**Redux cho người mới bắt đầu - Part 1 Introduction**

[Editors' Choice](https://viblo.asia/editors-choice) [redux](https://viblo.asia/tags/redux)[ReactJS](https://viblo.asia/tags/reactjs)



Hiện nay [Reactjs](https://facebook.github.io/react/) là một thư viện mạnh mẽ khá phổ biến. Khi làm việc với React hay các dự án ứng dụng Single Page nói chung, có một vấn đề khá đau đầu là làm sao quản lý được trạng thái của ứng dụng đó. Sau khi xem qua giới thiệu về một số thư viện hỗ trợ công việc này, mình đã quyết định về với đội của [Redux](https://github.com/reactjs/redux/).

Trong loạt bài viết **Redux cho người mới bắt đầu** mình sẽ cùng các bạn tìm hiểu những khái niệm cơ bản và demo một ứng dụng nhỏ với Redux.

Trong phạm vi bài này mình sẽ giới thiệu về Redux còn code thì xin phép để bài sau (yaoming)

**Chuẩn bị**

Trước khi bắt đầu có một số khái niệm nên tìm hiểu qua:

* [Flux](https://viblo.asia/theme/en/react_js/flux_)
* [Functional Programming](https://viblo.asia/categories/functionalprogramming): những khái niệm cơ bản như pure function, immuable...

**Redux đã ra đời như thế nào ?**

Năm Quý Tị (2013), Facebook gia tộc bố cáo thiên hạ rằng Ăn Gô La đại pháp (Angular) của Google gia tộc chậm chạp, nặng nề, cho xuất thế một bộ chiêu thức gọi là Rối An Tâm Pháp (React).

Thế nhưng Rối An Tâm Pháp lại chỉ là một bộ tâm pháp cường thân kiện thể, không thể dùng để rèn luyện nội công (chỉ là một library để render view). Do đó, không lâu sau Facebook gia tộc tiếp tục cho ra đời một bộ tâm pháp cơ bản (kiến trúc thiết kế) và một công pháp cùng tên là Phờ Lắc thần công (Flux). Nghe đồn Rối An Tâm Pháp và Phờ Lắc Thần Công kết hợp lại sẽ thành tuyệt học dời non lấp bể, không gì không làm đc. Nhân sĩ giang hồ (coder) vốn nhẹ dạ cả tin lại rủ nhau tu luyện.

Phờ Lắc thần công rối rắm khó học, nhân sĩ 10 phần học thì 4-5 phần tẩu hỏa nhập ma, phần còn lại cũng trầy da tróc vẩy mà công lực cũng chẳng được như lời Facebook gia tộc quảng cáo.

Bấy giờ có một nhân sĩ giang hồ tự là Đan (Dan Abramov), đang tu luyện đồng thời Phờ lắc thần công và Ê La thần công (Elm) mới nhận ra rằng hai môn võ công có nhiều điểm chung, chỉ khác chiêu thức, Đan bèn nảy ra ý định hợp nhất hai môn này lại. Không lâu sau (5/2015), Đan cho xuất thế một bộ công pháp mang tên Rì Đắt thần công (Redux), mang ưu điểm của cả hai môn võ công đồng thời loại bỏ những phức tạp dư thừa của Phờ Lắc thần công.

Nhân sĩ giang hồ nghe vậy mừng lắm, thế là lại kéo nhau đi học Rì Đắt, còn Đan thì được Facebook gia tộc mời về làm tộc nhân.

Trích "JavaScript Lược Sử Giang Hồ" có sửa đổi bổ sung (yaoming)

**Redux là gì ?**

Redux js là một thư viện Javascript giúp tạo ra thành một lớp quản lý trạng thái của ứng dụng.

Redux được xây dựng dựa trên nền tảng tư tưởng của ngôn ngữ Elm và kiến trúc Flux do Facebook giới thiệu.

Do vậy Redux thường là bộ đôi kết hợp hoàn hảo với React. Tuy nhiên hoàn toàn có thể sử dụng với các framework khác như Angular, Angular2, Backbone, Falcor, Deku.

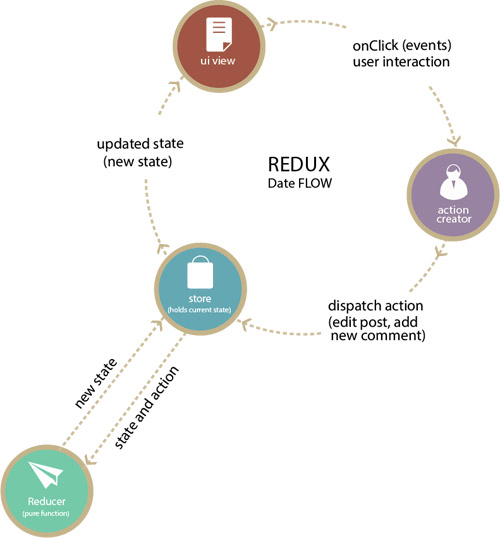
**Note:** Reduxjs là một thư viện Javascript khác với Redux Framework là một WordPress Framework.

**Nguyên lí**

Redux được xây dựng dựa trên 3 nguyên lý:

* Nguồn dữ liệu tin cậy duy nhất: State của toàn bộ ứng được chứa trong một object tree nằm trong Store duy nhất
* Trạng thái chỉ được phép đọc: Cách duy nhất để thay đổi State của ứng dụng là phát một Action (là 1 object mô tả những gì xảy ra)
* Thay đổi chỉ bằng hàm thuần túy: Để chỉ ra cách mà State được biến đổi bởi Action chúng ta dùng các pure function gọi là Reducer

**Cấu trúc**



Về cơ bản Redux có 4 thành phần như sau:

* Action: Là nơi mang các thông tin dùng để gửi từ ứng dụng đến Store. Các thông tin này là 1 object mô tả những gì đã xảy ra.

export const INCREASE = 'INCREASE'

export function increase() {

return {

type: INCREASE

}

}

* Reducer: Là nơi xác định State thay đổi như thế nào.

export default function counterApp (state = initialState, action) {

switch (action.type) {

case INCREASE:

return {

increase: ++state.increase,

decrease: state.decrease

}

case DECREASE:

return {

increase: state.increase,

decrease: ++state.decrease

}

default:

return state

}

}

* Store: Là nơi quản lý State, cho phép truy cập State qua getState(), update State qua dispatch(action), đăng kí listener qua subscribe(listener).

import { createStore } from 'redux'

import counterApp from './reducers'

let store = createStore(counterApp)

* View: Hiển thị dữ liệu được cung cấp bởi Store

**Data flow**

Trong một ứng dụng thực tế, chúng ta có nhiều thành phần hơn, và chúng ta sẽ khó mà hiểu hết được các hoạt động của Redux nếu chỉ nhìn vào 4 thành phần cơ bản trên. Chúng ta sẽ tìm hiểu kĩ hơn thông qua các hình tượng minh họa sau.

**Các nhân vật**

Tưởng tượng ứng dụng của chúng ta là một văn phòng, có các thành viên làm việc với nhau để giải quyết công việc chung. Cùng điểm qua các nhân vật trong phòng trước khi xem các họ tương tác với nhau để giải quyết công việc:

**The Action creators**

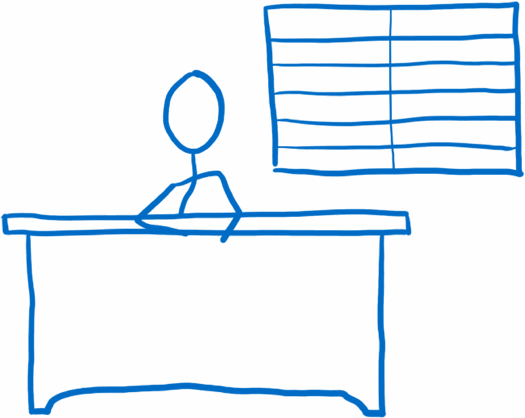


Đầu tiên là The Action Creators. Anh ta giữ nhiệm vụ tạo ra các Action, là bước đầu tiên trong luồng mà các thay đổi và tương tác đều đi qua. Bất cứ khi nào trạng thái của app hay là render của view thay đổi thì đầu tiên là một hành động sẽ được tạo ra.

Hãy hình dung anh này như một anh chàng đánh máy, anh ta biết bạn cần truyền đạt điều gì và cần phải đánh ra văn bản theo định dạng nào cho mọi người đều hiểu được.

The Action Creator tạo ra một action là formated object chứa type và thông tin của action đó. Type thường sẽ là một hằng số được định nghĩa trước, kiểu như INCREASE hay DECREASE.

**The Store**



Tiếp theo là The Store. Hãy hình dung đây là ông sếp đầy quyền lực, toàn bộ các thao tác với State tree (getState, updateState, registerListener...) đều do ông này quản lý.

