EBook Gratis

APRENDIZAJE redux

Free unaffiliated eBook created from
Stack Overflow contributors.

#redux

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: redux

It is an unofficial and free redux ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official redux.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con redux

Observaciones

Redux es una biblioteca de JavaScript que implementa el contenedor de estado de la arquitectura basada en Flux.

Redux se puede describir en tres principios fundamentales:

1. Única fuente de verdad (tienda única).
2. El estado es de solo lectura (se necesitan acciones para señalar el cambio)
3. Los cambios se realizan con funciones puras (esto crea un nuevo estado según las acciones)

Partes principales:

- constructor de tienda
- store.dispatch (acción)
- middleware
- reductores

Redux es sorprendentemente simple. Utiliza una función llamada reductor (un nombre derivado del método de reducción de JavaScript) que toma dos parámetros: una acción y un estado siguiente.

El reductor tiene acceso al estado actual (que pronto será anterior), aplica la acción dada a ese estado y devuelve el siguiente estado deseado.

Los reductores están diseñados para ser puras funciones; Es decir, no producen efectos secundarios. Si pasa los mismos valores de entrada a un reductor 100 veces, obtendrá exactamente el mismo valor de salida 100 veces. No pasa nada raro. Son completamente predecibles. Como alguien con una prominente nota adhesiva de "NO SORPRESA" en mi monitor, esta es una idea maravillosa para contemplar.

Los reductores no almacenan el estado y NO mutan el estado. Se pasan el estado, y vuelven estado. Así es como se ven los reductores en acción.
http://redux.js.org/docs/basics/Reducers.html

Referencia: http://redux.js.org/docs/introduction/Motivation.html

Versiones

Versiones	Fecha de lanzamiento
v3.6.0	2016-09-04
v3.5.0	2016-04-20

Versiones	Fecha de lanzamiento
v3.4.0	2016-04-09
v3.3.0	2016-02-06
v3.2.0	2016-02-01
v3.1.0	2016-01-28
v3.0.0	2015-09-13
v2.0.0	2015-09-01
v1.0.0	2015-08-14

Examples

Instalación o configuración

Instalación básica:

Puede descargar el archivo javascript de redux, usando este enlace .

También puedes instalar redux, usando bower :

```
bower install https://npmcdn.com/redux@latest/dist/redux.min.js
```

A continuación, debe incluir redux a su página:

```
<html>
  <head>
    <script type="text/javascript" src="/path/to/redux.min.js"></script>
  </head>
  <body>
    <div id="app"></div>
    <script>
      // Redux is available as `window.Redux` variable.
    </script>
  </body>
</html>
```

Instalación de Npm:

Si está utilizando npm , para instalar redux, debe ejecutar:

```
npm install redux --save
```

A continuación, para usar redux, debe exigirlo (asumiendo que está usando un agrupador de módulos, como un paquete web):

```
var redux = require('redux');
```

O si estás usando el transpiler es6, como babel :

```
import redux from 'redux';
```

Ejemplo de Vanilla Redux (sin React u otros)

Puedes ver la demo en ejecución haciendo clic aquí .

HTML:

```
<p>
  <span>Counter State</span><br />
  (<em>Will increase each minute</em>):
  <p>
    <span id="counter-state" style="font-weight: bolder"></span>
  </p>
</p>

<p>
  <button id="increment-action">+ Increase +</button>
  <button id="decrement-action">- Decrease -</button>
</p>
```

