux.js.org/index.html>

Git地址: <https://github.com/nmwei/redux>

### React-Redux介绍

为了方便使用，Redux 的作者封装了一个 React 专用的库 React-Redux。它虽然提供了便利，但是需要掌握额外的 API，并且要遵守它的组件拆分规范。

### 基本概念

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

* 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

UI 组件又称为"纯组件"，即它与纯函数一样，纯粹由参数决定它的值。

注释:没有状态state，没有业务逻辑，只有属性props。

1. 容器组件

* 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

注释:只负责数据和逻辑。

总结:UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。

1. 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，状态管理则全部交给它。

### React-Redux API

#### 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。

#### 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。

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。

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。

#### 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。

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。

注意: Why don't I have this.props.dispatch available in my connected component? <http://redux.js.org/docs/faq/ReactRedux.html#react-props-dispatch>

如果展示组件使用connect方法生成容器组件时没有提供mapDispatchToProps函数，react-redux自动将dispatch函数作为一个props。如果提供了mapDispatchToProps函数，react-redux将不会自动提供dispatch作为一个props。

#### <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。

### Middleware

1. 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。
2. Redux middleware 提供了位于 action 被发起之后，到达 reducer 之前的扩展点。 你可以利用 Redux middleware 来进行日志记录、创建崩溃报告、调用异步接口或者路由等等。
3. 使用 import thunk from 'redux-thunk' 的thunk这个middleware后，允许我们通过dispatch() 传递函数。此时我们可以写返回函数的Action Creator。若一个Action Creator返回函数，则返回的函数的参数是**dispatch**和**getState**这两个 Redux 方法。

总结:①dispatch(obj);

②dispatch(function(dispatch, getState){})

即:dispatch((参数)=> (dispatch, getState) =>{})

**注意:(参数A) => (参数B) => {代码块C} 含义为执行函数传递参数A，返回一个匿名函数，该匿名函数的参数为B，对应代码块C。**

### 示例总结

#### Actions 动作生成器

1. 向外暴露action动作名称和动作生成器函数名称。
2. 有时向外暴露某些事件类型的参数。

#### Reducers函数

1. import {combineReducers} from 'redux'
2. 通过combineReducers()方法将多个reducer合成。
3. reducer函数中返回新的state而不要修改原来的state。

#### components展示组件

1. 使用React函数式声明方式。
2. import React, {PropTypes} from 'react'
3. 组件中只有props，组件中的props分为两类。
4. 一类是容器组件通过mapStateToProps传过来的数据prop，作用是由外向内数据传递。一类是容器组件通过mapDispatchToProps传过来的函数prop，作用是由内向外数据传递。

#### Container 容器组件

1. 通过React Redux 的 connect() 方法自动生成。
2. connect()方法中的mapStateToProps和mapDispatchToProps要涵盖该容器组件对应的展示组件中所用到的所有props属性。
3. mapStateToProps函数用到两个参数——**该项目总的state树**，和**用到该容器组件时传递的props**。返回的键值对形式一般如下:

{数据prop : fn(state, ownProps)}

1. mapDispatchToProps函数用到两个参数——dispatch和用到该容器组件时传递的props。返回的键值对形式一般如下:

{函数props :dispath(actionCreater(data, ownProps))}

## 其他

### ES6写法

1. {type:'ADD\_TODO', id:1, text}

### React函数式声明组件

1. 参考地址: <http://www.tuicool.com/articles/2iiM3uj>
2. 对于无状态的简单组件，使用函数式组件的方式声明，会使得代码的可读性更好，并且减少冗余代码的数目。在React本身对于界面的抽象可以用 View = f(props) ，即纯粹的界面的渲染函数可以近似看做纯函数。

### 错误点

1. 函数式组件声明方式的易错写法

const AddTodo = (**{**onClick**}**) => {}) //忘记花括号

1. react-redux提供的connect方法

connect([mapStateToProps],[mapDispatchToProps], [mergeProps], [options])(TodoApp)

1. connect([mapStateToProps],[mapDispatchToProps])(TodoApp)
2. connect(null,[mapDispatchToProps])(TodoApp)
3. connect([mapStateToProps])(TodoApp)

### 插件

1. redux-actions

<https://github.com/acdlite/redux-actions>

1. normalizr

<https://github.com/paularmstrong/normalizr>