Cái ông này làm nhiều việc nhỉ (?) Không thực ra ông ấy chỉ quản lý trạng thái của State tree thôi. Khi nhận được Action ông ấy sẽ đi hỏi The reducers xem State sẽ thay đổi ra sao chứ không tự làm.

Nếu đã biết qua Flux thì hẳn là bạn nhận ra ông này nhận trực tiếp Action mà không thông qua Dispatcher. Đó là bởi vì trong Redux, nhờ tiếp nhận tư tưởng của Functional Programming, ông sếp này tự biết cách hiểu Action và tự điều phối nó (dispatch()), nên không cần thuê thêm anh Dispatcher.

**The Reducers**



Tiếp theo là The Reducers. Khi The Store muốn biết State thay đổi như thế nào, ông ấy sẽ gọi cho The Reducers. Ở đây có một ông là Root Reducer nữa, ông này sẽ chịu trách nhiệm cắt ra State cần thay đổi dựa trên keys mà The Store gửi cho và đưa nó cho Reducer biết cách xử lý.

Hãy hình dung đây là một nhóm các thanh niên cuồng Photocopy (yaoming). Họ không thích làm rối tung những thứ họ được đưa cho, nên họ tạo ra bản sao của chúng và thực hiện thay đổi trên bản sao đó.

Đây là một trong những ý tưởng quan trọng của Redux. State không được thao tác trực tiếp. Thay vào đó, mỗi phần được sao chép và sau đó tất cả các phần được kết hợp thành một đối tượng trạng thái mới.

Các reducer gửi bản sao của chúng cho root reducer, và root reducer sẽ ghép các bản sao với nhau để tạo State mới. Sau đó, root reducer sẽ gửi các State mới trở lại Store và Store sẽ sử dụng nó như State chính thức mới.

Nếu ta có một ứng dụng nhỏ, ta chỉ có thể có một reducer làm cho một bản sao của toàn bộ State và và thay đổi nó. Hoặc nếu ta có một ứng dụng lớn, ta có thể có một Cây Reducers (Tree of Reducers) . Đây là một sự khác biệt giữa Flux và Redux. Trong Flux, các Store không nhất thiết phải kết nối với nhau và có một cấu trúc phẳng. Trong Redux, Reducers là một hệ thống cấp bậc, hệ thống cấp bậc này có thể có nhiều mức độ cần thiết, giống như các hệ thống cấp bậc component.

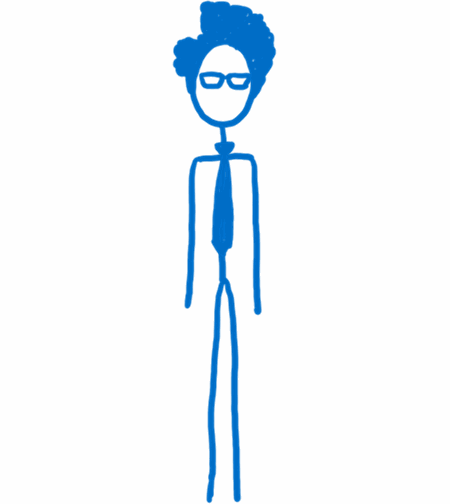
**The views: smart and dumb components**



Trong Redux có 2 khái khái niệm: smart and dumb components.

* Smart component: có thể gọi là containers
  + Hãy hình dung đây là anh quản lý của nhóm nhỏ, anh ta phụ trách các Action. Khi các thành viên dưới anh ta (dumb components) cần phát 1 action, anh ta sẽ gửi action cho các thành viên dưới dạng props, các thành viên chỉ cần coi đó là các callback mà không quan tâm nó là cái gì.
  + Anh ta không thích ăn diện (không có css).
  + Khi có việc cần thay đổi (DOM) thì anh ta sẽ sắp xếp các thành viên dưới làm chứ hiếm khi tự làm.
* Dumb components: có thể gọi là components
  + Hãy hình dung đây là mấy thanh niên học việc, thiên lôi chỉ đâu đánh đấy.
  + Mấy thanh niên này không phụ thuộc trực tiếp vào các Action, vì được anh quản lý đưa cho rồi. Điều này có nghĩa là mấy thanh niên này có thể đưa sang bộ phận khác làm cũng đc, miễn là có anh quản lý đưa "hàng" cho xài.
  + Mấy thanh niên này thì đẹp trai, tóc tai vuốt vuốt các thứ (có css riêng), nhưng đôi khi bị cấp trên bắt mặc theo ý sếp (nhận props style) thì vẫn phải chịu.

**The view layer binding**



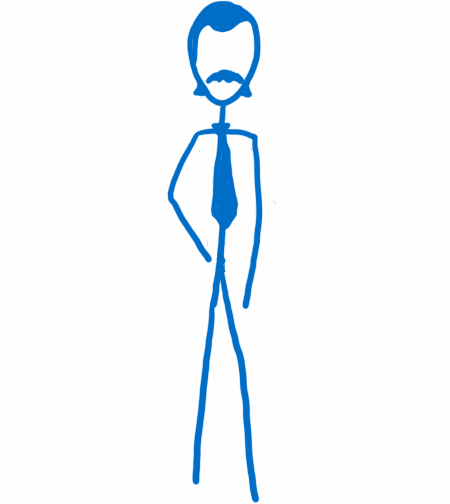
Để The Store giao tiếp đc với The views, chúng ta cần một ai đó kết nối họ lại với nhau, và chúng ta có The view layer binding, với React anh ta tên là react-redux.

Hãy hình dung anh này là nhân viên IT chịu trách nhiệm đảm bảo mạng trong công ty thông suốt để các components kết nối được với Store. Đồng thời cũng quản lý một đống thông tin kỹ thuật mà trong văn phòng chả ai hiểu (yaoming)

The view layer binding cung cấp 3 khái niệm:

* The Provider component: là thành phần bao quanh components tree. Giúp các components con kết nối với Store dễ dàng thông qua connect()
* connect() : là function cung cấp bởi The view layer binding như react-redux. Nếu một component muốn nhận được update State, nó phải tự bao lại bằng connect(). Sau đó, connect function sẽ thiết lập tất cả các hệ thống liên kết cho nó, bằng cách sử dụng selector
* selector : Đây là function mà bạn viết. Nó chỉ rõ phần nào của State mà component cần như properties.

**The root component**



Cuối cùng là The root component. Tất cả các React app đều có The root component. Là những component cao nhất của hệ thống component. Ở Redux thì nó đảm nhận nhiều trách nhiệm hơn.

Hãy hình dung đây là một giám đốc C-level (CEO, COO....). Ông giám đốc này sẽ tạo ra The Store và chỉ định The Reducers nào được sử dụng, tập hợp The view layer binding cùng với The views.

Sau khi chỉ định và tập hợp các thành phần trong team, giám đốc của chúng ta sẽ để cho các bộ phận bên dưới tự hoạt động.

**Phối hợp giữa các nhân vật kể trên**

Qua hết các nhân vật rồi, giờ cùng xem họ tương tác với nhau như thế nào để giải quyết công việc

**Setup**

Các bộ phận cần được nối với nhau. Việc này xảy ra lần đầu vào app.

**1. Bảo ông Store sẵn sàng**: ông giám đốc Root component tạo ra Store, chỉ cho ông Store dùng Root Reducer nào thông qua createStore(). Ông Root Reducer thì đã có team sẵn rồi, được tập hợp lại thông qua combineReducers()



**2. Chuẩn bị liên lạc giữa ông Store và các bộ phận khác.**

Root component bao các subcomponents với provider component (The view layer binding) và tạo kết nối giữa Store với các Provider.

Provider tạo ra 1 mạng cơ bản để cập nhật các components. Smart Components kết nối vào mạng bằng connect(), điều này đảm bảo họ nhận được cập nhật State.



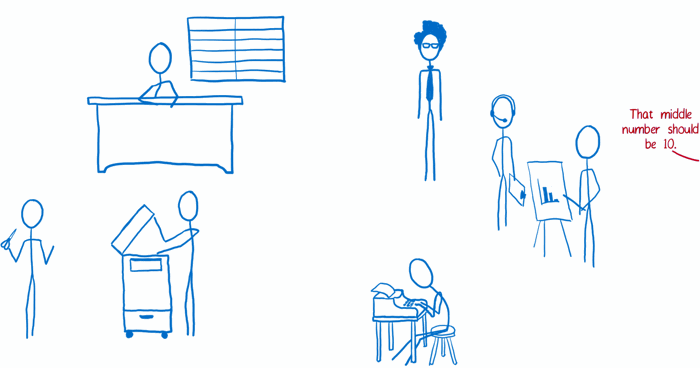
**3. Chuẩn bị các actions callback**

Để các Dump Components làm việc với Action dễ dàng hơn, các Smart Components có thể chuẩn bị các action callback thông qua bindActionCreators(). Bằng cách này, họ chỉ có cần truyền các callback cho Dump Components. Các Actions sẽ được tự động gửi đi sau khi nó được định dạng.

