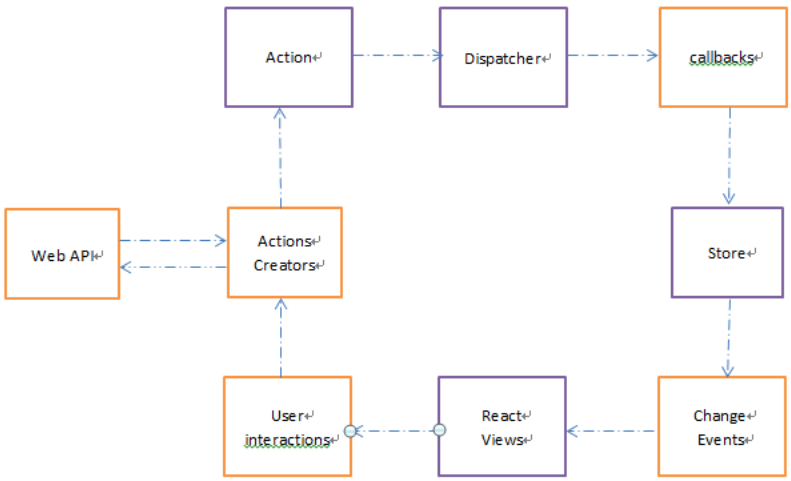
教程 0：introduction.js

缘由

学习 Redux 的时候，我才意识到，之前阅读关于 flux 的文章和个人经验让我积累了很多不正确的知识，我并不是指这些 flux 文章写的不好，只是我并没有正确理解罢了。到后来不过是申请各种 flux 框架（reflux,flummox,FB Flux），并将它们生拉硬套到我所读过的理论概念上来而已(actions/actions creators,store,dispatcher,etc)。

仅当我开始使用 Redux 时我才意识到，flux 比我想象中的简单。多亏了 Redux 的巧妙设计，移除了一些其他框架的“反模板特征”。此刻我认为，对比其他框架，Redux 是一个更好的选择。这也是我想和每个人分享对 Flux 概念理解（重点是使用 Redux）的原因。

相信你可能见过下面这张图，代表 flux 应用的单向数据流。



这篇教程里，我们会逐步介绍上面的概念。先把整个图表化整为零，试着理解它们为什么存在，发挥什么作用。最后理解各部分之后你会发现这张图表非常完美。

开始教程之前，让我说一些为什么 flux 存在和为什么我们需要它...

假设我们要创建一个 web 应用。想一想一般的 web 应用需要由什么组成？

1）模板（templates/html） = **View**

2)传递给视图的数据 =**Modes**

3)处理检索数据、整合视图、用户事件响应和数据修改的逻辑。=**Controller**

这是非常经典的 MVC。非常类似与 flux 的概念，只是表述上略有不同罢了。

Model 类似于 store

用户事件、数据修改和回调处理类似于“action creators”->action ->dispatcher ->callback

View 类似于 React 视图(或者 flux 关注的其它东西)

Flux 只是一个新的词汇吗？不全是，但确实涉及词汇，因为这些新词汇的引入我们现在可以更确切表达用术语重新组合的东西。举个例子，数据抓取是不是一个 action？鼠标点击是不是一个 action？输入框的 change 事件是不是一个 action，等等……然后我们都习惯于从我们的应用发出 action，只是调用它们方式不同。Flux 没有采用直接通过 actions 修改 Models 或者 Views 的处理方式，而是确

保所有 actions 先经过一个仓库中心(dispatcher)，然后派发给 stores，最后修改 stores 的观察者。

为了更清楚介绍 MVC 和 Flux 之间区别，我们将以一个经典的 use-case 在 MVC 应用为例：

经典 MVC 应用一般以下面的方式结束：

1）用户点击“A”按钮；

2）一个“A”按钮的 click 事件回调触发”A“模型的一个 change 事件；

3）一个“A”模型的 change 回调触发一个“B”模型的 change 事件；

4）一个“B”模型的 change 回调触发“B”视图的 change 事件，然后重新渲染。

这种方式寻找一个 bug 的来源成为一个非常大的挑战。这是因为每一个视图可以监视每一个模型，而每一个模型也可以监视其它模型，简单的说，数据可以从很多地方来，可以被很多的来源改变。

而当使用 Flux 和它的单向数据流，上面的例子会变为：

1）用户点击“A”按钮

2）一个“A“按钮回调触发一个 action，派发并产生一个 change 事件到 store“A“

3）因为这个 action 所有其它的 stores 也会修改，所有 store B 也会对同一个 action 有反应

4）由于 store A 和 store B 的 change 事件，View “B”产生

修改，执行重新渲染。

看看我们如何避免让 store A 直接于 store B 关联？每个 store 只能通过一个 action（或者 nothing）修改。并且一旦所有的 store 都已经响应这个 action，视图会在最后更新。最终，数据会向一个方向流动：

action ->store ->view -> action ->store ->view->action...

