Tabla de contenido

Redux es un contenedor predecible del estado de aplicaciones JavaScript.

Te ayuda a escribir aplicaciones que se comportan de manera consistente, corren en distintos ambientes (cliente, servidor y nativo), y son fáciles de probar. Además de eso, provee una gran experiencia de desarrollo, gracias a edición en vivo combinado con un depurador sobre una línea de tiempo.

Puedes usar Redux combinado con React, o cual cualquier otra librería de vistas. Es muy pequeño (2kB) y no tiene dependencias.

> Aprende Redux con su creador (en inglés): Getting Started with Redux (30 vídeos gratuitos)

Testimonios

> “Love what you’re doing with Redux” Jing Chen, creador de Flux
>
> “I asked for comments on Redux in FB's internal JS discussion group, and it was universally praised. Really awesome work.” Bill Fisher, author of Flux document Bill Fisher, autor de la documentación de Flux
>
> “It's cool that you are inventing a better Flux by not doing Flux at all.” André Staltz, creador de Cycle

Experiencia de desarrollo

Escribí Redux mientras trabajaba en mi charla de React Europe llamada "Hot Reloading with Time Travel". Mi meta era crear una librería de manejo de estado con una API mínima, pero completamente predecible, de forma que fuese posible implementar logging, hot reloading, time travel, aplicaciones universales/isomórficas, grabar y repetir.

Influencias

Redux evoluciona las ideas de Flux, pero evitando su complejidad tomando cosas de Elm.

Ya sea que los hayas usado o no, solo toma unos minutos para empezar a usar Redux.

Instalación

Para instalar la versión estable:

```
npm i -S redux
```

Normalmente también vas a querer usar la conexión a React y las herramientas de desarrollo.

```
npm i -S react-redux
npm i -D redux-devtools
```

Esto asumiendo que estas usando npm como administrador de paquetes con un empaquetador de módulos como Webpack o Browserify para usar módulos de CommonJS.

Si todavía no usas npm o algún empaquetador de módulos moderno, capaz prefiera el paquete en UMD que define `Redux` como un objeto global, puedes usar una desde cdnjs. *No* recomendamos este enfoque para ninguna aplicación seria, ya que la mayoría de las librerías complementarias a Redux está solo disponibles en npm.

El Gist

Todo el estado de tu aplicación esta almacenado en un único árbol dentro de un único *store*. La única forma de cambiar el árbol de estado es emitiendo una *acción*, un objeto describiendo que ocurrió.

Para especificar como las acciones transforman el árbol de estado, usas *reducers* puros.

¡Eso es todo!

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

/**
 * Esto es un reducer, una función pura con el formato (state, action) => newState.
 * Describe como una acción transforma el estado en el nuevo estado.
 *
 * La forma del estado depende de tí: puede ser un primitivo, un array, un objeto, o incl
 *
 * En este ejemplo, usamos `switch` y strings, pero puedes usar cualquier forma que desee
 */
function counter(state = 0, action) {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
```

```
      return state - 1;
    default:
      return state;
    }
}

// Creamos un store de Redux almacenando el estado de la aplicación.
// Su API es { subscribe, dispatch, getState }.
let store = createStore(counter);

// Puedes suscribirte manualmente a los cambios, o conectar tu vista directamente
store.subscribe(() => {
  console.log(store.getState())
});