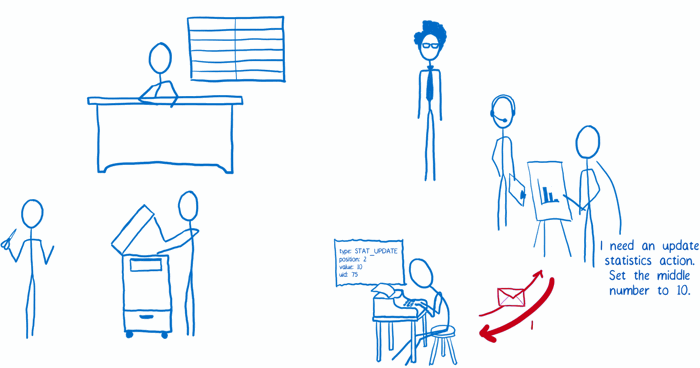


**Flow**

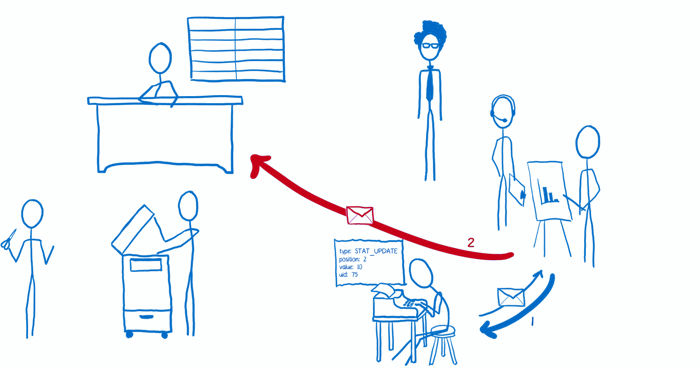
Sau phần setup thì web app đã sẵn sàng nhận input từ người dùng. Khi này trigger một action bằng cách để người dùng tạo ra một sự thay đổi.



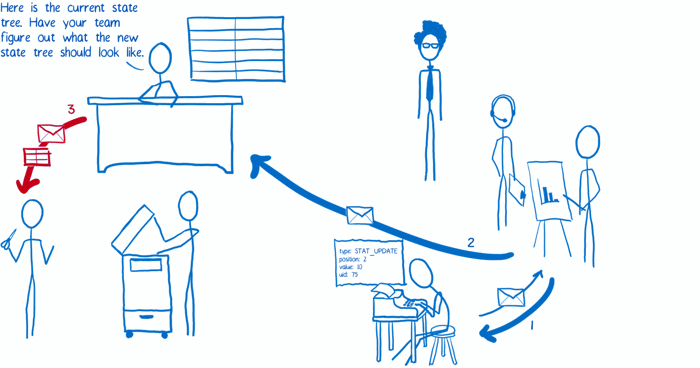
1. The View yêu cầu 1 action. Action Creator định dạng (format) yêu cầu và gửi lại



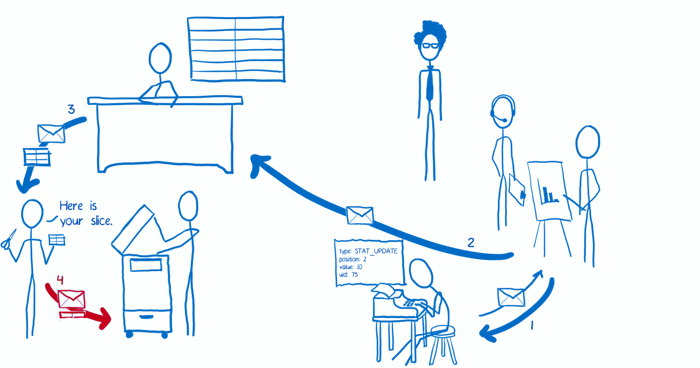
1. Action được gửi tự động (nếu bindActionCreators() đã được chuẩn bị) hoặc The View sẽ gửi



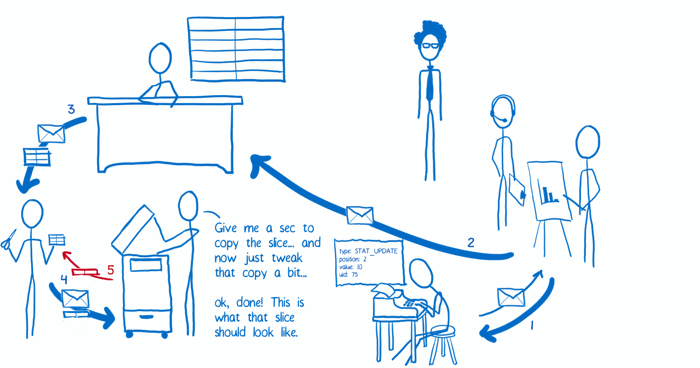
1. The Store nhận Action sau đó gửi State tree hiện tại và Action cho Root Reducer.



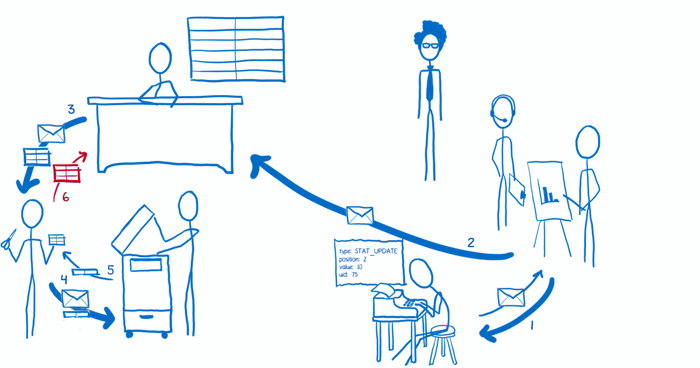
1. Root Reducer chia State ra thành nhiều phần và gửi cho từng subreducers biết cách xử lý chúng.



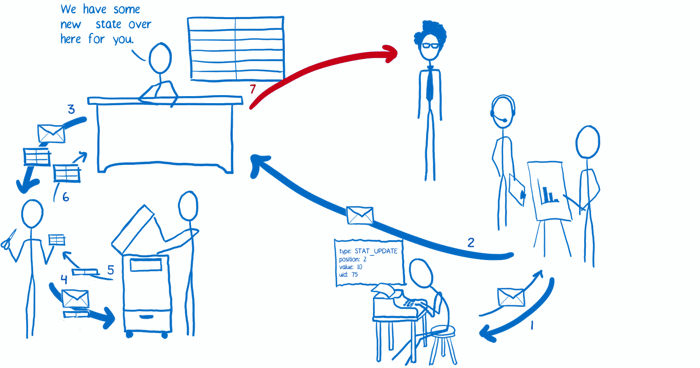
1. Subreducers tạo ra 1 bản copy từ phần nhận được và thay đổi trên bản copy. Sau đó gửi lại bản copy cho Root Reducer



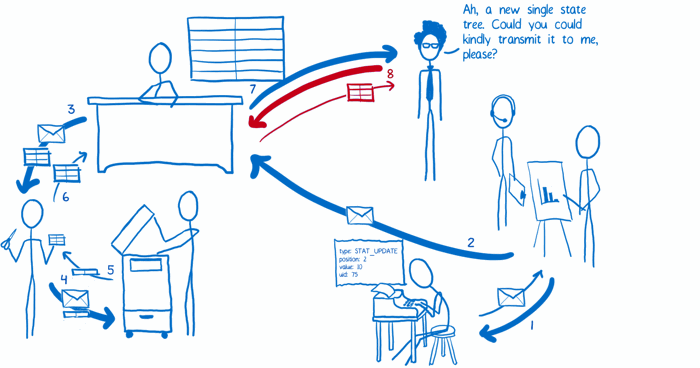
1. Khi tất cả subreducers trả về các phần copies, Root Reducer ghép chúng lại tạo thành 1 update State tree và gửi lại cho Store. Store thay thế State tree cũ bằng State tree mới.



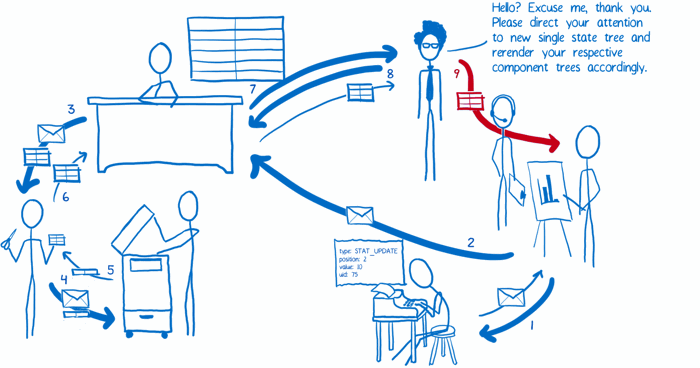
1. Store nói với The view layer binding là có State mới.