就从刚才开始使用的案例，让我们开始教程的 actions 和 action creators 部分。

前往下一教程：simple-action-creator.js

教程 1：simple-action-creator.js

我们开始在教程里探讨 actions，但是，到底什么才是“action creators”，它们怎么才能关联到“actions”上面？

其实非常简单，仅仅几行代码的事。

action creator 仅仅是一个函数……

```
var actionCreator = function() {
    // ...that creates an action (yeah the name action creator is pretty obvious now) and returns it
    return {
        type: 'AN_ACTION'
    }
}
```

就这些？是的。

但是有件事需要说明——action 的格式。这在 flux 是一种约定：action 是一个包含“type”属性的对象。这个类型允许对 action 进一步处理。当然，action 也可以包含任何你想传递数据的其它属性。

等会我们也会看到 action creator 实际上也会返回除 action 之外的一些东西，比如函数。这在异步 action 处理时非常有用（更多详见 dispatch-async-action.js）……

我们可以调用这个 action creator，然后获得一个期望的 action：

```
console.log(actionCreator())
```

输出：{type: ‘AN_ACTION ‘}

好了，它已经工作了，但还没用到其它地方……

我们需要做的是把这个 action 发送到某个地方，这样别人才会知道什么事情发生了，并做出相应的反应。

我们称这个过程为“派发一个 action”。

为了派发一个 action，我们需要……一个派发函数（dispatch function）,以及让感兴趣的人知道一个 action 发生了，我们需要一个机制来注册订阅用户。

截止目前，应用的数据流：

ActionCreator -> Action

关于 actions 和 action creator 信息详见

http://gaearon.github.io/redux/docs/recipes/ReducingBoilerplate.html

前往下一教程：simple-subscriber.js

教程 2：simple-subscriber.js

开始，先写一个空的订阅者：

```
var mySubscriber = function() {
    console.log('Something happened')
    // do something
}
```

很简单，不是吗？

它仍然没有在任何地方注册，但很快将会了。

一旦调用，它会执行设计的内容（此处为 console.log）。

```
mySubscriber()
```

到目前，应用的数据流向为：

ActionCreator ->Action ... Subscriber

前往下一教程：about-state- and –meet-redux.js

教程 3：about-state-and-meet-redux.js

有时，actions 不仅告知有事发生，而且也会告知有些数据需要被更新。

实际上在应用里实现这点是非常具有挑战性的。

1) 在应用里数据在其生命期内应该保存在哪里？

2) 我应该如何处理这些数据的修改？

3) 如何把这些修改传播给应用的其它部分？

这一切，Redux 来了。

简而言之，Redux（http://github.com/gaearon/redux）是一个“Javascript 应用的可预测状态的容器”。

让我们回顾上面的问题，用 Redux(flux)术语回答一下：

1) 在应用里数据在其生命期内应该保存在哪里？

你可以以你希望的方式保存它（js 对象，数组，不可变结构，……）

应用的数据被称为状态（state），由于这些数据是随时间变化的，是名副其实的应用状态。

但是你需要通过 Redux 来处理它们。

2) 我应该如何处理这些数据的修改？

使用 reducers 。

一个 reducer 其实是一个指定 actions 的订阅者。

一个 reducer 仅仅是一个接收应用当前 state 和 action 为参数，返

回一个新的修改过的 state 的函数。

3) 如何把这些修改传播给应用的其它部分？

使用订阅者传播状态的修改

Redux 已经为您把一切都糅合在一起了。

总结一下，Redux 会提供给你：

1）一个提交应用状态的地方

2）一种机制来订阅状态更新

3）一种机制来发布 actions 到 承担修改应用 state 的 modifiers。

Redux 实例称之为一个 store，并以如下方式创建：

```
/*
    import { createStore } from 'redux'
    var store = createStore()
*/
```

但是如果运行上面代码，会抛出如下错误

Error：Invariant Violation: Expected the reducer to be a function.

这是因为 createStore 期望至少是一个可以修改状态的函数。

接着继续……

```
import { createStore } from 'redux'

var store = createStore(() => {})
```

看起来比较完美……

前往下一教程：simple-reducer.js

教程 4：simple-**reducer**.js

现在我知道如何创建一个 Redux 实例来维护应用的状态。

我们将注意力集中在这些允许转换状态的 reducers 函数。

reducer vs store：

你可能注意到，简介里的 flux 图表里我们用了“Store”一词，而不是 Redux 期望的”Reducer”。因此如何准确的区分 Store 和 Reducer 之间的差异？

