Isomorphic-fetch

**Installation**

**NPM**

npm install --save isomorphic-fetch es6-promise

## Usage

require('es6-promise').polyfill();

require('isomorphic-fetch');

fetch('//offline-news-api.herokuapp.com/stories')

    .then(function(response) {

**if** (response.status **>=** 400) {

**throw** **new** Error("Bad response from server");

        }

**return** response.json();

    })

    .then(function(stories) {

        console.log(stories);

    });

AJAX半遮半掩的底层API是饱受诟病的一件事情. **XMLHttpRequest** 并不是专为Ajax而设计的. 虽然各种框架对 **XHR** 的封装已经足够好用, 但我们可以做得更好。

window.fetch 方法, 在最新版的 Firefox 和 Chrome 中已经提供支持。

// url (必须), options (可选)

fetch('/some/url', {

method: 'get'

}).then(function(response) {

}).catch(function(err) {

//中途任何地方出错...在此处理

:(

});

//使用 ES6 的 箭头函数 后：

fetch(url).then(response => response.json())

.then(data => console.log(data))

.catch(e => console.log("Oops, error", e))

### API

fetch(url,{ // url: 请求地址

method: "GET", // 请求的方法POST/GET等

headers : { // 请求头（可以是Headers对象，也可是JSON对象）

'Content-Type': 'application/json',

'Accept': 'application/json'

},

body: , // 请求发送的数据 blob、BufferSource、FormData、URLSearchParams（get或head方法中不能包含body）

cache : 'default', // 是否缓存这个请求

credentials : 'same-origin', //要不要携带 cookie 默认不携带 omit、same-origin 或者 include

mode : "",

/\*

mode,给请求定义一个模式确保请求有效

same-origin:只在请求同域中资源时成功，其他请求将被拒绝（同源策略）

cors : 允许请求同域及返回CORS响应头的域中的资源，通常用作跨域请求来从第三方提供的API获取数据

cors-with-forced-preflight:在发出实际请求前执行preflight检查

no-cors : 目前不起作用（默认）

\*/

}).then(resp => {

/\*

Response 实现了 Body, 可以使用 Body 的 属性和方法:

resp.type // 包含Response的类型 (例如, basic, cors).

resp.url // 包含Response的URL.

resp.status // 状态码

resp.ok // 表示 Response 的成功还是失败

resp.headers // 包含此Response所关联的 Headers 对象 可以使用

resp.clone() // 创建一个Response对象的克隆

resp.arrayBuffer() // 返回一个被解析为 ArrayBuffer 格式的promise对象

resp.blob() // 返回一个被解析为 Blob 格式的promise对象

resp.formData() // 返回一个被解析为 FormData 格式的promise对象

resp.json() // 返回一个被解析为 Json 格式的promise对象

resp.text() // 返回一个被解析为 Text 格式的promise对象

\*/

if(resp.status === 200) return resp.json();

// 注： 这里的 resp.json() 返回值不是 js对象，通过 then 后才会得到 js 对象

throw New Error ('false of json');

}).then(json => {

console.log(json);

}).catch(error => {

consolr.log(error);

})

// 使用!

fetch(request).then(function() { /\* handle response \*/ });

1.JSON格式

fetch('http://jartto.wang/test.json')

.then(function(response) {

// Convert to JSON

return response.json();

})

.then(function(result) {

console.log(result);

});

2.返回HTML/text

fetch('/next/page')

.then(function(response) {

return response.text();

}).then(function(text) {

// <!DOCTYPE ....

console.log(text);

});

3.发送form表单数据

//form data

var form = document.querySelector('form');

fetch('http://jartto.wang/submit', {

method: 'post',

body: new FormData(form)

});

//JSON

fetch('http://jartto.wang/submit-json', {

method: 'post',

body: JSON.stringify({

name: 'jartto',

blog: 'http://jartto.wang'

})

});

4.图片的处理

fetch('http://jartto.wang/logo.png')

.then(function(response) {

return response.blob();

})

.then(function(imageBlob) {

document.querySelector('img').src = URL.createObjectURL(imageBlob);

});

