原文地址

继 **Facebook** 提出 **Flux** 架构来管理 **React** 数据流后，相关架构开始百花齐放，本文简单分析 **React** 中管理数据流的方式，以及对 **Redux** 进行较为仔细的介绍。

# React

" A JAVASCRIPT LIBRARY FOR BUILDING USER INTERFACES "

在 React 中，UI 以组件的形式来搭建，组件之间可以嵌套组合。另，React 中组件间通信的数据流是单向的，顶层组件可以通过 props 属性向下层组件传递数据，而下层组件不能向上层组件传递数据，兄弟组件之间同样不能。这样简单的单向数据流支撑起了 React 中的数据可控性。

那么，更全面的组件间通信形式该怎么实现呢？

1. 嵌套组件间，上层组件向下层组件传递回调函数，下层组件触发回调来更新上层组件的数据。
2. 以事件的形式，使用发布订阅的方式来通知数据更新。
3. Flux —- Fackbook 提出的管理 React 数据流的架构。Flux 不像一个框架，更是一种组织代码的推荐思想。就像 “引导数据流流向的导流管”。
4. 其他的 “导流管”。ReFlux，Redux 等。

前两种形式其实也足够在小应用中跑起来。但当项目越来越大的时候，管理数据的事件或回调函数将越来越多，也将越来越不好管理了。 对于后两种形式，个人经过对比后，可以看出 Redux 对 Flux 架构的一些简化。如 Redux 限定一个应用中只能有单一的 store，这样的限定能够让应用中数据结果集中化，提高可控性。当然，不仅如此。

# Redux

Redux 主要分为三个部分 Action、Reducer、及 Store

## Action

在 Redux 中，action 主要用来传递操作 State 的信息，以 Javascript Plain Object 的形式存在，如

```
{
  type: 'ADD_FILM',
  name: 'Mission: Impossible'
}
```

在上面的 Plain Object 中，type 属性是必要的，除了 type 字段外，action 对象的结构完全取决于你，建议尽可能简单。type 一般用来表达处理 state 数据的方式。如上面的 'ADD_FILM' 表达要增加一个电影。而 name 表达了增加这个电影的电影名为 'Mission: Impossible'。那么，当我们需要表达增加另一部电影时，就需要另外一个action，如

```
{
  type: 'ADD_FILM',
  name: 'Minions'
}
```

上面写法没有任何问题，但细想，当我们增加的电影越来越多的时候，那这种直接声明的 Plain Object 将越来越多，不好组织。实际上，我们可以通过创建函数来生产 action，这类函数统称为 Action Creator，如

```
function addFilm(name) {
  return { type: 'ADD_FILM', name: name };
}
```

这样，通过调用 addFilm(name) 就可以得到对应的 Action，非常直接。

## Reducer

有了 Action 来传达需要操作的信息，那么就需要有根据这个信息来做对应操作的方法，这就是

Reducer。 Reducer 一般为简单的处理函数，通过传入旧的 state 和指示操作的 action 来更新 state，如

```
function films(state = initialState, action) {
  switch (action.type) {

  case 'ADD_FILM':
    // 更新 state 中的 films 字段
    return [{
      id: state.films.reduce((maxId, film) => Math.max(film.id, maxId), -1)
      name: action.name
    }, ...state];

  case 'DELETE_FILM':
    return state.films.filter(film =>
        film.id !== action.id
    );

  case 'SHOW_ALL_FILM':
    return Object.assign({}, state, {
        visibilityFilter: action.filter
      });

  default:
    return state;
  }
```

上面代码展示了 Reducer 根据传入的 action.type 来匹配 case 进行不同的 state 更新。

显然，当项目中存在越来越多的 action.type 时，上面的 films 函数（ Reducer ）将变得越来越大，越来越多的 case 将导致代码不够清晰。所以在代码组织上，通常会将 Reducer 拆分成一个个小的 reducer，每个 reducer 分别处理 state 中的一部分数据，最终将处理后的数据合并成为整个 state。

在上面的代码中，我们可以把 'ADD_FILM' 和 'DELETE_FILM' 归为操作 state.films 的类，而 'SHOW_ALL_FILM' 为过滤显示类，所以可以把大的 film Reducer 拆分成 filmReducer 和 filterReducer，如