其实要比你想象的简单：Store 可以保留数据，而 Reducer 不能。

因此在传统的 flux 里面，stores 维护状态，而在 Redux 里面，每一次 reducer 被调用，它传递需要被更新的 state。这种意义上讲，Redux 的 stores 只是“无状态的 stores”,而且被重命名为 reducers。

如上所述，创建一个 Redux 实例之时你需要给它一个 reducer 函数……

```
import { createStore } from 'redux'

var store_0 = createStore(() => {})
```

所以一旦某个 action 发生时，Redux 可以调用这个函数。

让我们写个 log 到 reducer 里面：

```
var reducer = function (...args) {
    console.log('Reducer was called with args', args)
}

var store_1 = createStore(reducer)
```

输出：Reducer was called with args[undefined,{type:

‘@@redux/INIT’ }]

见到没有？即使我们没有发布任何 aciton，但 reducer 已经确实被调用了。

这是因为为了初始化应用的状态，Redux 实际上派发了一个初始 action（{type: ‘@@redux/INIT’ }）

一旦被调用，reducer 被传递这个参数：（state，action）

这点非常合乎逻辑，在应用初始化阶段，state 还没有被初始化，是“undefined “。

但是在 Redux 发送它的“init “action 之后，我们应用的 state 又是什么？

前往下一教程：get-state.js

教程 5：get-state.js

如何从 Redux 实例中获取状态？

```
import { createStore } from 'redux'

var reducer_0 = function (state, action) {
    console.log('reducer_0 was called with state', state, 'and action',
action)
}

var store_0 = createStore(reducer_0)
```

输出：reducer_0 was called with state undefined and action {type: '@@redux/INIT'}

为了获取状态，我们调用 getState：

```
console.log('store_0 state after initialization:', store_0.getState())
```

输出：store_0 state after initialization: undefined

难道初始化之后应用的状态仍然是 undefined？当然了，我们的 reducer 什么都没做。回忆一下我们在 about-state-and-meet-redux 里面如何描述的 reducer 的期望行为。

“一个 reducer 仅仅是一个接收应用当前 state 和 action 为参数，返回一个新的修改过的 state 的函数。“

我们的 reducer 此刻没有返回任何值，所以应用的 state 为 reducer（）所返回的值，为”undefined”。

若传递到 reducer 的状态为 undefined 时，把初始化状态发送给应用：

```
var reducer_1 = function (state, action) {
    console.log('reducer_1 was called with state', state, 'and action',
action)
    if (typeof state === 'undefined') {
        return {}
    }

    return state;
}

var store_1 = createStore(reducer_1)
```

输出：

reducer_1 was called with state undefined and action {type: '@@redux/INIT' }

```
console.log('store_1 state after initialization:', store_1.getState())
```

输出：Redux state after initialization:{}

不出所料，初始化后的 Redux 返回的 state 为{}

ES6 提供了一种非常干净的写法来实现上面功能：

```
var reducer_2 = function (state = {}, action) {
    console.log('reducer_2 was called with state', state, 'and action',
action)

    return state;
}

var store_2 = createStore(reducer_2)
```

输出：

reducer_2 was called with state {} and action {type: '@@redux/INIT' }

```
console.log('store_2 state after initialization:', store_2.getState())
```

输出：

Redux state after initialization:{}

你可能注意到在 reducer_2 的 state 参数中我们使用了缺省的参数，我们不再获得 undefined 作为 reducer 内部的 state 值了。

让我们回忆一下，reducer 只会在响应 action 派发时会被调用，让我们假定一个状态修改来响应一个”SAY_SOMETHING”类型的 action。

```
var reducer_3 = function (state = {}, action) {
    console.log('reducer_3 was called with state', state, 'and action', action)

    switch (action.type) {
        case 'SAY_SOMETHING':
            return {
                ...state,
                message: action.value
            }
        default:
            return state;
    }
}

var store_3 = createStore(reducer_3)
```

输出：

reducer_3 was called with state {} and action {type: ‘@@redux/INIT’ }

```
console.log('redux state after initialization:', store_3.getState())
```

输出：

Redux state after initialization : {}

目前 state 里面还没有什么新的东西出现，这是因为我们没有派发任

何 action 的缘故。但是这里有几个重要的东西需要重点关注一下：

0）我假设 action 包含一个 type 和一个 value 属性。type 属性是在 flux actions 里的一个约定，而 value 属性可以使任何东西。

1）你看到很多类型涉及到 switch 来响应 reducer 接收到的 action

2）使用 switch 时，永远别忘记“default：return state ”，因为如果没有，会让你的 reducer 返回 undefined。

3）注意我们是如何通过合并当前状态{message :action.value}返回一个新的状态，感谢 ES7 神奇的符号：{...state,message ： action.value}

4)注意这个 ES7 对象传播符号适合我们的例子，是因为它做了个{message：action.value}的影子备份。但是如果我们遇到更复杂或嵌套的数据结构，可以选择以下方式来处理状态的更新：