1. The view layer binding báo Store gửi State mới cho mình



1. The view layer binding kích hoạt render view



**Kết**

Vậy là chúng ta đã tìm hiểu được những khái niệm cơ bản đầu tiên của Redux như nguyên lý, cấu trúc, và data flow.

Thông qua phần 1 này hi vọng mọi người hình dung được các thành phần trong 1 app sử dụng Redux. Để hiểu kĩ hơn về cách sử dụng Redux, hẹn mọi người ở phần sau **Project đầu tiên**. (Bài này mang tính chất chém gió và thả thính là chính (yaoming))

Và hãy nhớ những nhân vật của chúng ta, để còn gặp trực tiếp họ trong phần sau.

**Tài liệu tham khảo**

* <http://redux.js.org/>
* <https://code-cartoons.com/a-cartoon-intro-to-redux-3afb775501a6#.74jmungb6>

**Redux cho người mới bắt đầu - Part 2 First Project**

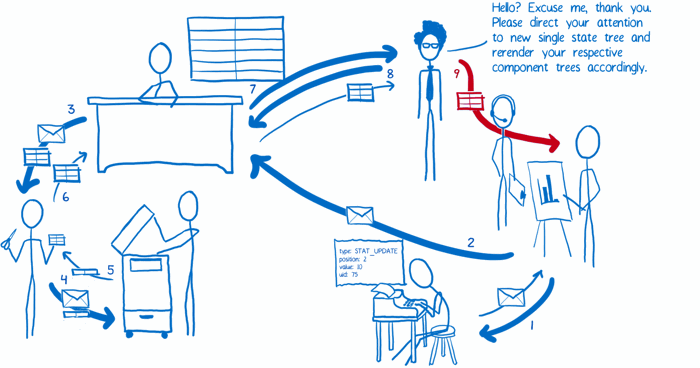
[Editors' Choice](https://viblo.asia/editors-choice) [Trending](https://viblo.asia/trending) [redux](https://viblo.asia/tags/redux)[ReactJS](https://viblo.asia/tags/reactjs)



**Mở đầu**

Tiếp nối bài viết về Redux cho người mới bắt đầu, trong bài viết này chúng ta sẽ cùng nhau thực hiện một Todo app để tìm hiểu cách sử dụng redux trong một project thực tế.

Trước khi bắt đầu cùng nhìn lại 1 lần các nhân vật ở [kì trước](https://viblo.asia/phamvanduc/posts/ZjleaBBZkqJ) :



**Setup**

Trong bài viết này mình sẽ sử dụng React + Redux. Chúng ta có thể tạo nhanh một react app thông qua [create-react-app](https://github.com/facebookincubator/create-react-app):

sudo npm install -g create-react-app

create-react-app test-react-redux

Tiếp theo là Redux:

cd test-react-redux

npm install --save redux react-redux

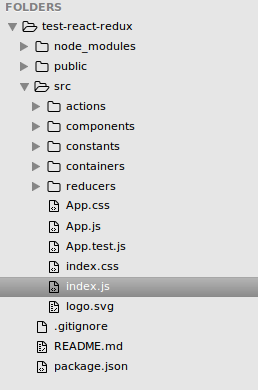
* [react-redux](https://github.com/reactjs/react-redux) : chính là The view layer binding trong kì trước, làm nhiệm vụ kết nối cho redux và react.

**Let's go**

**1.Cấu trúc thư mục**

Ở phần trước chúng ta đã biết một app sử dụng Redux có 4 thành phần cơ bản action reducer store và view. Trong đó store chúng ta chỉ việc khởi tạo trong root component còn việc quản lý Redux sẽ lo. view thì bao gồm smart components (containers) những components giao tiếp với Redux và dumb components (components) những components không giao tiếp với Redux. Các action type của dụng thường là các hằng số được định nghĩa trước.

Do đó để tiện việc quản lý chúng ta có thể tạo ra các thư mục actions constants reducers containers components, app của chúng ta sẽ có cấu trúc như sau:



**2.Actions**

Đầu tiên chúng ta sẽ tạo ra các actions, các bạn còn nhớ The Action creators không, anh ta tạo ra những action là formated object chứa type và thông tin của action đó. Type thường sẽ là một hằng số được định nghĩa trước.

Ở đây chúng ta có TodoActions định nghĩa ra các thao tác thêm sửa xóa... công việc. Các action mang type và các thông tin id, text.

// actions/TodoActions.js

import \* as types from '../constants/ActionTypes';

export function addTodo(text) {

return {

type: types.ADD\_TODO,

text

};

}

export function deleteTodo(id) {

return {

type: types.DELETE\_TODO,

id

};

}

export function editTodo(id, text) {

return {

type: types.EDIT\_TODO,

id,

text

};

}

export function markTodo(id) {

return {

type: types.MARK\_TODO,

id

};

}

export function markAll() {

return {

type: types.MARK\_ALL

};

}

export function clearMarked() {

return {

type: types.CLEAR\_MARKED

};

}

**3. Reducers**

Tiếp theo là các reducers, chúng ta tạo ra các sub-reducers và một root-reducer quản lý chung. Reducers là các pure function hoạt động theo nguyên lý :

(state, action) => (new state)

Vì là pure function nên các reducers sẽ không trực tiếp thay đổi state mà nó nhận được, mà tạo ra các bản copy và thay đổi trên đó. Để thực hiện điều này chúng ta có thể dùng các function filter() map() Object.assign()...

// reducers/todosReducers.js

import { ADD\_TODO, DELETE\_TODO, EDIT\_TODO, MARK\_TODO, MARK\_ALL, CLEAR\_MARKED } from '../constants/ActionTypes';

const initialState = [{

text: 'Use Redux',

marked: false,

id: 0

}];

export default function todos(state = initialState, action) {

switch (action.type) {

case ADD\_TODO:

return [{

id: (state.length === 0) ? 0 : state[0].id + 1,

marked: false,

text: action.text

}, ...state];

case DELETE\_TODO:

return state.filter((todo) => todo.id !== action.id);

case EDIT\_TODO:

return state.map((todo) => todo.id === action.id ? { ...todo, text: action.text } : todo);

case MARK\_TODO:

return state.map((todo) => todo.id === action.id ? { ...todo, marked: !todo.marked } : todo);

case MARK\_ALL:

const areAllMarked = state.every((todo) => todo.marked);

return state.map((todo) => ({...todo, marked: !areAllMarked}));

case CLEAR\_MARKED:

return state.filter((todo) => todo.marked === false);

default:

return state;

}

}

Root-reducer sẽ tập hợp các sub-reducers lại thông qua combineReducers() của Redux.

// reducers/rootReducers.js

import { combineReducers } from 'redux';

import todosReducers from './todosReducers';

const rootReducer = combineReducers({

todosReducers

});

export default rootReducer;

**4.Views**

* Smart Component (containers)

Containers là những component giao tiếp với Redux thông qua connect() của react-redux.

connect() nhận vào 4 tham số mapStateToProps mapDispatchToProps mergeProps options:

* + mapStateToProps(state, [ownProps]) là function. Nếu được định nghĩa, container sẽ được đăng ký (subscribe) với store. Mỗi khi store update mapStateToProps sẽ được gọi, object mà nó trả về sẽ được merge với props của container. Nếu ownProps được định nghĩa, giá trị của nó sẽ là props được gửi cho container, đồng thời mỗi khi container nhận được new props thì mapStateToProps cũng sẽ được gọi. Nếu mapStateToProps không được định nghĩa container sẽ không được đăng ký và nhận update từ store.
  + mapDispatchToProps là object hoặc function. Nếu là object mỗi function bên trong object sẽ được coi là một action creator, đồng thời tất cả function này sẽ được tự động chay bởi bindActionCreators() và merge chúng với props của container. Nếu là function mapDispatchToProps sẽ nhận 2 tham số (dispatch, [ownProps]), chúng ta sẽ tự định nghĩa cách bind action với dispatch, chúng ta cũng có thể sử dụng bindActionCreators({action}, dispatch) để tự động bind. Nếu ownProps được định nghĩa, giá trị của nó sẽ là props được gửi cho container, đồng thời mỗi khi container nhận được new props thì mapDispatchToProps cũng sẽ được gọi. Nếu mapDispatchToProps không được định nghĩa sẽ chỉ có dispatch được merge vào props của container.
  + mergeProps(stateProps, dispatchProps, ownProps) là function. Nếu được định nghĩa, nó sẽ nhận vào tham số là kết qủa của mapStateToProps mapDispatchToProps và parent props. Object mà nó trả về là props được gửi cho container. Nếu không được định nghĩa Object.assign({}, ownProps, stateProps, dispatchProps) sẽ được sử dụng mặc định.
  + options là object. Nếu được định nghĩa sẽ điều chỉnh hành vi của connector. Chứa 2 giá trị pure và withRef. Nếu pure = true thì thực thi shouldComponentUpdate() và so sánh kết qủa của mergeProps để tránh những update không cần thiết, mặc định là true. Nếu withRef = true thì lưu trữ lại ref đến container instance và có thể truy cập thông qua getWrappedInstance(), mặc định false.