Redux

React 只是 DOM 的一个抽象层，并不是 Web 应用的完整解决方案。有两个方面，它没涉及。

* 代码结构
* 组件之间的通信

对于大型的复杂应用来说，这两方面恰恰是最关键的。因此，只用 React 没法写大型应用。

a. 需要回调通知state (等同于回调参数) -> action  
b. 需要根据回调处理 (等同于父级方法) -> reducer  
c. 需要state (等同于总状态) -> store

redux最大的作用：

原先react每个component都有state可以进行set，如果是独立组件，没有任何麻烦，但是如果需要组件之间的通信，即A的状态会改变B，在复杂应用中假如出现，会引起很大的麻烦。当然，在一般应用中，似乎难以体现。举个例子，在每个界面都会共用的组件，比如用户信息，state传递就会遇到挫折

对Redux来说只有这三个要素：  
a. action是纯声明式的数据结构，只提供事件的所有要素，不提供逻辑。  
b. reducer是一个匹配函数，action的发送是全局的：所有的reducer都可以捕捉到并匹配与自己相关与否，相关就拿走action中的要素进行逻辑处理，修改store中的状态，不相关就不对state做处理原样返回。  
c. store负责存储状态并可以被react api回调，发布action.

为了解决这个问题，2014年 Facebook 提出了 [Flux](http://www.ruanyifeng.com/blog/2016/01/flux.html) 架构的概念，引发了很多的实现。2015年，[Redux](https://github.com/reactjs/redux) 出现，将 Flux 与函数式编程结合一起，很短时间内就成为了最热门的前端架构。

"只有遇到 React 实在解决不了的问题，你才需要 Redux 。"

* 用户的使用方式复杂
* 不同身份的用户有不同的使用方式（比如普通用户和管理员）
* 多个用户之间可以协作
* 与服务器大量交互，或者使用了WebSocket
* View要从多个来源获取数据

上面这些情况才是 Redux 的适用场景：多交互、多数据源。

从组件角度看，如果你的应用有以下场景，可以考虑使用 Redux。

* 某个组件的状态，需要共享
* 某个状态需要在任何地方都可以拿到
* 一个组件需要改变全局状态
* 一个组件需要改变另一个组件的状态
* Redux 的设计思想很简单，就两句话。
* （1）Web 应用是一个状态机，视图与状态是一一对应的。
* （2）所有的状态，保存在一个对象里面。
* 请务必记住这两句话，下面就是详细解释。

### 3.1 Store

Store 就是保存数据的地方，你可以把它看成一个容器。整个应用只能有一个 Store。

Redux 提供createStore这个函数，用来生成 Store。

import { createStore } from 'redux';

const store = createStore(fn);

上面代码中，createStore函数接受另一个函数作为参数，返回新生成的 Store 对象。

### 3.2 State

Store对象包含所有数据。如果想得到某个时点的数据，就要对 Store 生成快照。这种时点的数据集合，就叫做 State。

当前时刻的 State，可以通过store.getState()拿到。

import { createStore } from 'redux';

const store = createStore(fn);

const state = store.getState();

Redux 规定， 一个 State 对应一个 View。只要 State 相同，View 就相同。你知道 State，就知道 View 是什么样，反之亦然。

### 3.3 Action

State 的变化，会导致 View 的变化。但是，用户接触不到 State，只能接触到 View。所以，State 的变化必须是 View 导致的。Action 就是 View 发出的通知，表示 State 应该要发生变化了。除了 type 字段外，action 对象的结构完全由你自己决定

Action 是一个对象。其中的type属性是必须的，表示 Action 的名称。其他属性可以自由设置，社区有一个[规范](https://github.com/acdlite/flux-standard-action)可以参考。

const action = {

type: 'ADD\_TODO',

payload: 'Learn Redux'

};

上面代码中，Action 的名称是ADD\_TODO，它携带的信息是字符串Learn Redux。

可以这样理解，Action 描述当前发生的事情。改变 State 的唯一办法，就是使用 Action。它会运送数据到 Store。一般通过store.dispatch()将action传到store