-使用 Immutable.js(http://facebook.github.io/immutable-js/)

-使用 Object.assign

-使用手动 merge

-或者其他方法

现在我们开始处理 reducer 里面的 actions，谈论多 reducers 并且如何合并。

前往下一教程：combine-reducers.js

教程 6：combine-reducer.js

现在开始掌握 reducer 到底是什么……

```
var reducer_0 = function (state = {}, action) {
    console.log('reducer_0 was called with state', state, 'and action',
action)

    switch (action.type) {
        case 'SAY_SOMETHING':
            return {
                ...state,
                message: action.value
            }
        default:
            return state;
    }
}
```

下一步之前，我们要搞清楚多 actions 情况下 reducer 会是什么样：

```
var reducer_1 = function (state = {}, action) {
    console.log('reducer_1 was called with state', state, 'and action',
action)

    switch (action.type) {
        case 'SAY_SOMETHING':
            return {
                ...state,
                message: action.value
            }
        case 'DO_SOMETHING':
            // ...
        case 'LEARN_SOMETHING':
            // ...
        case 'HEAR_SOMETHING':
            // ...
        case 'GO_SOMEWHERE':
            // ...
        // etc.
        default:
            return state;
    }
}
```

很明显，单个 reducer 函数容纳不了一个应用里的所有 actions 处理。（当然可以容纳，但是不利于维护……）

庆幸的是，Redux 不关心我们是否有一个 reducer 或者多个 reducer，它甚至帮助我们合并。

让我们声明两个 reducers：

```
var userReducer = function (state = {}, action) {
    console.log('userReducer was called with state', state, 'and action', action)

    switch (action.type) {
        // etc.
        default:
            return state;
    }
}
var itemsReducer = function (state = [], action) {
    console.log('itemsReducer was called with state', state, 'and action', action)

    switch (action.type) {
        // etc.
        default:
            return state;
    }
}
```

对于拥有多个 reducer 的方法，可以让单个 reducer 只负责处理应用的一小片状态。

但是 createStore 只期望有且仅有一个 reducer，怎么办呢？

那么如何才能合并 reducers 呢？以及如何告诉 Redux 单个 reducer 只负责处理的一小片状态？

其实很简单，利用 Redux 的 combineReducers 辅助函数。

combineReducers() 所做的只是生成一个函数，这个函数来调用你的一系列 reducer，每个 reducer **根据它们的 key 来筛选出 state 中的一部分数据并处理**，然后这个生成的函数将所有 reducer 的结果合并成一个大的对象。

长话短说，这里有个处理多个 reducers 的例子：

```
import { createStore, combineReducers } from 'redux'

var reducer = combineReducers({
    user: userReducer,
    items: itemsReducer
})
```

输出：

userReducer was called with state {} and action { type: '@@redux/INIT' }

userReducer was called with state {} and action { type: 'a.2.e.i.j.9.e.j.y.v.i' }

itemsReducer was called with state [] and action { type: '@@redux/INIT' }

itemsReducer was called with state [] and action { type: 'i.l.j.c.a.4.z.3.3.d.i' }

```
var store_0 = createStore(reducer)
```

输出：

userReducer was called with state {} and action { type: '@@redux/INIT' }

itemsReducer was called with state [] and action { type: '@@redux/INIT' }

如你所见，每一个 reducer 都在初始 action（@@redux/INIT）时被正确调用。但怎么又输出了其它 action？这是 combineReducers 的正常检查操作，以确保 reducer 一直返回一个非 " undefined" 状态。

谨记 combineReducers 的初始化 actions 的第一次调用效果等同于

随即 actions。

```
console.log('store_0 state after initialization:', store_0.getState())
```

输出：

store_0 state after initialization:{user:{},items:[]}

我们感兴趣的是 Redux 如何来处理这些分片。最终的 state 实际上是由 userReducer 分片和 itemsReducer 分片组成的哈希值。

```
{
    user: {}, // {} is the slice returned by our userReducer
    items: [] // [] is the slice returned by our itemsReducer
 }
```

现在我们对 reducers 如何工作加深了印象，下一步是让 actions 被派发下去，以及如何影响 Redux state。

前往下一教程：dispatch-action.js

教程 7：dispatch-action.js

到现在， 我们都是一直关注创建 reducer(s) ， 没有派发任何 actions。

我们将在采用前面教程 reducers 基础上来处理一些 actions：

```
var userReducer = function (state = {}, action) {
    console.log('userReducer was called with state', state, 'and action',
action)

    switch (action.type) {
        case 'SET_NAME':
            return {
                ...state,
                name: action.name
            }
        default:
            return state;
    }
}
var itemsReducer = function (state = [], action) {
    console.log('itemsReducer was called with state', state, 'and action',
action)

    switch (action.type) {
        case 'ADD_ITEM':
            return [
                ...state,
                action.item
            ]
        default:
            return state;
    }
}

import { createStore, combineReducers } from 'redux'
var reducer = combineReducers({
    user: userReducer,
    items: itemsReducer
})
var store_0 = createStore(reducer)
console.log('store_0 state after initialization:', store_0.getState())
```