// La única forma de modificar el estado interno es despachando acciones.
// Las acciones pueden ser serializadas, registradas o almacenadas luego para volver a ej
store.dispatch({ type: 'INCREMENT' });
// 1
store.dispatch({ type: 'INCREMENT' });
// 2
store.dispatch({ type: 'DECREMENT' });
// 1
```

En vez de modificar el estado directamente, especificas las modificaciones que quieras que ocurren con objetos planos llamados *acciones*. Entonces escribes una función especial llamada *reducer* que decide como cada acción transforma el estado de la aplicación.

Si vienes de Flux, hay una única diferencia importante que necesitas entender. Redux no tiene Dispatcher o soporta múltiples stores. En cambio, hay un único store con una única función reductora. Cuando tu aplicación crezca, en vez de agregar más stores, divides tu reducer principal en varios reducers pequeños que operan de forma independiente en distintas partes del árbol de estado. Esto es exactamente como si solo hubiese un componente principal en una aplicación de React, pero esta compuesta de muchos componentes pequeños.

Esta arquitectura puede parecer una exageración para una aplicación de contador, pero los hermoso de este patrón es los bien que escala en aplicaciones grandes y complejas. También permite herramientas de desarrollo muy poderosas, ya que es posible registrar cada modificación que las acciones causan. Podrías registrar la sesión de un usuario y reproducirlas simplemente ejecutando las mismas acciones.

Aprende Redux con su Creador (en inglés)

Getting Started with Redux es un curso de 30 vídeos explicados por Dan Abramov, autor de Redux. Esta diseñado para complementar las partes "Básicas" de la documentación mientras incluye ideas adicionales sobre inmutabilidad, pruebas, buenas prácticas de

Redux, y como usar Redux con React.**Este curso es gratuito y siempre lo va a ser**.

> “Great course on egghead.io by @dan_abramov - instead of just showing you how to use #redux, it also shows how and why redux was built!” Sandrino Di Mattia
>
> “Plowing through @dan_abramov 'Getting Started with Redux' - its amazing how much simpler concepts get with video.” Chris Dhanaraj
>
> “This video series on Redux by @dan_abramov on @eggheadio is spectacular!” Eddie Zaneski
>
> “Come for the name hype. Stay for the rock solid fundamentals. (Thanks, and great job @dan_abramov and @eggheadio!)” Dan
>
> “This series of videos on Redux by @dan_abramov is repeatedly blowing my mind - gunna do some serious refactoring” Laurence Roberts

Así que ¿Qué estas esperando?

¡Mira los 30 vídeos gratuitos!

Sí se gusto el curso, considera soportar a Egghead comprando una suscripción. Los suscriptos tiene acceso al código fuentes de los ejemplos en cada uno los vídeos, así como un montón de lecciones avanzados en otros temas, incluyendo JavaScript en profundidad, React, Angular, y más. Muchas profesores de Egghead son también autores de librerías de código abierto, así que comprando una suscripción es una buena forma de ayudarlos por todo el trabajo que hicieron.

Documentación

- Introducción
- Básico
- Avanzado
- Recetas
- Solución de problemas
- Glosario
- Referencia API

Ejemplos

- Contador

- TodoMVC
- Todos con Deshacer
- Asíncrono
- Universal
- Mundo real
- Carrito de compra
- Vista de árbol

Si eres nuevo en el ecosistema de NPM y tiene problemas iniciando un proyecto, o no estas seguro de donde pegar el gist de arriba, revisa simples-redux-example que usa React junto a React y Browserify.

Discusiones

Únete al canal #redux de la comunidad en Discord Reactiflux.

Agradecimientos

- The Elm Architecture por una gran introducción al modelo de actualización de estado con reducers;
- Turning the database inside-out por explotarme la mente;
- Developing ClojureScript with Figwheel por convencerme que la re-evaluaciǿn debe "simplemente funcionar";
- Webpack por Hot Module Replacement;
- Flummox por enseñarme un enfoque a Flux sin boilerplate or singletons;
- disto por la prueba de concepto de hot reloadable Stores;
- NuclearJS por probar que esta arquitectua puede tener buen rendimiento;
- Om por popularizar la idea de un único Store;
- Cycle por demostrar que tan común una función puede ser la mejor herramienta;
- React por la innovación pragmática.

Agradecimientos especiales a Jamie Paton por liberar `redux` como nombre de módulo en NPM.

Registro de cambios

Este proyecto se adhiere al Versionamiento Semántico.

Cada lanzamiento, junto a sus instrucciones de migración, están documentados en la

página de Github de Lanzamientos.

Patrocinadores

El trabajo con Redux fue financiado por la comunidad.

Algunos de las compañías más destacadas que hicieron esto posible:

- Webflow
- Chess iX

Ve la lista completa de patrocinadores de Redux.

Licencia

MIT

Introducción

Motivación

Como los requisitos en aplicaciones JavaScript de una sola página se están volviendo cada vez más complicados, **nuestro código, mas que nunca, debe manejar el estado**. Este estado puede incluir respuestas del servidor y datos cacheados, así como datos creados localmente que todavía no fueron guardados en el servidor. El estado de las UI también se volvió más complejo, al necesitar mantener la ruta activa, el tab seleccionado, si mostrar o no un spinner, si deben mostrarse los controles de paginación o no.

Controlar ese cambiante estado es difícil. Si un modelo puede actualizar otro modelo, entonces una vista puede actualizar un modelo, el cual actualiza otro modelo, y esto causa que otra vista se actualice. En cierto punto, ya no se entiende que esta pasando en la aplicación ya que **perdiste control sobre el cuándo, el por qué y el cómo de su estado**. Cuando un sistema es opaco y no determinista, es difícil reproducir errores o agregar nuevas características.

Como si no fuera suficientemente malo, considera que **los nuevos requisitos son comunes en el desarrollo front-end**. Como desarrolladores, se espera que manejemos actualizaciones, renderizado en el servidor, obtener datos antes de realizar cambios de rutas, y más cosas. Nos encontramos tratando de controlar algo más complejo que nunca, e inevitablemente nos preguntamos: es tiempo de rendirse? La respuesta es *no*.

Esta complejidad es difícil de manejar debido a que **estamos mezclando dos conceptos** que son difícil de entender para la mente humana: **mutación y asincronicidad**. Les llamo Mentos y Coca-Cola. Ambos son geniales separados, pero juntos pueden causar un gran lío. Librerías como React tratan de resolver este problema en la capa de la vista removiendo tanto la asincronicidad como la manipulación directa del DOM. De todas formas, controlar el estado de tus datos es todavía tu responsabilidad. Acá es donde entra Redux.

Siguiendo los pasos de Flux, CQRS y Event Sourcing, **Redux intenta hacer predecibles las mutaciones del estado** imponiendo ciertas restricciones en cómo y cuándo pueden realizarse las actualizaciones. Estas restricciones se reflejan en los tres principios de Redux.

Tres Principios

Redux puede ser descrito en tres principios fundamentales:

Única fuente de la verdad

El estado de toda tu aplicación esta almacenado en un árbol guardado en un único store.

Esto hace fácil crear aplicaciones universales, ya que el estado en tu servidor puede ser serializado y enviado al cliente sin ningún esfuerzo extra. Como un único árbol de estado también hace más fácil depurar una aplicación; te permite también mantener el estado de la aplicación en desarrollo, para un ciclo de desarrollo más veloz. Algunas funcionalidades que históricamente han sido difíciles de implementar - como Deshacer/Rehacer, por ejemplo - se vuelven triviales si todo tu estado se guarda en un solo árbol.