### 3.4 Action Creator

View 要发送多少种消息，就会有多少种 Action。如果都手写，会很麻烦。可以定义一个函数来生成 Action，这个函数就叫 Action Creator。

const ADD\_TODO = '添加 TODO';

function addTodo(text) {

return {

type: ADD\_TODO,

text

}

}

const action = addTodo('Learn Redux');

上面代码中，addTodo函数就是一个 Action Creator。

### 3.5 store.dispatch()

store.dispatch()是 View 发出 Action 的唯一方法。

import { createStore } from 'redux';

const store = createStore(fn);

store.dispatch({

type: 'ADD\_TODO',

payload: 'Learn Redux'

});

上面代码中，store.dispatch接受一个 Action 对象作为参数，将它发送出去。

结合 Action Creator，这段代码可以改写如下。

store.dispatch(addTodo('Learn Redux'));

3.6 Reducer

[Action](http://www.redux.org.cn/docs/basics/Actions.html) 只是描述了**有事情发生了**这一事实，并没有指明应用如何更新 state。而这正是 reducer 要做的事情。

保持 reducer 纯净非常重要。**永远不要**在 reducer 里做这些操作：

* 修改传入参数；
* 执行有副作用的操作，如 API 请求和路由跳转；
* 调用非纯函数，如 Date.now() 或 Math.random()。

Store 收到 Action 以后，必须给出一个新的 State，这样 View 才会发生变化。这种 State 的计算过程就叫做 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。

const reducer = function (state, action) {

// ...

return new\_state;

};

整个应用的初始状态，可以作为 State 的默认值。下面是一个实际的例子。

const defaultState = 0;

const reducer = (state = defaultState, action) => {

switch (action.type) {

case 'ADD':

return state + action.payload;

default:

return state;

}

};

const state = reducer(1, {

type: 'ADD',

payload: 2

});

上面代码中，reducer函数收到名为ADD的 Action 以后，就返回一个新的 State，作为加法的计算结果。其他运算的逻辑（比如减法），也可以根据 Action 的不同来实现。

实际应用中，Reducer 函数不用像上面这样手动调用，store.dispatch方法会触发 Reducer 的自动执行。为此，Store 需要知道 Reducer 函数，做法就是在生成 Store 的时候，将 Reducer 传入createStore方法。

import { createStore } from 'redux';

const store = createStore(reducer);

上面代码中，createStore接受 Reducer 作为参数，生成一个新的 Store。以后每当store.dispatch发送过来一个新的 Action，就会自动调用 Reducer，得到新的 State。

为什么这个函数叫做 Reducer 呢？因为它可以作为数组的reduce方法的参数。请看下面的例子，一系列 Action 对象按照顺序作为一个数组。

const actions = [

{ type: 'ADD', payload: 0 },

{ type: 'ADD', payload: 1 },

{ type: 'ADD', payload: 2 }

];

const total = actions.reduce(reducer, 0); // 3

上面代码中，数组actions表示依次有三个 Action，分别是加0、加1和加2。数组的reduce方法接受 Reducer 函数作为参数，就可以直接得到最终的状态3。

3.7 纯函数

Reducer 函数最重要的特征是，它是一个纯函数。也就是说，只要是同样的输入，必定得到同样的输出。

纯函数是函数式编程的概念，必须遵守以下一些约束。

* 不得改写参数
* 不能调用系统 I/O 的API
* 不能调用Date.now()或者Math.random()等不纯的方法，因为每次会得到不一样的结果

由于 Reducer 是纯函数，就可以保证同样的State，必定得到同样的 View。但也正因为这一点，Reducer 函数里面不能改变 State，必须返回一个全新的对象，请参考下面的写法。

// State 是一个对象

function reducer(state, action) {

return Object.assign({}, state, { thingToChange });

// 或者

return { ...state, ...newState };

}

// State 是一个数组

function reducer(state, action) {

return [...state, newItem];

}

最好把 State 对象设成只读。你没法改变它，要得到新的 State，唯一办法就是生成一个新对象。这样的好处是，任何时候，与某个 View 对应的 State 总是一个不变的对象。

3.8 store.subscribe()