输出：

store_0 state after initialization: { user: {}, items: [] }

让我们来派发第一个 action……记住我们在’simple-action-creator.js’里面说的：

“派发一个 action 我们需要……一个 dispatch 函数”。

dispatch 函数由 Redux 提供，并将 actions 传播给所有的 reducers 。可以通过 Redux 实例的 dispatch 属性访问 dispatch 函数。

为了派发一个 action，简单调用：

```
store_0.dispatch({
    type: 'AN_ACTION'
})
```

输出

userReducer was called with state {} and action { type: 'AN_ACTION' }

itemsReducer was called with state [] and action { type: 'AN_ACTION' }

所有的 reducer 都被调用了，但由于 reducers 里面都没有处理这个 action type，所以 state 没有被改变。

```
console.log('store_0 state after action AN_ACTION:', store_0.getState())
```

输出：

store_0 state after action AN_ACTION: { user: {}, items: [] }

等等！我们不是假定用一个 action creator 来发送一个 action 吗？实际上我们是可以用 actionCreator，但是它所返回的值也是 action

而已，并不会产生其他的东西。但是从后面负责应用考虑，基于 flux 理论还是应该使用 actionCreator：

```
var setNameActionCreator = function (name) {
    return {
        type: 'SET_NAME',
        name: name
    }
}

store_0.dispatch(setNameActionCreator('bob'))
```

输出：

userReducer was called with state {} and action { type: 'SET_NAME', name: 'bob' }

itemsReducer was called with state [] and action { type: 'SET_NAME', name: 'bob' }

```
console.log('store_0 state after action SET_NAME:', store_0.getState())
```

输出：

store_0 state after initialization: { user: { name: 'bob' }, items: [] }

我们只是处理了第一个 action，并且它修改了应用的 state！

但是这看起来有点简单，没有达到实际使用的程度。举个例子，如果我想在派发 action 之前在 action creator 里面做些异步的工作，该怎么办？我们将在下一节讨论“dispatch-async-action.js”。

截止目前，应用的数据流向为：

ActionCreator ->Action ->dispatcher ->reducer

前往下一节：dispatch-async-action.js

教程 8：dispatch-async-action-1.js

前面介绍了如何派发 actions，以及这些 actions 怎么通过 reducers 来修改应用的 state。

但是到现在我们只考虑到同步 actions，更准确来说是同步创建一个 action 的 action creators: 一旦被调用，会立即返回一个 action。

让我想想一个简单的异步例子：

1）用户点击按钮 " say Hi in 2 second"

2) 当按钮 "A" 被点击，我们希望 2 秒之后显示消息 "Hi"

3）再过 2 秒，用 "Hi" 这条消息更新视图。

当然，这条消息是我们应用状态的一部分，所以应该在 Redux store 里面保存它。但我们希望的是仅仅在 action creator 调用之后，再过 2 秒，才让 store 保存这条消息（因为一旦我们立即更新 state，所有状态更改的订阅者-比如视图-会立即修改并且在这 2 秒之间也会响应更新）。

先按前面的调用一个 action creator：

```
import { createStore, combineReducers } from 'redux'

var reducer = combineReducers({
    speaker: function (state = {}, action) {
        console.log('speaker was called with state', state, 'and action',
action)

        switch (action.type) {
```

```
            case 'SAY':
                return {
                    ...state,
                    message: action.message
                }
            default:
                return state;
        }
    }
})
var store_0 = createStore(reducer)

var sayActionCreator = function (message) {
    return {
        type: 'SAY',
        message
    }
}

console.log("\n", 'Running our normal action creator:', "\n")

console.log(new Date());
store_0.dispatch(sayActionCreator('Hi'))

console.log(new Date());
console.log('store_0 state after action SET_NAME:', store_0.getState())
```

输出:（略去初始化输出）

Sun Aug 02 2015 01:03:05 GMT+0200 (CEST)

speaker was called with state {} and action { type: 'SAY', message: 'Hi' }

Sun Aug 02 2015 01:03:05 GMT+0200 (CEST)

store_0 state after action SET_NAME: { speaker: { message: 'Hi' } }

……然后看到 store 已经被立即更新了。

但是我们想看到的是像这样的 action creator：

```
var asyncSayActionCreator_0 = function (message) {
    setTimeout(function () {
```

```
        return {
            type: 'SAY',
            message
        }
    }, 2000)
}
```