```
console.log(store.getState())
/* Imprime
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Considerar usar Redux',
      completed: true
    },
    {
      text: 'Mantener todo el estado en un solo árbol',
      completed: false
    }
  ]
}
*/
```

El estado es de solo lectura

La única forma de modificar el estado es emitiendo una acción, un objeto describiendo que ocurrió.

Esto te asegura que ni tu vista ni callbacks de red van a modificar el estado directamente. En vez de eso, expresan un intento de modificar el estado. Ya que todas las modificaciones están centralizadas y suceden en un orden estricto, no hay que preocuparse por una carrera entre las acciones. y como las acciones son objetos planos, pueden ser registrados,

serializados y almacenados para volver a ejecutarlos por cuestiones de depuración y pruebas.

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
});

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
});
```

Los cambios se realizan con funciones puras

Para especificar como el árbol de estado es transformado por las acciones, se utilizan reducers puros.

Los reducers son funciones puras que toman el estado anterior y una acción, y devuelven un nuevo estado. Recuerda devolver un nuevo objeto de estado en vez de modificar el anterior. Puedes empezar con un único reducer, y mientras tu aplicación crece, dividirlo en varios reducers pequeños que manejan partes específicas del árbol de estado. Ya que los reducers son funciones puras, puedes controlar el orden en que se ejecutan, pasarle datos adicionales, o incluso hacer reducers reusables para tareas comunes como paginación.

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter;
    default:
      return state;
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          text: action.text,
          completed: false,
        },
      ];
    case 'COMPLETE_TODO':
      return [
        ...state.slice(0, action.index),
        Object.assign({}, state[action.index], {
          completed: true
        }),
```

```
          ...state.slice(action.index + 1),
        ];
      default:
        return state;
    }
  }