LÓGICA REDUX:

```
// ------------------------ reducer helpers -------------------------
let reducers = {}

let addReducer = (reducers, actionType, reducer) =>
  reducers[actionType] = (state, action) => {
    if (action.type == actionType) {
      return reducer(state, action)
    }
  }

let reducer = (state, action) => {
  if (reducers[action.type]) {
    return reducers[action.type](state, action)
  }
  return state
}


// ------------------------ redux setup ------------------------

const {
  createStore,
  applyMiddleware
  } = Redux


// apply logging middleware (not necessary)
// open the console to see the action logger output on each time new action dispatched
const actionLogger = ({ dispatch, getState }) => next => action => {
```

```
  console.log("action logger: action.type=%s state=%d", action.type, getState())
  return next(action)
}


// ------------------------ reducers ------------------------
// those will be creating new states and returning it,
// depending on the dispatched actions
addReducer(reducers, 'INCREMENT', (state, action) => ++state)
addReducer(reducers, 'DECREMENT', (state, action) => --state)


const DEFAULT_STATE = 0

const store = createStore(
  reducer,
  DEFAULT_STATE,
  applyMiddleware(actionLogger)
);


console.log(createStore)


// ------------------------ rendering ------------------------
let render = () => document.getElementById('counter-state').innerHTML = store.getState()

//
// IMPORTANT BINDING:
//
// store will be dispatching the new state to the render method each time the state changes
//
store.subscribe(render)

//
// render with the state for the first time
//
render()



// ------------------------ redux actions ------------------------

// continously increment the counter (the state) each second
setInterval(() => store.dispatch({type: 'INCREMENT'}), 1000)

// only increment the counter on click to the increase button
document
  .getElementById('increment-action')
  .addEventListener('click', () => store.dispatch({type: 'INCREMENT'}))

// only decrement the counter on click to the decrease button
document
  .getElementById('decrement-action')
  .addEventListener('click', () => store.dispatch({type: 'DECREMENT'}))
```

Lea Empezando con redux en línea: https://riptutorial.com/es/redux/topic/1101/empezando-con-redux

Capítulo 2: Cómo enlazar redux y reaccionar.

Sintaxis

- `<Provider store>`
- `connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])`

Parámetros

Argumento	Descripción
almacenar	Tienda redux
mapStateToProps	Mapeo provisto por el usuario: `(state, ownProps) => resultProps`

Examples

Proveedor

Para vincular fácilmente su tienda Redux a sus componentes React puede usar una biblioteca adicional: react-redux .

Primero, debe envolver su aplicación en un `Provider` , que es un componente que pasa a su tienda para que lo utilicen los componentes secundarios:

```
import { Provider } from 'react-redux';

/// ... store = createStore()

const App = () => (
  <Provider store={store}>
    <MyComponent>
  </Provider>
)
```

Mapa del estado a las propiedades

Después de ajustar su aplicación al proveedor, puede usar la función de `connect` para suscribir el componente para almacenar cambios y proporcionar la asignación entre las propiedades de estado de Redux y las propiedades de los componentes de React:

```
import { connect } from 'react-redux';

const MyComponent = ({data})  => (
  <div>{data}</div>
);
```

```
const mapStateToProps = (state, ownProps) => ({
  data: state.myComponentData
});

connect(mapStateToProps)(MyComponent);
```

Memorización de datos derivados.

En tu tienda de redux tienes los datos en bruto. Algunas veces los datos en bruto son todo lo que necesita, pero otras veces necesita obtener nuevos datos a partir de los datos en bruto, a menudo combinando partes de los datos en bruto.

Un caso de uso común para derivar datos es filtrar una lista de datos en función de un criterio, donde tanto la lista como los criterios pueden cambiarse.

El siguiente ejemplo implementa mapStateToProps y filtra una lista de cadenas para mantener a aquellos que coinciden con una cadena de búsqueda, produciendo una nueva propiedad `filteredStringList` prop que puede ser procesada por un componente React.

```
// Example without memoized selectors
const mapStateToProps(state) => {
    const {stringList, searchString} = state;

    return {
        filteredStringList: stringList
            .filter(string => string.indexOf(searchString) > -1)
    };
}
```

Para mantener los reductores simples, solo debe mantener la lista de datos y los criterios de filtrado en la tienda, lo que significa que debe derivar los datos sobre el tiempo de lectura (como hicimos en el ejemplo anterior).