但是这样我们的 action creator 返回的不是一个 action，这样它会返回 “undefined”。因此这不是我们寻找的解决方案。

这里有个技巧：我们返回一个函数来代替返回一个 action。这个函数就是负责派发这个 action 的那个函数。但是如果我们想要这个函数能够派发这个 action 到它应该被给定的 dispatch 函数那里。应该像下面的这样：

```
var asyncSayActionCreator_1 = function (message) {
    return function (dispatch) {
        setTimeout(function () {
            dispatch({
                type: 'SAY',
                message
            })
        }, 2000)
    }
}
```

你会再一次发现我们的 action creator 返回的不是一个 action，而是一个函数。

因此这便增加了 reducers 不知道如何处理的概率。

让我们试着找出到底发生了什么？

前往下一节：dispatch_action_creator-2.js

教程 9：dispatch_action_creator-2.js

让我们运行 *dispatch-async-action-1.js* 教程里的第一个异步 action creator。

```
import { createStore, combineReducers } from 'redux'

var reducer = combineReducers({
    speaker: function (state = {}, action) {
        console.log('speaker was called with state', state, 'and action',
action)

        switch (action.type) {
            case 'SAY':
                return {
                    ...state,
                    message: action.message
                }
            default:
                return state;
        }
    }
})
var store_0 = createStore(reducer)

var asyncSayActionCreator_1 = function (message) {
    return function (dispatch) {
        setTimeout(function () {
            dispatch({
                type: 'SAY',
                message
            })
        }, 2000)
    }
}

console.log("\n", 'Running our async action creator:', "\n")
store_0.dispatch(asyncSayActionCreator_1('Hi'))
```

输出：

```
...

/Users/classtar/Codes/redux-tutorial/node_modules/redux/node_modul
```

es/invariant/invariant.js:51

throw error;

^

Error: Invariant Violation: Actions must be plain objects. Use custom middleware for async actions.

...

看起来我们的函数没有运行到 reducers。但是 Redux 友善地提示我们："异步 actions 需要使用经典中间件"。看起来我们方向对了，但究竟什么是"中间件"？

可以向你保证，asyncSayActionCreator_1 这个 action creator 写法正确，而且只要使用合适的中间件，也会按预想运行。

前往下一节：middleware.js

教程 10：middleware.js

在 dispatch-async-action-2.js 一节我们留下了一个新的概念：“中间件”。中间件是用来设想解决异步 action 处理的。那么中间件到底是什么呢？

一般而言，中间件是介于应用的 A 部分和 B 部分的中间层， 所有 A 发出的信息经过中间件过滤，然后才传递到 B。

正常应该是：

A -----> B

加上中间件后变成

A ---> middleware 1 ---> middleware 2 --> ... ---> B

中间件如何才能在 Redux 上下文里面起作用？猛一看，我们的异步 action creator 返回的函数无法被 Redux 自身处理，但是如果我们在 action creator 和 reducers 之间添加一个中间件，经过中间件的转换，可以和 Redux 任意适配。

action--> dispatcher --> middleware 1--> middleware 2--> reducers

这样，每次派发一个 action，我们的中间件都会被调用。它会有效地帮助我们的 action creator 派发真实的 action。

在 Redux 里，中间件是必须符合特定规则和遵循严格结构的函数集：

```
var anyMiddleware = function ({ dispatch, getState }) {
    return function(next) {
```

```
        return function (action) {
            // your middleware-specific code goes here
        }
    }
}
```

如上所述，一个中间件有 3 个嵌套的函数组成：

1）第一层提供了 dispatch 函数和 getState 函数（如果中间件或你的 action creator 需要在 state 中读取数据）到其它两层；

2）第二层提供了 next 函数，允许你将输入变换处理后，移交给下一个中间件或 Redux（因此 Redux 最终可以调用所有的 reducers）.

3)第三层提供派发或从前一中间件接收来的 action。而且可以触发下一个中间件或者用合适的方式处理此 action。

我们创建一个 thunk middleware 来处理我们的异步 action creator，代码提供在此：https://github.com/gaearon/redux-thunk。

这是代码（为了可读性，转换成了 es5）：

```
var thunkMiddleware = function ({ dispatch, getState }) {
    // console.log('Enter thunkMiddleware');
    return function(next) {
        // console.log('Function "next" provided:', next);
        return function (action) {
            // console.log('Handling action:', action);
            return typeof action === 'function' ?
                action(dispatch, getState) :
                next(action)
        }
    }
}
```

为了告诉 Redux 使用了一个或多个中间件，必须引入 Redux 的辅助

函数：applyMiddleware。

“applyMiddleware” 以所有中间件为参数 ,返回一个可以被 Redux createStore 调用的函数。当这个最后函数被注入 ,它会创建一个 “更高阶的 store 运用中间件实现 store 的派发 “。

https://github.com/gaearon/redux/blob/v1.0.0-rc/src/utils/applyMiddleware.js

这里整合了一个中间件到 Redux store：

```
import { createStore, combineReducers, applyMiddleware } from 'redux'

const finalCreateStore = applyMiddleware(thunkMiddleware)(createStore)
```