  import { combineReducers, createStore } from 'redux';
  let reducer = combineReducers({ visibilityFilter, todos });
  let store = createStore(reducer);
```

¡Y eso es todo! Ahora sabes de que se trata Redux.

Herencia

Redux tiene una herencia mixta. Es similar a ciertos patrones y tecnologías, pero también es diferente en varias formas importantes. Vamos a ver las similitudes y diferencias debajo.

Flux

Puede Redux ser considerado una implementación de Flux? Sí y no.

(No te preocupes, los creadores de Flux lo aprueba, si es lo que necesitas saber.)

Redux se inspiro en muchas de las cualidades importantes de Flux. Como Flux, Redux te hacer concentrar la lógica de actualización de modelos en una capa específica de tu aplicación ("stores" en Flux, "reducers" en Redux). En vez de dejar al código de la aplicaiøn modificar los datos directamente, ambos te hacen describir cada mutación como objetos planos llamados "acciones".

A diferencia de Flux, **en Redux no existe el concepto de Dispatcher**. Esto es porque se basa en funciones puras en vez de emisores de ventos, y las funciones puras son fáciles de componer y no necesitan entidades adicionales para controlarlas. Dependiendo de como veas Flux, puedes ver esto tanto como una desviación o un detalle de implementación. Flux siempre fue descripto como `(state, action) => state`. En ese sentido, Redux es una verdadera arquitectura Flux, pero más simple gracias a las funciones puras.

Otra diferencia importante con Flux es que **Redux asume que nunca modificas los datos**. Puede usar objetos planos o arrays para tu estado y esta perfecto, pero modificarlos dentro de los reducers no es recomendable. Siempre deberías devolver un nuevo objeto, lo cual es simple con la propuesta del operador spread, o con librerías como Immutable.

Mientras que es tecnicamente *posible* escribir reducers impuros que modifican los datos en algunos por razones de rendimiento, desalentamos activamente hacer eso. Características de desarrollo como time travel, registrar/rehacer, o hot reloading pueden romperse. Además características como inmutabilidad no parecen causar problemas de rendimiento en aplicaciones reales, ya que, como Om demostró, incluso si pierdes la posibilidad de usar asignación en objetos, todavía ganas por evitar re-renders y recalculos caros, debido a que sabes exactamente que cambio gracias a los reducers puros.

Elm

Elm es un lenguaje de programación funcional inspirado por Haskell y creado por Evan

Czaplicki. Utiliza una arquitectura 'modelo vista actualizador", donde el actualizador tiene el siguiente formato: `(action, state) => state`. Los "actualizadores" de Elm sirven para el mismo propósito que los reducers en Redux.

A diferencia de Redux, Elm es un lenguaje, así que puede beneficiarse de de muchas cosas como pureza forzada, tipado estático, inmutabilidad, y coincidencia de patrones (usando la expresión `case`). Incluso si no planeas usar Elm, deberías leer sobre la arquitectura de Elm, y jugar con eso. Hay un interesante playground de librerías de JavaScript que implementa ideas similares. ¡Debemos mirar ahí por inspiración en Redux! Una forma de acercarnos al tipado estático de Elm es usando una solución de tipado gradual como Flow.

Immutable

Immutable es una librería de JavaScript que implementa estructuras de datos persistentes. Es una API JavaScript idiomática y performante.

Immutable y la mayoría de las librerías similares son ortogonales a Redux. ¡Sientete libre de usarlos juntos!

Redux no se preocupa por si guardas el estado en objetos planos, objetos de Immutable o cualquier otra cosa. Probablemente quieras un mecanismo de (de)serialización para crear aplicaciones universales y rehidratar su estado desde el servidor, pero aparte de eso, puedes usar cualquier librería de almacenamiento de datos *mientras soporte inmutabilidad*. Por ejemplo, no tiene sentido usar Backbone para tu estado de Redux, ya que estos son mutables.

Ten en cuenta que que incluso si tu librería inmutable soporta punteros, no deberías usarlos en una aplicación de Redux. Todo el árbol de estado debería ser considerado de solo lectura, y deberías usar Redux para actualizar el estado, y suscribirse a los cambios. Por lo tanto, actualizar mediante recuerdos no tiene sentido en Redux. **Si tu único caso de uso para los punteros es desacoplar el árbol de estado del el árbolde UI y gradualmente refinar tus punteros, deberías revisar los selectores mejor.** Los selectores son funciones getter combinables. Mira reselect para una muy buena y concisa implementación de selectores combinables.

Baobab

Baobab es otra popular librería para implementar inmutabilidad para actualizar objetos planos en JavaScript. Aunque puedes usarlo con Redux, hay muy pocos beneficios de usarlos juntos.

La mayorías de las funcionalidades que provee Baobab están relacionados con actualizar

los datos con punteros, pero Redux impone que la única forma de actualizar los datos es despachando acciones. Por lo tanto, resuelven el mismo problema de forma diferente, y no se complementar uno con otro.

A diferencia de Immutable, Baobab todavía no implemente ninguna estructura de datos especialmente eficiente, así que no ganas nada realmente por usarlo junto a Redux. Es más fácil simplemente usar objetos planos en su lugar.

Rx

Reactive Extensions (y su reescritura moderna en proceso) es una formas magnífica de manejar la complejidad de aplicaciones asíncronas. De hecho hay un esfuerzo de crear una librería para controlar la interacción entre humano y computadora como observables independientes.

¿Tiene sentido usar Redux junto con Rx? ¡Seguro! Funcionan genial juntos. Por ejemplo, es fácil exponer el store de Redux como un observable:

```
function toObservable(store) {
  return {
    subscribe({ onNext }) {
      let dispose = store.subscribe(() => onNext(store.getState()));
      onNext(store.getState());
      return { dispose };
    },
  };
}
```

De forma similar, puedes componer diferentes streams asíncronos para convertirlos en acciones antes de enviarlos al `store.dispatch()`.

La pregunta es: de verdad necesitas REdux si ya usar Rx? Probablemente no. No es dificil reimplementar Redux en Rx. Algunos dicen, que son solo 2 líneas usando el método `.scan()` de Rx. ¡Y probablemente lo sea!

Si tienes dudas, revisa el código fuente de Redux (no hay mcuho ahí), así como su ecosistema (por ejemplo, las herramientas de desarrolladores). Si no te interesa tanto eso y quieres que los datos reactivos simplemente fluyan, probablemente quieras usar algo como Cycle en su lugar, o incluso combinarlos con Redux. ¡Déjanos saber como resulta eso!

Ecosistema

Redux es una librería pequeña, pero su API fue elegida cuidadosamente para permitir un ecosistema de herramientas y extensiones.

Para una lista completa de todo lo relacionado a Redux, recomendamos Awesome Redux. Contiene ejemplos, boilerplates, middlewares, librerías utilitarias y más.

En esta página solo mostramos algunas de ellas que los colaboradores de Redux han seleccionado personalmente ¡No dejes que eso te desaliente a probar el resto! El ecosistema esta creciendo muy rápido, y tenemos un tiempo limitado para ver y probar todo. Considera estas como "elegidas por el staff", y no dudes en enviar un PR si haces algo increible con Redux.

Aprendiendo Redux (en español)

- Introducción a Redux.js - Introducción a conceptos de Redux.js
- Combinando React.js y Redux.js - Explicación de como usar conjuntamente estas dos tecnologías.
- Middlewares en Redux.js - Explicación de como hacer middlewares propios para Redux.js
- Pruebas unitarias en Redux.js - Ejemplos de como hacer pruebas a nuestro código de Redux.js.
- Ruteo en aplicaciones de Redux y React.js - Explicación de como manejar las rutas de una aplicación hecha con Redux y React.js.
- Estructura de archivos Ducks para Redux.js - Buena práctica de como organizar creadores de acciones, reducers y tipos de acciones en módulos.
- Glosario de términos de Redux - Colección de términos usados en Redux junto a su explicación.

Aprendiendo Redux (en inglés)

Vídeos

- Getting Started with Redux - Aprende los conceptos básicos de Redux directamente de su creador (30 vídeos gratuitos)

Aplicaciones de ejemplo

- SoundRedux - Un cliente de SoundCloud hecho con Redux
- Shopping Cart (Flux Comparison) - Un ejemplo de carrito de completo de Flux Comparison

Tutoriales y artículos

- Redux Tutorial — Aprende a usar Redux paso a paso
- Redux Egghead Course Notes — Notas del curso en vídeo de Egghead
- What the Flux?! Let's Redux. — Una introducción a Redux
- A cartoon intro to Redux — Una explicación visual del flujo de datos de Redux
- Understanding Redux — Aprende los conceptos básicos de Redux
- Handcrafting an Isomorphic Redux Application (With Love) — Una guía para crear aplicaciones universales con rutas y data fetching
- Full-Stack Redux Tutorial — Una guía completa de desarrollo TDD con Redux, React e Immutable
- Understanding Redux Middleware — Una guía en profundidad de como implementar middlewares de Redux
- A Simple Way to Route with Redux — Una introducción a Redux Simple Router

Charlas

- Live React: Hot Reloading and Time Travel — Ve como las restricciones impuestas por Redux hacen hot reloading con time travel fácil
- Cleaning the Tar: Using React within the Firefox Developer Tools — Aprender como gradualmente migrar una aplicación MVC existente a Redux
- Redux: Simplifying Application State — Una introducción a la arquitectura de Redux

Usando Redux

Conexiones

- react-redux — React
- ng-redux — Angular
- ng2-redux — Angular 2
- backbone-redux — Backbone
- redux-falcor — Falcor
- deku-redux — Deku

Middleware

- redux-thunk — La forma más fácil de trabajar con acciones asíncronas
- redux-promise — Middleware de promesas compatible con FSA
- redux-rx — Utilidades de RxJS para Redux, inlcuye un middleware para Observables
- redux-logger — Registra cada acción de Redux y el siguiente estado
- redux-immutable-state-invariant — Advierto sobre modificaciones al estado en desarrollo
- redux-analytics — Middleware de analiticas para Redux
- redux-gen — Middleware de generadores para Redux
- redux-saga — Un modelo alternativo para efectos secundarios en aplicaciones de Redux

Ruteo

- react-router-redux — Manten React Router y Redux sincronizados fácilmente
- redux-router — Conecta Redux con React Router

Componentes

- redux-form — Mantén el estado de los formularios de React en Redux
- react-redux-form — Crea formularios facilmente en React con Redux

Potenciadores

- redux-batched-subscribe — Customiza batching y debouncing a los suscriptos al store
- redux-history-transitions — Transiciones basadas en History en acciones arbitrarias
- redux-optimist — Aplicac acciones que luego pueden ser revertidas
- redux-undo — Historial de acciones y deshacer/rehacer sin esfuerzo
- redux-ignore — Ignora acciones de Redux
- redux-recycle — Reinicia el estado de Redux en ciertas acciones
- redux-batched-actions — Despacha varias acciones con una sola notificación a los suscriptos
- redux-search — Automáticamente indexa recursos en un web worker y busca en ellos sin bloquear la aplicación
- redux-electron-store — Sincroniza el store de Redux entre varios procesos de Electron
- redux-loop — Ejecuta acciones de forma secuencial retornandolas desde tus reducers

Utilidades

- reselect — Selector de datos derivados eficiente inspirado por NuclearJS

- normalizr — Normaliza respuestas de API anidades para consumirlas facilmente desde un reducer
- redux-actions — Disminuye el boilerplate al escribir reducers y creadores de acciones
- redux-act — Una librería opinionada para crear acciones y reducers
- redux-transducers — Utilidades de transductores para Redux
- redux-immutable — Usado para crear un equivalente a la funcíon `combineReducers` que funcione con un estado de Immutable.js
- redux-tcomb — Estado y acciones immutable y tipadas
- redux-mock-store — Simula un store de Redux para probar tu aplicación

Herramientas de desarrollo

- Redux DevTools — Un logger de acciones con una UI para time travel, hot reloading y manejo de errores para reducers, mostrada por primera vez en React Europe
- Redux DevTools Extension — Una extensión de Chrome que envuelve las Redux DevTools y provee funcionalidades adicionales

Convenciones de la Comunidad

- Flux Standard Action — Un estándar de acciones de Flux amigable con humanos
- Canonical Reducer Composition — Un estándar opinionado para combinar reducers anidados
- Ducks: Redux Reducer Bundles — Una propuesta para empaquetar reducers, acciones y sus tipos

Traducciones

- Redux — Inglés
- 中文文档 — Chino
- 繁體中文文件 — Chino Trandicional
- Redux in Russian — Ruso

Más

Awesome Redux es una lista más completa de repositorios relacionados con Redux.

Ejemplos

Redux es distribuido con algunos ejemplos en su código fuente.

Counter Vanilla

Ejecuta el ejemplo de Counter Vanilla

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/counter-vanilla
open index.html
```