La obtención de datos sobre el tiempo de lectura plantea dos problemas:

1. Si los mismos datos se derivan muchas veces, se podría comprometer el rendimiento.
2. Si se necesitan los mismos datos en diferentes componentes, es posible que esté duplicando el código para derivar datos.

La solución es utilizar selectores memorizados que se definen una sola vez. La comunidad Reaccionar sugiere el uso de la biblioteca de NPM Vuelva a seleccionar para crear selectores memoized. En el siguiente ejemplo, obtenemos el mismo resultado que en el primer ejemplo, solo con los selectores memorizados.

```
// Create memoized selector for filteredStringList
import {createSelector} from 'reselect';

const getStringList = state => state.stringList;

const getSearchString = state => state.searchString;

const getFilteredStringList = createSelector(
```

```
    getStringList,
    getSearchString,
    (stringList, searchString) => stringList
        .filter(string => string.indexOf(searchString) > -1)
);

// Use the memoized selector in mapStateToProps
const mapStateToProps(state) => {
    return {
        filteredStringList: getStringList(state)
    };
}
```

Tenga en cuenta que los dos primeros selectores `getStringList` y `getSearchString` *no* están memorizados, porque son tan simples que no proporcionarían ningún aumento de rendimiento. Todavía deben crearse porque debemos pasarlos como dependencias a `createSelector` , para que sepa cuándo reutilizar el resultado memorizado y cuándo calcular un nuevo resultado.

El selector memorizado utilizará la función pasada como el último argumento pasado a `createSelector` para calcular los datos derivados (en nuestro ejemplo, la función que devuelve la lista de cadenas filtradas). Cada vez que se llama al selector memoized, si las dependencias no se cambian desde la última vez que se llamó, (en nuestro ejemplo `stringList` y `searchString`), el selector memoized devolverá el resultado anterior, ahorrando el tiempo que tardaría en volverlo a calcular.

Puede pensar en los selectores (memorizados o no memorizados) como captadores para el estado de la tienda, al igual que los creadores de acciones son definidores.

Puede encontrar más ejemplos sobre el cálculo de datos derivados en la sección de recetas de la documentación de Redux .

Lea Cómo enlazar redux y reaccionar. en línea: https://riptutorial.com/es/redux/topic/6621/como-enlazar-redux-y-reaccionar-

Capítulo 3: Flujo de datos asíncrono

Examples

Redux-thunk: conceptos básicos

Si bien el propio redux es completamente síncrono, puede usar un middleware como `redux-thunk` para manejar acciones asíncronas.

> Un "procesador" es otro nombre para una devolución de llamada. Es una función que generalmente se pasa como un argumento para ser llamado en un momento posterior.

Para usar, aplique el middleware a su tienda de redux:

```
import ReduxThunk from 'redux-thunk';

const store = createStore(
    reducer,
    applyMiddleware(ReduxThunk)
);
```

Esto le permite pasar un thunk para `dispatch` lugar de un objeto plano. El middleware reconocerá el procesador y lo llamará. El procesador toma el método de `dispatch` la tienda como un parámetro:

```
// an asynchronous action - "thunk"
// This will wait 1 second, and then dispatch the 'INCREMENT' action
const delayedIncrement = dispatch => setTimeout(() => {
    dispatch({
        type: 'INCREMENT'
    });
}, 1000);

// dispatch the thunk.
// note: no () as we're passing the function itself
store.dispatch(delayedIncrement);
```

Usando redux-thunk con jQuery.ajax

```
const loadUser = userId => dispatch => {
    dispatch({ type: 'USER_LOADING' });
    $.ajax('/users/' + userId, {
        type: 'GET',
        dataType : 'json'
    }).done(response => {
        dispatch({ type: 'USER_LOADED', user: response });
    }).fail((xhr, status, error) => {
        dispatch({ type: 'USER_LOAD_ERROR', status, error });
    });
};
```

Para usar, despacha como cualquier otro creador de acciones:

```
store.dispatch(loadUser(123));
```

Esto dará como resultado que se `USER_LOADING` acción `USER_LOADING` inicial, que se puede usar para mostrar un indicador de carga (si así se desea), y luego de que se haya recibido la respuesta, se `USER_LOADED` una acción `USER_LOAD_ERROR` o `USER_LOAD_ERROR` , dependiendo del resultado Solicitud `$.ajax` .