Store 允许使用store.subscribe方法设置监听函数，一旦 State 发生变化，就自动执行这个函数。

import { createStore } from 'redux';

const store = createStore(reducer);

store.subscribe(listener);

显然，只要把 View 的更新函数（对于 React 项目，就是组件的render方法或setState方法）放入listen，就会实现 View 的自动渲染。

store.subscribe方法返回一个函数，调用这个函数就可以解除监听。

let unsubscribe = store.subscribe(() =>

console.log(store.getState())

);

unsubscribe();

## http://www.ruanyifeng.com/blogimg/asset/2016/bg2016091802.jpg

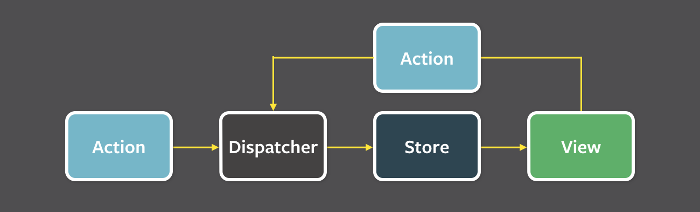
Flux

Flux 是一种架构思想，专门解决软件的结构问题。它跟[MVC 架构](http://www.ruanyifeng.com/blog/2007/11/mvc.html)是同一类东西，但是更加[简单和清晰](http://www.infoq.com/news/2014/05/facebook-mvc-flux)。

Flux存在多种实现（[至少15种](https://github.com/voronianski/flux-comparison)），本文采用的是[Facebook官方实现](https://github.com/facebook/flux)。

Flux将一个应用分成四个部分。

* **View**： 视图层
* **Action**（动作）：视图层发出的消息（比如mouseClick）
* **Dispatcher**（派发器）：用来接收Actions、执行回调函数
* **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



Flux 的最大特点，就是数据的"单向流动"。

1. 用户访问 View
2. View 发出用户的 Action
3. Dispatcher 收到 Action，要求 Store 进行相应的更新
4. Store 更新后，发出一个"change"事件
5. View 收到"change"事件后，更新页面

上面过程中，数据总是"单向流动"，任何相邻的部分都不会发生数据的"双向流动"。这保证了流程的清晰。

读到这里，你可能感到一头雾水，OK，这是正常的。接下来，我会详细讲解每一步

## View（第一部分）

请打开 Demo 的首页[index.jsx](https://github.com/ruanyf/extremely-simple-flux-demo/blob/master/index.jsx) ，你会看到只加载了一个组件。

// index.jsx

var React = require('react');

var ReactDOM = require('react-dom');

var MyButtonController = require('./components/MyButtonController');

ReactDOM.render(

<MyButtonController/>,

document.querySelector('#example')

);

上面代码中，你可能注意到了，组件的名字不是 MyButton，而是 MyButtonController。这是为什么？

这里，我采用的是 React 的 [controller view](http://blog.andrewray.me/the-reactjs-controller-view-pattern/) 模式。"controller view"组件只用来保存状态，然后将其转发给子组件。MyButtonController的[源码](https://github.com/ruanyf/extremely-simple-flux-demo/blob/master/components/MyButtonController.jsx)很简单。

// components/MyButtonController.jsx

var React = require('react');

var ButtonActions = require('../actions/ButtonActions');

var MyButton = require('./MyButton');

var MyButtonController = React.createClass({

createNewItem: function (event) {

ButtonActions.addNewItem('new item');

},

render: function() {

return <MyButton

onClick={this.createNewItem}

/>;

}

});

module.exports = MyButtonController;

上面代码中，MyButtonController将参数传给子组件MyButton。后者的[源码](https://github.com/ruanyf/extremely-simple-flux-demo/blob/master/components/MyButton.jsx)甚至更简单。

// components/MyButton.jsx

var React = require('react');

var MyButton = function(props) {

return <div>

<button onClick={props.onClick}>New Item</button>

</div>;

};

module.exports = MyButton;

上面代码中，你可以看到[MyButton](https://github.com/ruanyf/extremely-simple-flux-demo/blob/master/components/MyButton.jsx)是一个纯组件（即不含有任何状态），从而方便了测试和复用。这就是"controller view"模式的最大优点。