No requiere un sistema de build o un framework de vista y existe para mostrar la API pura de Redux usando ES5.

Counter

Ejecuta el ejemplo de Counter:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/counter
npm install
npm start

open http://localhost:3000/
```

Este es el más básico ejemplo usando Redux junto a React. Por simpleza, vuelve a renderizar el componente de React manualmente cuando el store cambia. En un proyecto real, deberías usar algo con mejor rendimiento como React Redux.

Este ejemplo incluye pruebas.

Todos

Ejecuta el ejemplo de Todos:

```
git clone https://github.com/reactjs/redux.git
```

```
cd redux/examples/todos
npm install
npm start

open http://localhost:3000/
```

Este es el mejor ejemplo para obtener un conocimiento más profundo de como las actualizaciones de estado funcionan en Redux. Muestra como los reducers pueden delegar el manejo de acciones a otros reducers, y como usar React Redux para generar componentes contenedores para tus componentes presentacionales.

Este ejemplo incluye pruebas.

Todos with Undo

Ejecuta el ejemplo de Todos with Retroceder:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/todos-with-undo
npm install
npm start

open http://localhost:3000/
```

Este es una variación del ejemplo anterior. Es practicamente identico, pero además muestra como envolver tus reducers con Redux Undo te permite agregar la funcionalidad de Deshacer/Rehacer a tu aplicaciones con una pocas líneas.

TodoMVC

Ejecuta el ejemplo de TodoMVC:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/todomvc
npm install
npm start

open http://localhost:3000/
```

Este es el clásico ejemplo de TodoMVC. Esta aquí por razones de comparación, pero cubre los mismos puntos que el ejemplo de Todos.

Este ejemplo incluye pruebas.

Shopping Cart

Ejecuta el ejemplo de Shopping Cart:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/shopping-cart
npm install
npm start

open http://localhost:3000/
```

Este ejemplo muestra importantes patrones de Redux que se vuelven más importantes mientras tu aplicación crece. En particular, muestra como guardar entidades de una forma normalizada usando sus IDs, como componer reducer en varios niveles, y como definir selectores junto a los reducers así el conocimiento de la forma del estado está encapsulado. Además muestra como registrar cambios con Redux Logger y despachar acciones con el middleware Redux Thunk.