Middleware

Cuando llamas a `store.dispatch(actionObject)` se maneja de forma síncrona. Es decir, se llamaría a los reductores y se notificaría a los oyentes de la tienda, y las vistas de reacción se volverían a representar en cada acción enviada.

Middleware es lo que le permite retrasar el envío o incluso el envío de diferentes acciones en el medio. Es decir, el middleware hace que sus acciones asíncronas parezcan sincrónicas.

```
const myAsyncMiddleware = (store) => {
    return (next) => {
        return (action) => {
            if(action.type === "ASYNC_ACTION") {
              setTimeout(() => {
                store.dispatch({ type: "ASYNC_ACTION_RESPONSE" });
              }, 1000);
            } else {
              return next(action);
            }
        }
    }
}

const store = createStore(
  reducer,
  applyMiddleware(myAsyncMiddleware)
);
```

Creadores de accion

Otro enfoque para manejar la asincronía en Redux es usar creadores de acción. En Flux, los creadores de acciones son funciones especiales que construyen objetos de acción y los distribuyen.

```
myActionCreator(dispatch) {
  dispatch({ type: "ASYNC_ACTION_START" });
  setTimeout(() => {
    dispatch({ type: "ASYNC_ACTION_END" });
  }, 1000)
}
```

Usar middleware personalizado + Superagente

Este es un ejemplo extraído de esta placa de calderas .

Middleware personalizado:

```
export default function clientMiddleware() {
  return ({dispatch, getState}) => {
    return next => action => {
      if (typeof action === 'function') {
        return action(dispatch, getState);
      }

      const { promise, types, ...rest } = action; // eslint-disable-line no-redeclare
      if (!promise) {
        return next(action);
      }

      const [REQUEST, SUCCESS, FAILURE] = types;
      next({...rest, type: REQUEST});

      const client = new ApiClient();
      const actionPromise = promise(client);
      actionPromise.then(
        (result) => next({...rest, result, type: SUCCESS}),
        (error) => next({...rest, error, type: FAILURE})
      ).catch((error)=> {
        console.error('MIDDLEWARE ERROR:', error);
        next({...rest, error, type: FAILURE});
      });

      return actionPromise;
    };
  };
}
```

Envolviendo la biblioteca de `Superagent` para la llamada a la API:

```
import superagent from 'superagent';
import config from '../config';

const methods = ['get', 'post', 'put', 'patch', 'del'];

function formatUrl(path) {
  const adjustedPath = path[0] !== '/' ? '/' + path : path;
  return adjustedPath;
}

export default class ApiClient {
  constructor(req) {
    methods.forEach((method) =>
      this[method] = (path, { params, data } = {}) => new Promise((resolve, reject) => {
        const request = superagent[method](formatUrl(path));

        if (params) {
          request.query(params);
        }

        if (data) {
          request.send(data);
        }
```

```
        request.end((err, { body } = {}) => err ? reject(body || err) : resolve(body));
      }));
  }
  empty() {}
}
```

Usando redux-thunk con promesas

```
import 'whatwg-fetch';

function checkStatus(response) {
  if (response.status >= 200 && response.status < 300) {
    return response;
  }
  const error = new Error(response.statusText);
  error.response = response;
  throw error;
}

function parseJSON(response) {
  return response.json();
}

function getJSON(endpoint, params) {
  return fetch(endpoint, params)
    .then(checkStatus)
    .then(parseJSON);
}

export function action() {
  return dispatch => getJSON('/example-endpoint')
    .then((result) => {
      dispatch({
        type: GET_SUCCESS,
        result,
      });
    })
    .catch((error) => {
      dispatch({ type: GET_FAILURE, error });
    });
}
```

Lea Flujo de datos asíncrono en línea: https://riptutorial.com/es/redux/topic/3474/flujo-de-datos-asincrono

Capítulo 4: Pruebas de aplicaciones redux

Examples

Redux + Mocha

Redux es muy funcional, por lo que las pruebas unitarias son muy sencillas.