1 filmReducer

```
function filmReducer(state = [], action) {
  switch (action.type) {
  case 'ADD_FILM':
  // 更新 state 中的 films 字段
  return [{
    id: state.films.reduce((maxId, film) => Math.max(film.id, maxId), -1) +
    name: action.name
  }, ...state];
  case 'DELETE_FILM':
  return state.films.filter(film =>
      film.id !== action.id
  );
  default:
    return state;
  }
```

2 filterReducer

```
function filterReducer(state, action) {
  switch (action.type) {
  case 'SHOW_ALL_FILM':
    return Object.assign({}, state, {
    visibilityFilter: action.filter
    });
  default:
    return state;
  }
```

最后，通过组合函数将上面两个 reducers 组合起来，如

```
function rootReducer(state = {}, action) {
  return {
    films: filmReducer(state.films, action),
    filter: filterReducer(state.filter, action)
  };
```

上面的 rootReducer 将不同部分的 state 传给对应的 reducer 处理，最终合并所有 reducer 的返回值，组成整个state。

实际上，Redux 提供了 combineReducers() 方法来做 rootReducer 所做的事情。使用 combineReducers 来重构 rootReducer，如

```
var rootReducer = combineReducers({
    films: filmReducer,
    filter: filterReducer
});
```

combineReducers() 将调用一系列 reducer，并根据对应的 key 来筛选出 state 中的一部分数据给相应的 reducer，这样也意味着每一个小的 reducer 将只能处理 state 的一部分数据，如：filterReducer 将只能处理及返回 state.filter 的数据，如果需要使用到其他 state 数据，那还是需要为这类 reducer 传入整个 state。

在 Redux 中，一个 action 可以触发多个 reducer，一个 reducer 中也可以包含多种 action.type 的处理。属于多对多的关系。

## Store

回顾 Action 及 Reducer：

Action 用来表达操作消息，Reducer 根据 Action 来更新 State。

在 Redux 项目中，Store 是单一的。维护着一个全局的 State，并且根据 Action 来进行事件分发处理 State。可以看出 Store 是一个把 Action 和 Reducer 结合起来的对象。

Redux 提供了 createStore() 方法来 生产 Store，并提供三个 API，如

```
var store = createStore(rootReducer);   // 其中 rootReducer 为顶级的 Reducer
```

store 对象可以简单的理解为如下形式

```
function createStore(reducer, initialState) {

    //闭包私有变量
    var currentReducer = reducer;
    var currentState = initialState;
    var listeners = [];

    function getState() {
      return currentState;
    }

    function subscribe(listener) {
      listeners.push(listener);

      return function unsubscribe() {
        var index = listeners.indexOf(listener);
        listeners.splice(index, 1);
      };
    }

    function dispatch(action) {
        currentState = currentReducer(currentState, action);
        listeners.slice().forEach(listener => listener());
        return action;
    }

    //返回一个包含可访问闭包变量的公有方法
    return {
      dispatch,
      subscribe,
      getState
    };
}
```

store.getState() 用来获取 state 数据。

store.subscribe(listener) 用于注册监听函数。每当 state 数据更新时，将会触发监听函数。

而 store.dispatch(action) 是用于将一个 action 对象发送给 reducer 进行处理。如

```
store.dispatch({
  type: 'ADD_FILM',
  name: 'Mission: Impossible'
});
```

store 对象使得我们可以通过 store.dispatch(action) 来减少对 reducer 的直接调用，并且能够更好地对 state 进行统一管理。没有 store，可能会出现 reducer(currentState, action) 这样的频繁地传入 state 参数的更新形式。

### bindActionCreators

从上面的 Action 相关介绍中可知，我们使用了 ActionCreator 来生产 action。所以在实际的 store.dispatch(action) 中，我们需要这样调用 store.dispatch(actionCreator(…args))。