Tree View

Ejecuta el ejemplo de Tree View:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/tree-view
npm install
npm start

open http://localhost:3000/
```

Este ejemplo muestra como renderizas una vista de un árbol profundamente anidado y representar su estado de forma normalizada así es fácil actualizarlo mediante reducers. Un buen rendimiento al renderizar al hacer que los componentes contenedores se suscriban solo a los nodos del árbol que renderizan.

Este ejemplo incluye pruebas.

Async

Ejecuta el ejemplo de Async:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/async
npm install
npm start

open http://localhost:3000/
```

Este ejemplo incluye leer desde un API asínrono, obtener datos en respuestas a la acciones del usuario, mostrar indicadores de cargando, cachear respuestas e invalidar la cache. Usa el middleware It uses Redux Thunk para encapsular efectos secundarios asíncronos.

Universal

Ejecuta el ejemplo de Universal:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/universal
npm install
npm start

open http://localhost:3000/
```

Esta es las más básica demostración de renderizado en el servidor con Redux y React. Muestra como preparar el estado inicial en el servidor y pasarlo al cliente así se inicia desde el estado existente.

Real World

Ejecuta el ejemplo de Real World:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/real-world
npm install
npm start

open http://localhost:3000/
```

Este es el ejemplo más avanzado. Es grande por diseño. Cubre como mantener entidades en una cache normalizada, implementando un middleware personalizado para llamadas a

API, renderizando páginas parcialmente cargadas, paginación, cacheo de respeustas, mostrar mensajes de error y ruteo. Adicionalmente, incluye las Redux DevTools.

Más Ejemplos

Puedes encontrar más ejemplos en Awesome Redux.

Básico

No te dejes engañar por charlas fancy sobre reducers, middlewares, potenciadores de store, etc. — Redux es increíblemente simple. Si alguna vez hiciste una aplicación Flux, te vas a sentir como en casa. Y si eres nuevo en Flux ¡También es fácil!

En este guía, vamos a recorrer el proceso de crear una aplicación de TODOs simple.

Acciones

Primero, vamos a definir algunas acciones.

Las **acciones** son un bloque de información que envia datos desde tu aplicación a tu store. Son la *única* fuente de información para el store. Las envias al store usando `store.dispatch()`.

Aquí hay unas acciones de ejemplo que representan agergar nuevas tareas pendientes:

```
const ADD_TODO = 'ADD_TODO'
```

```
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

Las acciones son objetos planos de JavaScript. Una acción debe tener una propiedad `type` que indica el tipo de acción a realizar. Los tipos normalmente son definidos como strings constantes. Una vez que tu aplicación sea suficientemente grande, quizas quieras moverlos a un módulo separado.

```
import { ADD_TODO, REMOVE_TODO } from '../actionTypes'
```

> **Nota**
>
> No necesitas definir tus tipos de acciones constantes en un archivo separado, o incluso definirlas. Para proyectos pequeños, probablemente sea más fácil usar strings directamente para los tipos de acciones. De todas formas, hay algunos beneficios explícitos en declarar constantes en grandes bloques de código. Lee Reduciendo el Boilerplate para más consejos prácticos sobre como mantener tu código limpio.

Además del `type`, el resto de la estructura de los objetos de acciones depende de tí. Si estas interesado, revisaFlux Standard Action para recomendaciones de como una acción debe armarse.

Vamos a agregar una acción más para describir un usuario marcando una tarea como completa. Nos referimos a una tarea en particular como su `index` ya que vamos a almacenarlos en un array. En una aplicación real, es más inteligente generar un ID único cada vez que creamos una nueva.

```
{
  type: COMPLETE_TODO,
  index: 5
}
```

Es una buena idea pasar la menor cantidad de información posible. Por ejemplo, es mejor pasar el `index` que todo el objeto de tarea.

Por último, vamos a agregar una acción más para cambiar las tareas actualmente visibles.