对于多个中间件，书写成：

applyMiddleware(middleware1, middleware2, ...)(createStore)

```
var reducer = combineReducers({
    speaker: function (state = {}, action) {
        console.log('speaker was called with state', state, 'and action',
action)

        switch (action.type) {
            case 'SAY':
                return {
                    ...state,
                    message: action.message
                }
            default:
                return state
        }
    }
})

const store_0 = finalCreateStore(reducer)
```

输出：

speaker was called with state {} and action { type: '@@redux/INIT' }

speaker was called with state {} and action { type: 'v.b.k.7.s.e.8.9.f.6.r' }

speaker was called with state {} and action { type: '@@redux/INIT' }

现在我们有了中间件处理后的 store 实例，下面去派发这个异步 action：

```
var asyncSayActionCreator_1 = function (message) {
    return function (dispatch) {
        setTimeout(function () {
            console.log(new Date(), 'Dispatch action now:')
            dispatch({
                type: 'SAY',
                message
            })
        }, 2000)
    }
}

console.log("\n", new Date(), 'Running our async action creator:', "\n")

store_0.dispatch(asyncSayActionCreator_1('Hi'))
```

输出：

Mon Aug 03 2015 00:01:20 GMT+0200 (CEST) Running our async action creator:

Mon Aug 03 2015 00:01:22 GMT+0200 (CEST) 'Dispatch action now:'

speaker was called with state {} and action { type: 'SAY', message: 'Hi' }

我们的 action 在调用异步 action creator 2 秒后被正确派发！

为了你的好奇心，这里是一个中间件如何输出所有被派发 actions 的日志的例子：

```
function logMiddleware ({ dispatch, getState }) {
    return function(next) {
        return function (action) {
            console.log('logMiddleware action received:', action)
            return next(action)
```

```
        }
    }
}
```

下面的中间件作用是丢弃所有经过的 actions（用处不大，但是略添加些逻辑便可以在向下一个中间件或 Redux 传递时，有选择性舍去一些 actons）：

```
function discardMiddleware ({ dispatch, getState }) {
    return function(next) {
        return function (action) {
            console.log('discardMiddleware action received:', action)
        }
    }
}
```

试着用 logMiddleware 或者 discardMiddleware 修改上面的 finalCreateStore 调用,看看会出现什么情况……

比如：

const finalCreateStore = applyMiddleware(discardMiddleware, thunkMiddleware)(createStore)

会让你的 actions 永远不会到达 thunkMiddleware 甚至是减少 reducers。

见 http://gaearon.github.io/redux/docs/introduction/Ecosystem.html 的 Middlewares 章节，有更多的 middleware 例子。

让我们总结到目前学到的内容：

1）我们知道如何写 action 和 action creators

2）我们知道如何派发 actions

3）知道如何处理经典 actions，如异步 actions(多亏 middlewares)

Flux 应用闭环里唯一遗忘的部分是即将开始的 state 更新部分（通过重渲染组件）。

那么我们如何来订阅 Redux store 的更新呢？

前往下一节：state-subscriber.js

教程 11: state-subscriber.js

我们接近完成一个 Flux 闭环，但是遗忘了一个非常重要了部分。

没有它，store 发生改变时无法更新我们的视图。

幸亏，有一个非常简单的方法可以“监视“Redux 里的 store 更新：

```
store.subscribe(function() {
    // retrieve latest store state here
    // Ex:
    console.log(store.getState());
})
```

我们试试吧：

```
import { createStore, combineReducers } from 'redux'

var itemsReducer = function (state = [], action) {
    console.log('itemsReducer was called with state', state, 'and action',
action)

    switch (action.type) {
        case 'ADD_ITEM':
            return [
                ...state,
                action.item
            ]
        default:
            return state;
    }
}

var reducer = combineReducers({ items: itemsReducer })
var store_0 = createStore(reducer)

store_0.subscribe(function() {
    console.log('store_0 has been updated. Latest store state:',
store_0.getState());
    // Update your views here
})
```

```
var addItemActionCreator = function (item) {
    return {
        type: 'ADD_ITEM',
        item: item
    }
}

store_0.dispatch(addItemActionCreator({ id: 1234, description:
'anything' }))
```

输出：

store_0 has been updated. Latest store state: { items: [{ id: 1234, description: 'anything' }] }