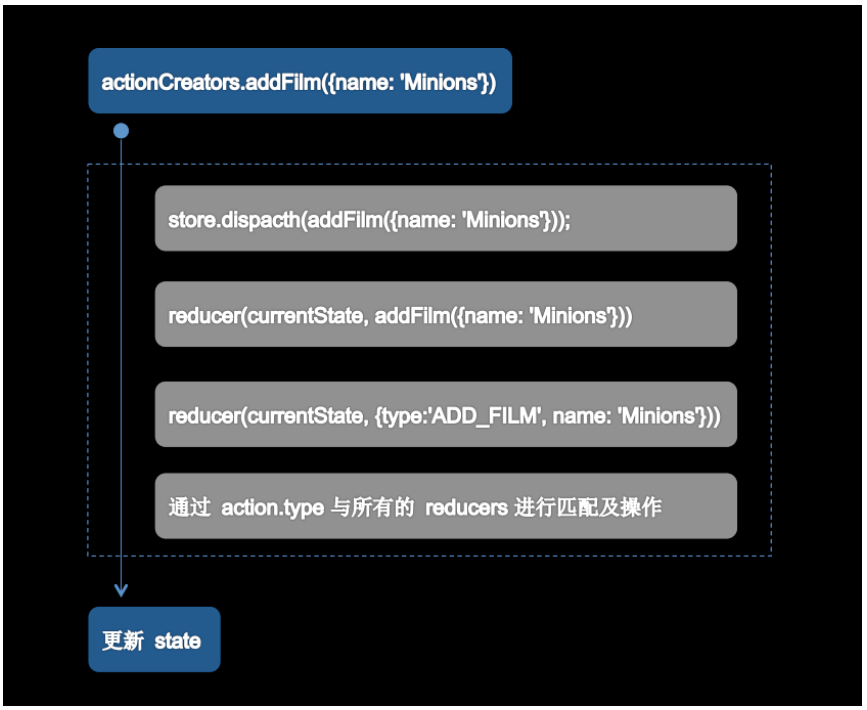
借鉴 Store 对 reducer 的封装（减少传入 state 参数）。可以对 store.dispatch 进行再一层封装，将多参数转化为单参数的形式。 Redux 提供的 bindActionCreators 就做了这件事。如

```
var actionCreators = bindActionCreators(actionCreators, store.dispatch);
```

现在，经 bindActionCreators 包装过后的 action Creator 形成了具有改变全局 state 数据的多个函数，将这些函数分发到各个地方，即能通过调用这些函数来改变全局的 state。

### Redux 中的函数传递及原理

当调用了具备操作全局 state 的函数时，将经过一系列的函数传递及调用，如




问：为什么不直接使用 reducer(currentState, {type:'ADD_FILM', name: 'Minions'})) 呢？

答：这样做除了在代码组织和扩展维护上提供了便利，同时也涵盖了函数式编程的许多优点。

## React-Redux

Redux 并不依赖于 React，它支持多种框架 Ember、Angular、jQuery 甚至纯 JavaScript。但实际上，它更合适由 数据更新 UI 的框架。如 React、Deku。

上面的章节最终通过 bindActionCreators 得到具有操作全局 state 的函数集合，在与 React 搭配时，就会将这些函数分发到各个对应的组件中，从而组件具备了操作全局的 state 的功能。在上节中可以得到，调用操作全局 state 的函数，最终将更新 state。当 redux 与 react 结合，在更新 state 时，将会触发 重新渲染 组件的函数，进而组件得到更新。

react-redux 主要提供两个组件来实现上述功能。

### Connect

Connect 组件主要为 React 组件提供 store 中的部分 state 数据 及 dispatch 方法，这样 React 组件就可以通过 dispatch 来更新全局 state。在 React 组件中，如果你希望让组件通过调用函数来更新 state，可以通过使用 const actions = bindActionCreators(FilmActions, dispatch); 将 actions 和 dispatch 揉在一起，成为具备操作 store.state 的 actions。最终将 actions 和 state（state.films）以 props 形式传入子组件中。如

```
import { connect } from 'react-redux';
import * as flimActions from '../actions/films';
// 其他模块引入..

class FilmApp extends Component {
  render() {
    // 从 react-redux 注入
    const { films, dispatch } = this.props;

  // 生成具有操作 state 能力的 actions
  const actions = bindActionCreators(flimActions, dispatch);

  // 为各个 React 组件提供 state 数据 及 actions
  return (
    <div>
      <Header films={films} actions={actions}/>
      <Section films={films} deleteFilm={actions.deleteFilm}/>
    </div>
  );
  }
}

// state 将由 store 提供
function select(state) {
  return {
    films: state.films
  };
}

// 最终暴露 经 connect 处理后的组件
export default connect(select)(FilmApp);
```