Trong app của chúng ta có 1 container là TodoApp:

// containers/TodoApp.js

import React, { Component } from 'react';

import { connect } from 'react-redux';

import { bindActionCreators } from 'redux';

import Header from '../components/Header';

import MainSection from '../components/MainSection';

import \* as TodoActions from '../actions/TodoActions';

class TodoApp extends Component {

render() {

const { todos, actions } = this.props;

return (

<div>

<Header addTodo={actions.addTodo} />

<MainSection todos={todos} actions={actions} />

</div>

);

}

}

function mapStateToProps(state) {

return {

todos: state.todosReducers

};

}

function mapDispatchToProps(dispatch) {

return {

actions: bindActionCreators(TodoActions, dispatch)

};

}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp);

* Dump Components

Là những component thông thường, chúng không giao tiếp với Redux, chỉ nhận giá trị và thao tác thông qua props.

Các xử lý hiển thị dữ liệu sẽ thực thi ở đây và các action nhận được từ container sẽ sử dụng như callback.

Trong app của chúng ta chúng là Header, MainSection .....

**5.Root component**

Trong mọi React-app đều có root component, ở app sử dụng Redux root component đảm nhận thêm việc khởi tạo store và bao các component lại với Provider của react-redux giúp component có thể giao tiếp với redux.

// index.js

import 'todomvc-app-css/index.css';

import React from 'react';

import ReactDOM from 'react-dom';

import { createStore } from 'redux';

import { Provider } from 'react-redux'

import TodoApp from './containers/TodoApp';

import rootReducer from './reducers/rootReducer';

// initialState

const initialState = {}

// Create store

const store = createStore(rootReducer, initialState);

const appRoot = (

<Provider store={store}>

<div>

<TodoApp />

</div>

</Provider>

)

ReactDOM.render(appRoot, document.getElementById('root'))

**Kết**

Vậy là chúng ta đã tạo được một ứng dụng đơn giản sử dụng React-Redux. Kết hợp với Data Flow trong phần 1 hi vọng các bạn phần nào hình dung được cách mà Redux hoạt động và cách sử dụng Redux trong một ứng dụng.

Redux còn rất nhiều khái niệm khác như Async Middleware... các bạn có thể tìm hiểu thêm ở trang chủ của [Redux](http://redux.js.org/) hoặc hẹn gặp lại các bạn trong một bài viết khác (có thể (yaoming))

**Tài liệu**

1. [Source code](https://github.com/ducpv193/test-react-redux)

**Redux cho người mới bắt đầu - Part 3 Middleware**

[ReactJS](https://viblo.asia/tags/reactjs)[redux](https://viblo.asia/tags/redux)



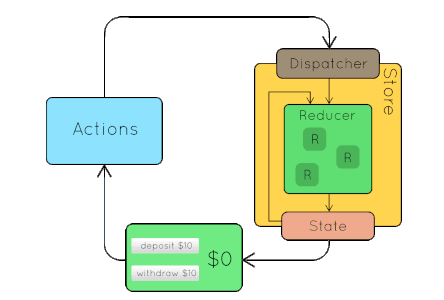
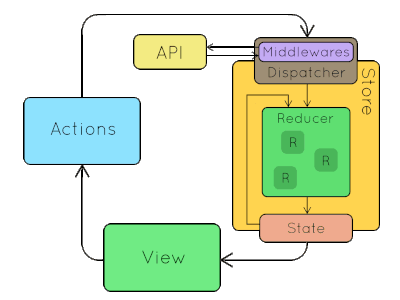
**Mở đầu**

Chào tất cả mọi người, chúng ta lại gặp nhau trong bài viết thứ 3 của series Redux cho người mới bắt đầu sau 1 thời gian rất rất dài (vì đứa viết bài lười quá (yaoming) ). Ở bài viết [số 2](https://viblo.asia/phamvanduc/posts/aRBvXWQZkWE), chúng ta đã làm thử một ứng dụng To-Do đơn giản. Trong thực tế, ứng dụng của chúng ta còn cần giải quyết nhiều vấn đề hơn như ghi log hoạt động, handle error, async... Để giải quyết những vấn đề này, Redux đã đưa ra giải pháp gì, chúng ta sẽ tìm hiểu trong bài viết này.

**Middleware**

Middleware khá phổ biến với các Framework server-side, nó được đặt giữa thời điểm server nhận request và thời điểm server response. Ở Redux, Middleware giải quyết vấn đề khác với các Framework server-side nhưng định nghĩa có phần tương tự:

It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.

Hiểu một cách đơn giản middleware cho phép chúng ta can thiệp vào giữa thời điểm dispatch một action và thời điểm action đến được reducer. Chúng ta có thể thấy sự thay đổi của flow khi có sử dụng middleware qua hình dưới:  

**Sử dụng Middleware**

Để sử dụng được Middleware chúng ta cần sử dụng function applyMiddleware của redux khi khởi tạo store

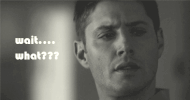
// index.js

import { createStore, applyMiddleware } from 'redux';

import 'yourMiddleware' from 'your-middleware';

import rootReducer from './reducers/rootReducer';

const store = createStore(rootReducer, applyMiddleware(yourMiddleware));

 À thế cái yourMiddleware kia ở đâu ra ? Tất nhiên là chúng ta sẽ tự viết, hoặc cũng có thể dùng những thư viện middleware xây dựng sẵn. Để có thể tự viết được một Middleware cho redux, chúng ta cần hiểu bản chất Middleware mà Redux cung cấp. Chúng ta sẽ đề cập đến nó trong một phần khác, còn trong phần này cứ dùng thử một vài thư viện đã.

**Middleware ways**

Hiện tại, có khá nhiều thư viện middleware cho Redux, cá nhân mình thì thấy có 3 thư viện chúng ta nên thử qua redux-thunk redux-saga và redux-observable . Mỗi thư viện có phương pháp giải quyết vấn đề side effects riêng. Nhưng các bạn yên tâm hôm nay chúng ta sẽ chỉ thử duy nhất 1 thư viện thôi (yaoming) đó là redux-thunk thư viện được giới thiệu bởi Redux và cũng là cái dễ hiểu nhất.

**Redux-Thunk: Function Action**

“Redux Thunk middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch and getState as parameters.” — Redux Thunk documentation

Khá là dễ hiểu, thunk cho phép chúng ta viết action là function thay vì bắt buộc là object như định nghĩa action mà Redux đưa ra. Hơi khó tưởng tượng ? chúng ta demo thử cái xem sao.

**Demo**

Trước hết, chúng ta cứ chuẩn bị một project mới đã. Lần này chúng ta sẽ làm một ứng dụng load dữ liệu.

create-react-app redux\_middleware\_ways

cd redux\_middleware\_ways

yarn add redux react-redux redux-thunk redux-saga redux-observable

cd src

mkdir reducers

touch reducers/index.js reducers/dataReducer.js

touch configureStore.js constants.js actions.js

// constants.js

export const FETCHING\_DATA = 'FETCHING\_DATA'

export const FETCHING\_DATA\_SUCCESS = 'FETCHING\_DATA\_SUCCESS'

export const FETCHING\_DATA\_FAILURE = 'FETCHING\_DATA\_FAILURE'

// actions.js

import { FETCHING\_DATA, FETCHING\_DATA\_SUCCESS, FETCHING\_DATA\_FAILURE } from './constants'

export function getData() {

return {

type: FETCHING\_DATA

}

}

export function getDataSuccess(data) {

return {

type: FETCHING\_DATA\_SUCCESS,

data,

}

}

export function getDataFailure() {

return {

type: FETCHING\_DATA\_FAILURE

}

}

export function fetchData() {}

// reducer/dataReducer.js

import { FETCHING\_DATA, FETCHING\_DATA\_SUCCESS, FETCHING\_DATA\_FAILURE } from '../constants'

const initialState = {

data: [],

dataFetched: false,

isFetching: false,

error: false

}

export default function dataReducer (state = initialState, action) {

switch (action.type) {

case FETCHING\_DATA:

return {

...state,

data: [],

isFetching: true

}

case FETCHING\_DATA\_SUCCESS:

return {

...state,

isFetching: false,

data: action.data

}

case FETCHING\_DATA\_FAILURE:

return {

...state,

isFetching: false,

error: true

}

default:

return state

}

}

// reducer/index.js

import { combineReducers } from 'redux'

import appData from './dataReducer'

const rootReducer = combineReducers({

appData

})

export default rootReducer

// App.js

import React from 'react'

import { connect } from 'react-redux'

import { fetchData } from './actions'

let styles

const App = (props) => {

const {container, text, button, buttonText} = styles

return (

<div style={container}>

<div style={text}>Redux Examples</div>

<div style={button}>

<div style={buttonText}>Load Data</div>

</div>

</div>

)