MyButton只有一个逻辑，就是一旦用户点击，就调用[this.createNewItem](https://github.com/ruanyf/extremely-simple-flux-demo/blob/master/components/MyButtonController.jsx#L27) 方法，向Dispatcher发出一个Action。

// components/MyButtonController.jsx

// ...

createNewItem: function (event) {

ButtonActions.addNewItem('new item');

}

上面代码中，调用createNewItem方法，会触发名为addNewItem的Action。

## Action

每个Action都是一个对象，包含一个actionType属性（说明动作的类型）和一些其他属性（用来传递数据）。

在这个Demo里面，[ButtonActions](https://github.com/ruanyf/extremely-simple-flux-demo/blob/master/actions/ButtonActions.js) 对象用于存放所有的Action。

// actions/ButtonActions.js

var AppDispatcher = require('../dispatcher/AppDispatcher');

var ButtonActions = {

addNewItem: function (text) {

AppDispatcher.dispatch({

actionType: 'ADD\_NEW\_ITEM',

text: text

});

},

};

上面代码中，ButtonActions.addNewItem方法使用AppDispatcher，把动作ADD\_NEW\_ITEM派发到Store。

## Dispatcher

Dispatcher 的作用是将 Action 派发到 Store。你可以把它看作一个路由器，负责在 View 和 Store 之间，建立 Action 的正确传递路线。注意，Dispatcher 只能有一个，而且是全局的。

Facebook官方的 [Dispatcher 实现](https://github.com/facebook/flux)输出一个类，你要写一个[AppDispatcher.js](https://github.com/ruanyf/extremely-simple-flux-demo/blob/master/dispatcher/AppDispatcher.js)，生成 Dispatcher 实例。

// dispatcher/AppDispatcher.js

var Dispatcher = require('flux').Dispatcher;

module.exports = new Dispatcher();

AppDispatcher.register()方法用来登记各种Action的回调函数。

// dispatcher/AppDispatcher.js

var ListStore = require('../stores/ListStore');

AppDispatcher.register(function (action) {

switch(action.actionType) {

case 'ADD\_NEW\_ITEM':

ListStore.addNewItemHandler(action.text);

ListStore.emitChange();

break;

default:

// no op

}

})

上面代码中，Dispatcher收到ADD\_NEW\_ITEM动作，就会执行回调函数，对ListStore进行操作。

记住，Dispatcher 只用来派发 Action，不应该有其他逻辑。

## Store

Store 保存整个应用的状态。它的角色有点像 MVC 架构之中的Model 。

在我们的 Demo 中，有一个[ListStore](https://github.com/ruanyf/extremely-simple-flux-demo/blob/master/stores/ListStore.js)，所有数据都存放在那里。

// stores/ListStore.js

var ListStore = {

items: [],

getAll: function() {

return this.items;

},

addNewItemHandler: function (text) {

this.items.push(text);

},

emitChange: function () {

this.emit('change');

}

};

module.exports = ListStore;

上面代码中，ListStore.items用来保存条目，ListStore.getAll()用来读取所有条目，ListStore.emitChange()用来发出一个"change"事件。

由于 Store 需要在变动后向 View 发送"change"事件，因此它必须实现事件接口。

// stores/ListStore.js

var EventEmitter = require('events').EventEmitter;

var assign = require('object-assign');

var ListStore = assign({}, EventEmitter.prototype, {

items: [],

getAll: function () {

return this.items;

},

addNewItemHandler: function (text) {

this.items.push(text);

},

emitChange: function () {

this.emit('change');

},

addChangeListener: function(callback) {

this.on('change', callback);

},

removeChangeListener: function(callback) {

this.removeListener('change', callback);

}

});

上面代码中，ListStore继承了EventEmitter.prototype，因此就能使用ListStore.on()和ListStore.emit()，来监听和触发事件了。

Store 更新后（this.addNewItemHandler()）发出事件（this.emitChange()），表明状态已经改变。 View 监听到这个事件，就可以查询新的状态，更新页面了。

**React-script**

以前通过手配置的webpack.config.js的内容react-scripts都已经帮我们做了