我们的订阅回调被正确调用，以及我们的 store 现在包含添加的新的 item。

理论上说我们可以在这结束了。我们的 Flux 环以及闭合了，我们理解 Flux 的所有概念，在眼里 Flux 看起来不再是那么神秘了。但是诚实地讲，这远远不够，在最后的那个例子里，我们为了展示最简单形式的 Flux 概念，去掉了很多东西：

——为什么我们的订阅者回调没有接收 state 作为参数？

——由于没有接收新的 state，势必要利用闭塞的 store(store_0)，导致这种方案在实际多模块应用里不能接受。

——实际中怎样更新我们的视图？

——怎样从状态更新取消订阅？

——更广义上讲，如何将 Redux 与 React 进行整合？

现在我们进入一个更“Redux 深入 React”的领域。

有一点需要明白，Redux 不是非要绑定到 React。它实际上是“Javascript 应用的可预测状态容器”，你可以在很多场合用它，React 应用只是其中一项。

鉴于此，我们推荐使用 react-redux(https://github.com/gaearon/react-redux)。react-redux 已经提前将 Redux 进行了整合，处理了所有的绑定，方便我们在 React 里面使用 Redux。

返回到“订阅”的例子……准确点说，为什么我们要有这个看似简单的订阅函数，同时也看起来没有提供足够的特点？

简便性是它的作用。Redux，拥有目前最小的 API（包含“subscribe”）具有高度可扩展性，并允许创建像 Redux DevTools 这样疯狂的产品。

但最后，我们仍然需要一个“更好”的 API，订阅我们的 store 变化。这就是 redux-react 带给我们的：一个允许我们跨越原始 Redux 订阅机制和我们的开发期望之间鸿沟的 API。最后，你无须直接使用“订阅”。你只需要使用绑定（如“provide”或“connect”）替代即可，这样会隐藏“subscribe”具体实现方法。

这样，“订阅”方法会为你通过一个更高级 API 处理接入 redux 状态的细节。

你现在只需要处理绑定，展示如何简单把组件写入 Redux 状态里里面。

前往下一教程：provide-and-connect.js

教程 12: Provider-and-connect.js

这是教程最后一节，展示如何把 Redux 和 React 绑定一起。

为运行这个例子，你需要一个浏览器：

这个例子的解释部分都在源码 ./12_src/src/里面。

先把下面几行读完，然后从./12_src/src/server.js 开始。

为了创建我们的 React 应用，并且发送服务到浏览器，我们将使用：

— 一个非常简单的 node HTTP 服务器

— Webpack 打包应用

— Webpack Dev Server 从 node 服务器发送 JS 文件服务

— React Hot Loader

此处我不会探讨 Webpack 和 React Hot Loader 构建细节，文档部分解释非常清楚。

```
import webpackDevServer from './12_src/src/webpack-dev-server'
```

我们请求应用主服务器，执行开始：

```
import server from './12_src/src/server'
```

如果端口 5050 已被占用，更改端口。

如果端口等于 X，我们将利用 X 为服务器端口，而 X+1 为 webpack-dev-server 的端口

```
const port = 5050
```

开始我们的 webpack dev server……

```
webpackDevServer.listen(port)
```

……以及主应用服务器

```
server.listen(port)
console.log(`Server is listening on 127.0.0.1:${port}`)
```

前往 12_src/src/server.js……

12_src/src/server.js 部分：

这里！那么你已经在 React 应用里面尝试 Redux 了吗？

我们需要关注 react-redux 两个主要绑定的使用：

1）Provider 组件

2）连接装饰器

```
// But before we get to that, let's see the basic setup of this application
and how it
// will be served to browser...

// We won't use Express (http://expressjs.com/) in this app since we don't
really need
// it to serve a simple html page.

// "http" module will be used to create the http server
import http from 'http'
import React from 'react'

// We create our main application server here. It will serve the same page
on all URIs
// so you won't find any route specific logic below (except for rejecting
favicon request)
var server = http.createServer(function(req, res) {

  // Forget this, it's just to avoid serving anything when browser
automatically
  // requests favicon (if not, this server would send back an html page).
  if (req.url.match('favicon.ico')) {
    return res.end()
  }

  // And of course, here is our Application HTML that we're sending back to
the browser.
  // Nothing special here except the URI of our application JS bundle that
```

```
points to our
  // webpack dev server (located at http://localhost:5051)
  res.write(
    `<!DOCTYPE html>
    <html>
      <head>
        <meta charSet="utf-8" />
      </head>
      <body>
        <div id="app-wrapper"></div>
        <script type="text/javascript"
src="http://localhost:5051/static/bundle.js"></script>
      </body>
    </html>`
  )

  res.end()
})

export default server

// Go to ./index.jsx, where our app is initialized. For those of you who
are not familiar with webpack,
// index.jsx is defined as the entry point (the first file) of our JS bundle
(in 12_src/webpack.config.js)
// and is automatically executed when the JS bundle is loaded in our browser.
```