}

styles = {

container: {

marginTop: 100

},

text: {

textAlign: 'center'

},

button: {

display: 'flex',

minHeight: 60,

margin: 10,

justifyContent: 'center',

alignItems: 'center',

backgroundColor: '#0b7eff'

},

buttonText: {

color: 'white'

}

}

function mapStateToProps (state) {

return {

appData: state.appData

}

}

function mapDispatchToProps (dispatch) {

return {

fetchData: () => dispatch(fetchData())

}

}

export default connect(

mapStateToProps,

mapDispatchToProps

)(App)

// index.js

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import configureStore from './configureStore'

const store = configureStore()

import App from './App';

const ReduxApp = () => (

<Provider store={store}>

<App />

</Provider>

);

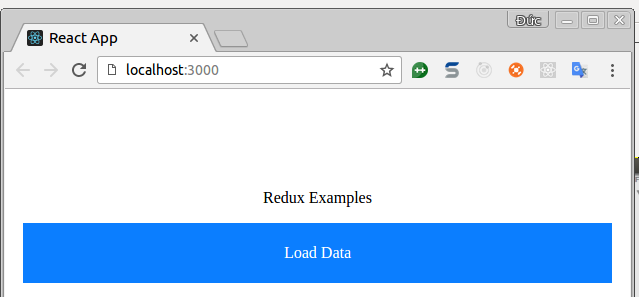
ReactDOM.render(

<ReduxApp />,

document.getElementById('root')

);

`npm start`

Khâu chuẩn bị đã xong. À khoan chúng ta còn cần 1 api để load dữ liệu nữa. Respone chậm một chút để nhận thấy sự thay đổi.

touch api.js

// api.js

const people = [

{ name: 'Nader', age: 36 },

{ name: 'Amanda', age: 24 },

{ name: 'Jason', age: 44 }

]

export default () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

return resolve(people)

}, 3000)

})

}

Để sử dụng thunk chúng ta sẽ chỉ cần thêm một vài dòng code:

// configureStore.js

import { createStore, applyMiddleware } from 'redux'

import rootReducer from './reducers'

import thunk from 'redux-thunk' //import thunk

export default function configureStore() {

let store = createStore(rootReducer, applyMiddleware(thunk)) // create store sử dụng thunk

return store

}

// actions.js

//...

import getPeople from './api'

export function fetchData() {

return (dispatch) => {

dispatch(getData())

getPeople()

.then((data) => {

dispatch(getDataSuccess(data))

})

.catch((err) => console.log('err:', err))

}

}

Khi action fetchData được gọi, nó trả về một function nhận dispatch làm tham số, function sẽ dispatch getData action. Tiếp theo function getPeople sẽ được gọi. Sau khi getPeople hoàn thành sẽ dispatch getDataSuccess action. Action fetchData bây giờ đã trở thành một [thunk](https://github.com/gaearon/redux-thunk#whats-a-thunk).

Thunk là một function được bao lại để tạm dừng nó cho đến khi được gọi

Chúng ta cũng cần thay đổi view một chút:

// app.js

const App = (props) => {

const {container, text, button, buttonText} = styles

return (

<div style={container}>

<div style={text}>Redux Examples</div>

<div style={button} onClick={() => props.fetchData()}>

<div style={buttonText}>

{

props.appData.isFetching && <div>Loading</div>

}

{

props.appData.data.length ? (

props.appData.data.map((person, i) => {

return <div key={i} >

<div>Name: {person.name}</div>

<div>Age: {person.age}</div>

</div>

})

) : null

}

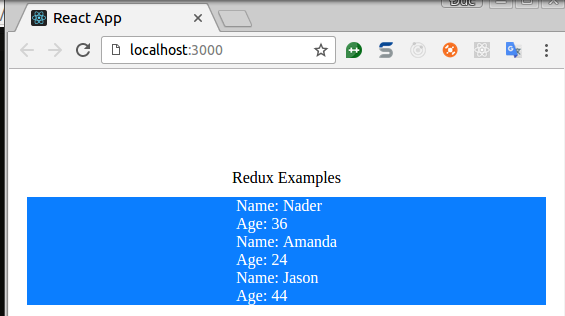
</div>

</div>

</div>

)

}

Giờ thì thử click vào nút màu xanh và xem kết quả.  Vậy là chúng ta đã sử dụng được redux-thunk khá đơn giản. Điểm khác biệt duy nhất là:

Chúng ta có thể viết action là một function thay vì bắt buộc là object

***Ưu điểm:***

* Redux-thunk không yêu cầu chúng ta phải hiểu thêm concept nào của riêng nó để có thể sử dụng, tất cả vẫn là Redux. Ý tưởng cơ bản của Thunk là nếu cần kích hoạt side effects, hãy dùng chính actions. Một function trả về một function có thể thực hiện mọi thứ mà async call cần và dispatch bất kì action nào ta muốn.

***Nhược điểm:***

* Chính ý tưởng của Thunk sớm dẫn nó tới sự phức tạp về khả năng test actions, do chỉ có thể biết các thunk làm gì cho đến khi nó được thực thi. Chúng ta vẫn có thể test với mocks, nhưng điều đó sẽ phá vỡ các quy tắc functional programming của Redux.

**Kết**

Chúng ta đã cùng nhau điểm qua khái niệm và sử dụng một middleware đơn giản. Không quá nhiều nhưng chúng ta cũng đã có được những cái nhìn đầu tiên. Middleware là một khái niệm khá thú vị của Redux. Nó hoạt động ra sao, làm thế nào để tự viết được một Middleware hay những thư viện Middleware xuất hiện sau nó đều là những thứ hay ho đáng xem qua. Hẹn gặp lại mọi người trong phần tiếp theo (hi vọng là không xa lắm (yaoming) )

**[Redux] Middleware Redux-saga**

[redux](https://viblo.asia/tags/redux)[ReactJS](https://viblo.asia/tags/reactjs)[Redux-saga](https://viblo.asia/tags/redux-saga)



Xin chào mọi người, chúng ta lại gặp nhau trong loạt bài viết về Redux. Ở bài viết [trước](https://viblo.asia/p/redux-cho-nguoi-moi-bat-dau-part-3-middleware-3Q75wDXMKWb) chúng ta đã cũng tìm hiểu về middleware trong redux là gì và sử dụng thử redux-thunk. Trong bài viết này chúng ta sẽ đề cập đến [Redux-saga](https://github.com/redux-saga/redux-saga) những câu chuyện cổ tích hoành tráng về redux (yaoming).

**Saga In a Nutshell**

**Đầu tiên, redux-saga là cái gì ?**

Nghe có vẻ liên quan đến truyện cổ tích, nhưng thực ra thì cũng chẳng liên quan lắm. Theo [Yassine Elouafi](https://twitter.com/yassineelouafi2) , tác giả của redux-saga:

redux-saga is a library that aims to make side effects (i.e. asynchronous things like data fetching and impure things like accessing the browser cache) in React/Redux applications easier and better.

Đơn giản thì nó cũng cũng là một thư viện hỗ trợ việc xử lí các side effect trong ứng dụng React/Redux, ví dụ như các xử lí bất đồng bộ khi load data chẳng hạn. Thế thì nó cũng giống redux-thunk à ?? À tất nhiên là về mục tiêu thì nó chẳng khác gì cả (yaoming) nhưng phương pháp tiếp cận và xử lí side effect thì có khác chút chút.

**Redux saga trông như thế nào**

Một saga của chúng ta thì thường trông như thế này:

import { put, call, takeEvery } from 'redux-saga/effects';

function\* someSaga() {

// Wait for (every) SOME\_ACTION action

takeEvery('SOME\_ACTION', doSomeThing);

}

function\* doSomeThing(action}) {

try {

// Tell redux-saga to call fetchSomeThing with the param from action

yield call(fetchSomeThing, action.payload)

// Tell redux-saga to dispatch the someThingSuccess action

yield put(someThingSuccess())

}

catch (err) {

// You get it

yield put (someThingFailed(err))

}

}

Và để sử dụng saga thì cũng giống như với các middleware khác

...

import createSagaMiddleware from 'redux-saga';

const sagaMiddleware = createSagaMiddleware();

const createStoreWithSagas = createStore(reducer, applyMiddleware(sagaMiddleware));

sagaMiddleware.run(someSaga); // run saga

**Ý tưởng của Redux saga là gì ?**

Redux-saga có một giải pháp thực sự tốt cho vấn đề side effect. Nó tách side effect khỏi action bằng cách đặt chúng vào các saga. Có thể coi những saga như những mẩu truyện nhỏ mô tả hành vi của 1 hay nhiều action.