Creador de accion:

```
export function showSidebar () {
  return {
    type: 'SHOW_SIDEBAR'
  }
}
```

Prueba de unidad de creadores de acción:

```
import expect from 'expect'
import actions from './actions'
import * as type from './constants'

describe('actions', () => {
  it('should show sidebar', () => {
    const expectedAction = {
      type: type.SHOW_SIDEBAR
    }
    expect(actions.showSidebar()).toEqual(expectedAction)
  })
})
```

Probando una tienda Redux con Mocha y Chai

```
import { expect } from 'chai';
import { createStore } from 'redux';

describe('redux store test demonstration', () => {
  describe('testReducer', () => {
    it('should increment value on TEST_ACTION', () => {
      // define a test reducer with initial state: test: 0
      const testReducer = (state = { test: 0 }, action) => {
        switch (action.type) {
          case 'TEST_ACTION':
            return { test: state.test + 1 };
          default:
            return state;
        }
      });

      // create a redux store from reducer
      const store = createStore(testReducer);

      // establish baseline values (should return initial state)
```

```
      expect(store.getState().test).to.equal(0);

      // dispatch TEST_ACTION and expect test to be incremented
      store.dispatch({ type: 'TEST_ACTION' });
      expect(store.getState().test).to.equal(1);

      // dispatch an unknown action and expect state to remain unchanged
      store.dispatch({ type: 'UNKNOWN_ACTION' });
      expect(store.getState().test).to.equal(1);
    });
  });
});
```

Lea Pruebas de aplicaciones redux en línea: https://riptutorial.com/es/redux/topic/4059/pruebas-de-aplicaciones-redux

Capítulo 5: Pure Redux - Redux sin ningún marco

Parámetros

Paramenter	Descripción
acción	Debe ser un objeto con al menos la propiedad `type` . Cualquier otra propiedad se puede pasar y será accesible dentro de la función de reductor.

Observaciones

Si no está utilizando paquetes como Webpack y Browserify, cambie la primera línea a:

```
const { createStore } = Redux;
```

O simplemente llámelo directamente desde el Redux global al crear la tienda:

```
const store = Redux.createStore(counter);
```

Examples

Ejemplo completo

index.html

```
<button id="increment">Increment</button>
<button id="decrement">Decrement</button>
<p id="app"></p>
```

---

index.js

```
import { createStore } from 'redux';

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;

    case 'DECREMENT':
      return state - 1;

    default:
      return state;
```

```
  }
}

const store = createStore(counter);

function render() {
  const state = store.getState();
  document.querySelector('#app').innerHTML = `Counter: ${state}`;
}

const incrementButton = document.querySelector('#increment');
const decrementButton = document.querySelector('#decrement');

incrementButton.addEventListener('click', () => {
  store.dispatch({ type: 'INCREMENT' });
});

decrementButton.addEventListener('click', () => {
  store.dispatch({ type: 'DECREMENT' });
});

store.subscribe(() => {
  render();
});

render();
```

Lea Pure Redux - Redux sin ningún marco en línea:
https://riptutorial.com/es/redux/topic/4079/pure-redux---redux-sin-ningun-marco

Capítulo 6: Reductor

Observaciones

Los reductores cambian el estado de la aplicación en función de las acciones disparadas.

El estado es inmutable, lo que significa que los reductores deben ser puros: para la misma entrada, **siempre** se debe obtener la misma salida. Debido a esto, la mutabilidad está prohibida en los reductores.