由上，在 redux 提供的 connect 函数中，select 函数用于筛选 state 的部分数据，最终和 dispatch 以 props 的形式传给 React 组件（FilmApp）。FilmApp 就可通过 this.props 来得到 store 中的 state 及 dispatch。

在 redux 中，没有与 redux 有直接关联的组件称为木偶组件，如 FilmApp 下的子组件，不理外面纷纷扰扰，只知道自己拥有了 state 及 具备操作 state 数据的 actions 方法。

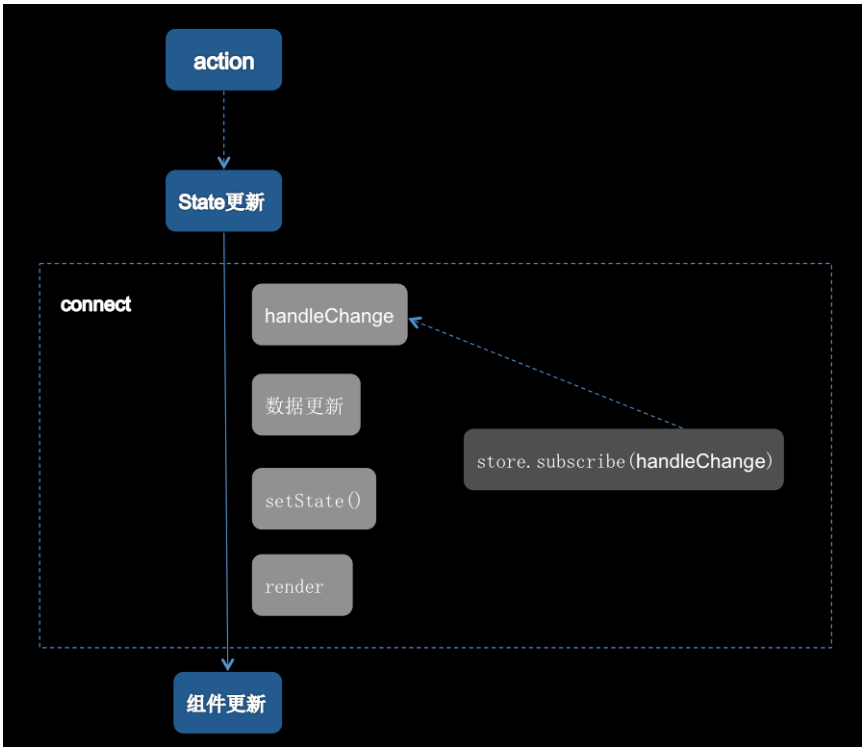
当木偶组件使用 actions 方法，更新了 store.state 的数据时，将会触发 store 中的 subscribe 所注册的函数。而其中一个注册函数，就在 Connect 组件中静默注册了。

```
// 在 Connect 中
this.store.subscribe(this.handleChange.bind(this));
```

即当 actions 更改了 state 时，会调用注册函数 handleChange。从而进行 “阿米诺骨牌式” 的函数执行连锁反应。更新了 state，并使用新的数据重新 render 组件。实际上是为智能组件 FilmApp（传入 connect 的组件）传入新的 props，因为各个子元素是通过引用父级组件的 props，所以将进行一级一级的差异数据更新，最终效果就是页面更新了。

实际上，这里与简单的发布订阅模式类似。使用 store.subscribe(cb); 来订阅一个回调函数，子组件进行 action 操作 store.state 时进行发布，执行了回调函数。

在 react-redux 中，数据的流向及对应的反应，如




## Provider

Connect 组件需要 store，这个需求由 Redux 提供的另一个组件 Provider 来提供，源码中，Provider 继