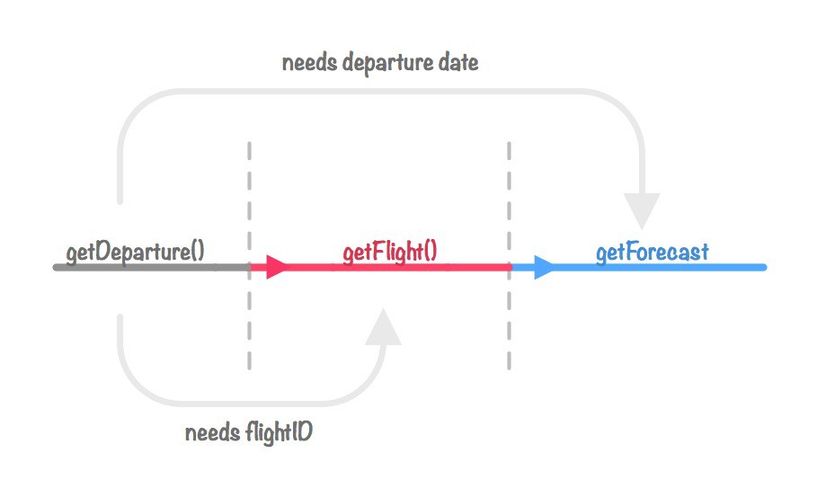
Saga hoạt động như các daemon (chương trình chạy nền) được điều khiển bởi sagaMiddleware của redux-saga. Và như các bạn thấy cú pháp định nghĩa 1 saga:

function\* someSaga()

Đây là cú pháp khai báo một generator function (ES6). Bản thân các saga là một generator function, điều này cho phép redux-saga có thể điều khiển từng bước mà saga hoạt động.

Để hiểu hơn về cách mà redux-saga điều khiển từng bước của saga, chúng ta cần có một chút khái niệm về generator function, các bạn có thể đọc thêm [tại đây](https://davidwalsh.name/es6-generators).

**The Flight Dashboard Case**

Để cùng xem cách dùng saga trong một ứng dụng thực tế thế nào chúng ta sẽ xây dựng một bảng điều khiển chuyến bay. Source code ở [đây](https://github.com/andresmijares/async-redux-saga) và demo ở [đây](http://async-redux-saga.surge.sh/). Kịch bản của chúng ta như sau:

**Setup**

Như chúng ta thấy, một chuỗi 3 APIs được gọi: getDeparture -> getFlight ->getForecast , nên API của chúng ta sẽ trông như thế này:

class TravelServiceApi {

static getUser() {

return new Promise((resolve) => {

setTimeout(() => {

resolve({

email : "somemockemail@email.com",

repository: "http://github.com/username"

});

}, 3000);

});

}

static getDeparture(user) {

return new Promise((resolve) => {

setTimeout(() => {

resolve({

userID : user.email,

flightID : “AR1973”,

date : “10/27/2016 16:00PM”

});

}, 2500);

});

}

static getForecast(date) {

return new Promise((resolve) => {

setTimeout(() => {

resolve({

date: date,

forecast: "rain"

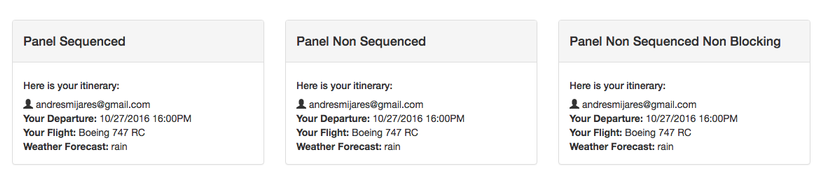
});

}, 2000);

});

}

}

API này chứa một số thông tin cho phép chúng ta thiết lập kịch bản. Đầu tiên chúng ta sẽ có thông tin một người dùng. Sau đó với thông tin người dùng, chúng ta sẽ có giờ khởi hành, chuyến bay và dự báo thời tiết từ đó chúng ta có thể tạo ra một số bảng điều khiển trông như thế này:Chúng là 3 component khác nhau với 3 đại diện trong store được tạo ra bởi 3 reducer (dashboard, dashboard2, dashboard3). Mỗi reducer sẽ trông như thế này:

export const dashboard = (state = {}, action) => {

switch(action.type) {

case 'FETCH\_DASHBOARD\_SUCCESS':

return Object.assign({}, state, action.payload);

default :

return state;

}

};

Chúng ta sử dụng các reducer khác nhau cho mỗi bảng điều khiển, với 3 kịch bản khác nhau và cung cấp cho component vào user data chung:

const mapStateToProps =(state) => ({

user : state.user,

dashboard : state.dashboard

});

Và chúng ta mới xong phần setup... giờ thì mới bắt đầu vào việc chính.

**Show me the Sagas**

William Deming từng nói:

Nếu bạn không thể mô tả những gì bạn đang làm như quá trình, thì bạn không hề biết mình đang làm gì.

Chúng ta sẽ đi từng bước để xem cách làm việc với Redux Saga

**1. Đăng ký Saga**

Đầu tiên chúng ta sẽ tạo ra root saga:

function\* rootSaga() {

yield[

fork(loadUser),

takeLatest('LOAD\_DASHBOARD', loadDashboardSequenced)

];

}

Redux-saga cung cấp một số method gọi là ***effect*** , chúng ta sẽ định nghĩa một số chúng:

* **Fork**: thực hiện một hoạt động non-blocking trên function được truyền cho nó.
* **Take**: *tạm dừng* cho đến khi nhận được *action*
* **Race**: chạy nhiều effect đồng thời, sau đó hủy tất cả nếu một trong số đó kết thúc.
* **Call**: gọi function. Nếu nó return về một *promise*, tạm dừng saga cho đến khi promise được giải quyết.
* **Put**: dispatch một action.
* **Select**: chạy một selector function để lấy data từ state.
* **takeLatest**: có nghĩa là nếu chúng ta thực hiện một loạt các actions, nó sẽ chỉ thực thi và trả lại kết quả của của actions cuối cùng.
* **takeEvery**: thực thi và trả lại kết quả của mọi actions được gọi.

Hiện tại chúng ta đăng kí cho *loadUser* sử dụng fork và một effect takeLatest chờ actions "**LOAD\_DASHBOARD**" để thực thi. Chúng ta sẽ tiếp tục với các step tiếp theo.

1. Inject Saga Middleware vào Redux Store

const sagaMiddleware = createSagaMiddleware();

const store = createStore(rootReducer, [], compose(

applyMiddleware(sagaMiddleware)

);

sagaMiddleware.run(rootSaga); /\* inject our sagas into the middleware\*/

1. Tạo các Saga Đầu tiên chúng ta sẽ định nghĩa loadUser saga:

function\* loadUser() {

try {

//1st step

const user = yield call(getUser);

//2nd step

yield put({type: 'FETCH\_USER\_SUCCESS', payload: user});

} catch(error) {

yield put({type: 'FETCH\_FAILED', error});

}

}

Chúng ta có thể đọc đoạn code như sau:

* Gọi function **getUser** và gán kết quả vào hằng **user**.
* Sau đó, dispatch actions "**FETCH\_USER\_SUCCESS**" với payload là **user** để store sử dụng.
* Nếu xảy ra lỗi dispatch action "**FETCH\_FAILED**". Như các bạn thấy, chúng ta có thể gán kết quả của **yeild** cho một biến.

Bây giờ chúng ta sẽ tạo ra **loadDashboardSequenced** saga:

function\* loadDashboardSequenced() {

try {

yield take(‘FETCH\_USER\_SUCCESS’);

const user = yield select(state => state.user);

const departure = yield call(loadDeparture, user);

const flight = yield call(loadFlight, departure.flightID);

const forecast = yield call(loadForecast, departure.date);

yield put({type: ‘FETCH\_DASHBOARD\_SUCCESS’, payload: {forecast, flight, departure} });

} catch(error) {

yield put({type: ‘FETCH\_FAILED’, error: error.message});

}

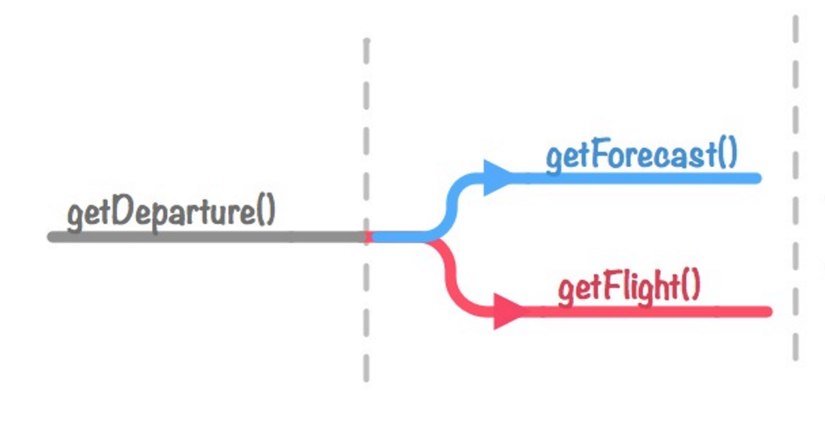
}

Chúng ta có thể đọc flow của saga như sau:

* Chờ action "**FETCH\_USER\_SUCCESS**". Về cơ bản thì nó sẽ chờ đến khi event được dispatch.
* Lấy **user** từ store, **select** effect nhận 1 function có thể truy cập vào store. Chúng ta gán thông tin user vào hằng số **user**.
* Thực thi async **loadDeparture** để load giờ khởi hành
* Sau khi **loadDeparture** hoàn thành, thực thi **loadFlight** với kết quả của loadDeparture.
* Tiếp tục thực thi **loadForecast** với kết quả của **loadFlight**.
* Cuối cùng, dispatch action "**FETCH\_DASHBOARD\_SUCCESS**" với payload là kết quả của 3 thực thi bên trên.