Examples

Reductor basico

Un reductor básico se vería así:

```
// Import the action types to recognise them
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';


// Set up a default state
const initialState = {
    error: undefined,
    entities: []
};

// If no state is provided, we take the default state
export default (state = initilState, action) => {

    // Based on the type of action received, we calculate the new state
    switch(action.type) {

        // Set the error
        case ACTION_ERROR:
            // Note that we create a new object, copy the current state into it,
            // and then we make the relevant changes, so we don't mutate the state
            return Object.assign({}, state, { error: action.error });

        // Unset any error, and load the entitites
        case ACTION_ENTITIES_LOADED:
            return Object.assign({}, state, {
                entities: action.entities,
                error: undefined
            };

        // Add only one entity. Again, note how we make a new entities array
        // combining the previous one with the new entity
        // instead of directly modifying it, so we don't mutate the state.
        case ACTION_ENTITY_CREATED:
            return Object.assign({}, state, {
                entities: [action.entity].concat(state.entities)
            }):
```

```
        // If the action is not relevant to this reducer, just return the state
        // as it was before.
        default:
            return state;
    }
};
```

Utilizando inmutable

Immutable es una excelente biblioteca que nos proporciona versiones inmutables de los tipos de colecciones más utilizados, como listas, pilas, mapas y más.

Simplifica la manipulación del estado y facilita la realización de cálculos puros y evita la mutación.

Veamos cómo se puede reescribir el reductor básico utilizando las estructuras de Mapa y Lista de Immutable:

```
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';

// Import Immutable
import Immutable from 'immutable';


// Set up a default state using a Map, a structure very similar to objects
// Note that states in Redux can be anything, not just objects
const initialState = Immutable.Map({
    error: undefined,
    entities: Immutable.List()
});

export default (state = initialState, action) => {

    switch(action.type) {

        case ACTION_ERROR:
            return state.set('error', action.error);

        case ACTION_ENTITIES_LOADED:
            return state.merge({
                entities: Immutable.List(action.entities)
                error: undefined
            });

        case ACTION_ENTITY_CREATED:
            return state.set('entities', state.entities.push(action.entity));

        default:
            return state;
    }
};
```

Como habrás visto, el manejo del estado inmutable se vuelve más fácil al usar Inmutable.

Ejemplo básico utilizando la propagación de ES6

```
// Import the action types to recognize them
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';

// Set up a default state
const initialState = {
    error: undefined,
    entities: [],
    loading: true
};

// If no state is provided, we take the default state
export default (state = initialState, action) => {

    // Based on the type of action received, we calculate the new state
    switch(action.type) {

        // Set the error
        case ACTION_ERROR:
            // We will create new object with state,
            // which should be produced by error action
            return {
                entities: [],
                loading: false,
                error: action.error
            };

        // Unset any error, and load the entities
        case ACTION_ENTITIES_LOADED:
            return {
                entities: action.entities,
                error: undefined,
                loading: false
            };

        // Add only one entity. We will use spread operator (...) to merge
        // objects properties and to create new entity
        case ACTION_ENTITY_CREATED:
            return {
                ...state,
                entities: [action.entity].concat(state.entities)
            };

        // Every action is processed by each reducer,
        // so we need to return same state if we do not want to mutate it
        default:
            return state;
    }
};
```

Lea Reductor en línea: https://riptutorial.com/es/redux/topic/6615/reductor

Creditos

S. No	Capítulos	Contributors
1	Empezando con redux	1ven, Arijit Bhattacharya, Community, Inanc Gumus, jpdelatorre , Random User, uddhab, Vanuan
2	Cómo enlazar redux y reaccionar.	ArneHugo, Vanuan
3	Flujo de datos asíncrono	Ali Sepehri.Kh, Arijit Bhattacharya, Franco Risso, Ming Soon, rossipedia, Vanuan
4	Pruebas de aplicaciones redux	Mario Tacke, Shuvo Habib, Vanuan
5	Pure Redux - Redux sin ningún marco	Guilherme Nagatomo, Vanuan, Vishnu Y S
6	Reductor	Jurosh, Marco Scabbiolo