Như các bạn thấy, saga là một tập hợp các bước chờ đợi các hành động trước nó để thay đổi hành vi. Sau khi tất cả hoàn thành, thông tin đã sẵn sàng để sử dụng trọng store.

Khá là rõ ràng, như đọc văn miêu tả (yaoming).

Giờ chúng ta sẽ kiểm tra một trường hợp khác. Xem xét **getFlight** và **getForecast** có thể gọi cùng lúc. Chúng không phải chờ một cái khác hoàn thành để bắt đầu cái còn lại, chúng ta có thể tạo 1 bảng điều khiển khác cho trường hợp đó 

**Non-blocking Saga**

Để thực thi 2 hoạt động non-blocking, chúng ta cần một sửa đổi nhỏ với saga trước đó.

function\* loadDashboardNonSequenced() {

try {

//Wait for the user to be loaded

yield take('FETCH\_USER\_SUCCESS');

//Take the user info from the store

const user = yield select(getUserFromState);

//Get Departure information

const departure = yield call(loadDeparture, user);

//Here is when the magic happens

const [flight, forecast] = yield [call(loadFlight, departure.flightID), call(loadForecast, departure.date)];

//Tell the store we are ready to be displayed

yield put({type: 'FETCH\_DASHBOARD2\_SUCCESS', payload: {departure, flight, forecast}});

} catch(error) {

yield put({type: 'FETCH\_FAILED', error: error.message});

}

}

Sự thay đổi ở đây là chúng ta sẽ đăng ký yield như một array:

const [flight, forecast] = yield [call(loadFlight, departure.flightID), call(loadForecast, departure.date)];

Vì thế, cả 2 hoạt động được gọi song song, nhưng cuối cùng chúng ta vẫn đợi cả 2 hoàn thành để cappj nhật UI nếu cần thiết.

Sau đó chúng ta cần đăng ký saga với rootSaga:

function\* rootSaga() {

yield[

fork(loadUser),

takeLatest('LOAD\_DASHBOARD', loadDashboardSequenced),

takeLatest('LOAD\_DASHBOARD2' loadDashboardNonSequenced)

];

}

Vậy nếu chúng ta muốn cập nhật UI ngay khi load xong dữ liệu thì sao ?

**Non-Sequenced và Non-Blocking Saga**

Chúng ta có thể cô lập các saga và kết hợp chúng, có nghĩa là chúng có thể hoạt động độc lập. Đây chính xác là những gì chúng ta cần.

**Bước #1**: Chúng ta sẽ cô lập *Forecast* và *Flight* saga. Cả 2 đều phụ thuộc vào *departure*

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*Flight Saga\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

function\* isolatedFlight() {

try {

/\* departure will take the value of the object passed by the put\*/

const departure = yield take('FETCH\_DEPARTURE3\_SUCCESS');

const flight = yield call(loadFlight, departure.flightID);

yield put({type: 'FETCH\_DASHBOARD3\_SUCCESS', payload: {flight}});

} catch (error) {

yield put({type: 'FETCH\_FAILED', error: error.message});

}

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*Forecast Saga\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

function\* isolatedForecast() {

try {

/\* departure will take the value of the object passed by the put\*/

const departure = yield take('FETCH\_DEPARTURE3\_SUCCESS');

const forecast = yield call(loadForecast, departure.date);

yield put({type: 'FETCH\_DASHBOARD3\_SUCCESS', payload: { forecast, }});

} catch(error) {

yield put({type: 'FETCH\_FAILED', error: error.message});

}

}

Điều quan trọng ở đây là gì? Đây là cách mà chúng ta kiến trúc các saga:

* Cả 2 nhóm đều đang chờ action "**FETCH\_DEPARTURE3\_SUCCESS**" để bắt đầu.
* Chúng sẽ nhận được một giá trị khi event này được kích hoạt. (Chi tiết hơn ở bước tiếp theo)
* Chúng sẽ cùng thực thi **call** và cả 2 sẽ cùng dispatch 1 action "**FETCH\_DASHBOARD3\_SUCCESS**" sau khi hoàn thành. Nhưng cả 2 sẽ gửi đến dữ liệu khác nhau, nhờ sức mạnh của Redux chúng ta có thể làm việc này mà ko phải sửa đổi reducer.

**Bước #2**: Chúng ta thay đổi với saga ban đầu 1 chút, chắc chắn nó sẽ gửi departure đến cho 2 saga phía trên:

function\* loadDashboardNonSequencedNonBlocking() {

try {

//Wait for the action to start

yield take('FETCH\_USER\_SUCCESS');

//Take the user info from the store

const user = yield select(getUserFromState);

//Get Departure information

const departure = yield call(loadDeparture, user);

//Update the store so the UI get updated

yield put({type: 'FETCH\_DASHBOARD3\_SUCCESS', payload: { departure, }});

//trigger actions for Forecast and Flight to start...

//We can pass and object into the put statement

yield put({type: 'FETCH\_DEPARTURE3\_SUCCESS', departure});

} catch(error) {

yield put({type: 'FETCH\_FAILED', error: error.message});

}

}

Không có gì khác biệt ở đây cho đến phần **put** effect. Chúng ta sẽ thêm 1 action "**FETCH\_DEPARTURE3\_SUCCESS**" và kèm theo *departure* cho 2 saga phía trên.

**Thế còn testing thì sao?**

Saga rất dễ test, nhưng chúng bị ràng buộc với các bước bên trong, được thiết lập theo trình tự do tính chất của generator. Chúng ta cùng xem ví dụ sau (hoặc test trong [repo demo](https://github.com/andresmijares/async-redux-saga)):

describe('Sequenced Saga', () => {

const saga = loadDashboardSequenced();

let output = null;

it('should take fetch users success', () => {

output = saga.next().value;

let expected = take('FETCH\_USER\_SUCCESS');

expect(output).toEqual(expected);

});

it('should select the state from store', () => {

output = saga.next().value;

let expected = select(getUserFromState);

expect(output).toEqual(expected);

});

it('should call LoadDeparture with the user obj', (done) => {

output = saga.next(user).value;

let expected = call(loadDeparture, user);

done();

expect(output).toEqual(expected);

});

it('should Load the flight with the flightId', (done) => {

let output = saga.next(departure).value;

let expected = call(loadFlight, departure.flightID);

done();

expect(output).toEqual(expected);

});

it('should load the forecast with the departure date', (done) => {

output = saga.next(flight).value;

let expected = call(loadForecast, departure.date);

done();

expect(output).toEqual(expected);

});

it('should put Fetch dashboard success', (done) => {

output = saga.next(forecast, departure, flight ).value;

let expected = put({type: 'FETCH\_DASHBOARD\_SUCCESS', payload: {forecast, flight, departure}});

const finished = saga.next().done;

done();

expect(finished).toEqual(true);

expect(output).toEqual(expected);

});

});

Chúng ta cần 1 số lưu ý ở đây:

* Đảm bảo import tất cả các effect và các helpers mà chúng ta sẽ test.
* Khi lưu một giá trị vào yeild, cần truyền dữ liệu mô phỏng (mock) tới function tiếp theo. Hãy chú ý tới ví dụ 3, 4, 5 phía trên.
* Generator du chuyển đến dòng tiếp theo sau yeild khi next() được gọi. Đây là lí do vì sao chúng ta sử dụng saga.next().value ở đây.
* Trình tự bên trong saga là 1 set ràng buộc, nếu thay đổi các bước bên trong, test sẽ fail.

**Kết luận**

So sánh với Thunk, tiếp cận Redux-Saga khó khăn hơn 1 chút, do chúng ta cần hiểu về generator đồng thời cần phải hiểu các khái niệm mới mà saga đưa ra. Thế nhưng sau khi làm quen với Saga, những hiệu quả mà nó mang lại thật sự khác biệt. Việc tách side effect hoàn toàn khỏi action khiến việc testing và maintain trở nên dễ dang hơn.

**Source**

<https://wecodetheweb.com/2016/10/01/handling-async-in-redux-with-sagas/> <https://medium.freecodecamp.com/async-operations-using-redux-saga-2ba02ae077b3> <https://github.com/andresmijares/async-redux-saga> <http://async-redux-saga.surge.sh/>