```
{
  type: SET_VISIBILITY_FILTER,
  filter: SHOW_COMPLETED
}
```

Creadores de acciones

Los **creadores de acciones** son exactamente eso—funciones que crean acciones. Es fácil combinar los términos "acción" con "creador de acción", así que haz lo mejor por usar los términos correctos.

En implementaciones de Flux tradicional, los creadores de acciones ejecutan el despacho cuando son invocadas, algo así:

```
function addTodoWithDispatch(text) {
  const action = {
    type: ADD_TODO,
    text
  }
  dispatch(action)
}
```

En cambio, en Redux los creadores de acciones simplemente regresan una acción:

```
function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}
```

Esto las hace más portables y fáciles de probar. Para efectivamente inicial un despacho, pasa el resultado a la función `dispatch()`:

```
dispatch(addTodo(text))
dispatch(completeTodo(index))
```

Alternativamente, puedes crear un **creador de acciones conectados** que despache automaticamente:

```
const boundAddTodo = (text) => dispatch(addTodo(text))
const boundCompleteTodo = (index) => dispatch(completeTodo(index))
```

Ahora puedes llamarlas directamente:

```
boundAddTodo(text)
boundCompleteTodo(index)
```

La función `dispatch()` puede ser accedida directamente desde el store como `store.dispatch()`, pero comunmente vas a querer usar utilidades como `connect()` de react-redux. Puedes usar `bindActionCreators()` para automaticamente conectar muchos creadores de acciones a `dispatch()`.

Los creadores de acciones pueden además ser asíncronos y tener efectos secundarios. Puedes leer más sobre las acciones asíncronas en el tutorial avanzado para aprender como manejar respuestas AJAX y combinar creadores de acciones en un flujo de control asíncrono. No salta ahora mismo hasta las acciones asíncronas hasta que completes el tutorial básico, ya que cubre otros conceptos importantes que son prerequisitos para el tutorial avanzado y las acciones asíncronas.

Código fuente

`actions.js`

```
/*
 * tipos de acciones
 */

export const ADD_TODO = 'ADD_TODO'
export const COMPLETE_TODO = 'COMPLETE_TODO'
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'

/*
 * otras constantes
 */

export const VisibilityFilters = {
```

```
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}

/*
 * creadores de acciones
 */

export function addTodo(text) {
  return { type: ADD_TODO, text }
}

export function completeTodo(index) {
  return { type: COMPLETE_TODO, index }
}

export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter }
}
```

Siguientes pasos

¡Ahora vamos a definir algunos reducers para especificar como el estado se actualiza cuando despachas estas acciones!

Reducers

Las acciones describen que *algo pasó*, pero no especifican cómo cambió el estado de la aplicación en respuesta. Esto es trabajo de los reducers.

Diseñando la forma del estado

En Redux, todo el estado de la aplicación es almacenado en un único objeto. Es una buena idea pensar en su forma antes de escribir código. ¿Cuál es la mínima representación del estado de la aplicación como un objeto?

Para nuestra aplicación de tareas, vamos a querer guardar dos cosas diferentes:

- El filtro de visibilidad actualmente seleccionado;
- La lista actual de tareas.

Algunas veces verás que necesitas almacenar algunos datos, así como el estado de la UI, en el árbol de estado. Esto está bien, pero trata de mantener los datos separados del estado de la UI.

```
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
```

> **Nota sobre relaciones**
>
> En aplicaciones más complejas, vas a necesitar tener diferentes entidades que se referencien una a otra. Sugerimos mantener el estado tan normalizado como sea posible, sin nada de anidación. Mantener cada entidad en un objeto con el ID como llave, y usa los IDs para referenciar otras entidades, o para listas. Piensa en el estado de la aplicación como una base de datos. Este enfoque se describe en la documentación de normalizr más detalladamente. Por ejemplo, manteniendo `todosById: { id -> todo }` y `todos: array<id>` dentro del estado es mejor para una

> aplicación real, pero lo vamos a matener simple para el ejemplo.

Manejando Acciones

Ahora que decidimos cómo se verá nuestro objeto de estado, estamos listos para escribir nuestro reducer. El reducer es una función pura que toma el estado anterior y una acción, y devuelve en nuevo estado.

```
(previousState, action) => newState
```

Se llama reducer porque es el tipo de función que pasarías a `Array.prototype.reduce(reducer, ?initialValue)`. Es muy importante que los reducer se mantengan puros. Cosas que **nunca** deberías hacer dentron de un reducer:

- Modificar sus argumentos;
- Realizar tareas con efectos secundarios como llamas a un API o transiciones de rutas.
- Llamar una función no pura, por ejemplo `Date.now()` o `Math.random()`.

En la guía avanzada vamos a ver como realizar efectos secundarios. Por ahora, solo recuerda que los reducers deben ser puros. **Dados los mismos argumentos, debería calcular y devolver el siguiente estado. Sin sorpresas. Sin efectos secundarios. Sin llamadas a APIs. Sin mutaciones. Solo cálculos.**

Con esto dicho, vamos a empezar a escribir nuestro reducer gradualmente enseñandole como entender las acciones que definimos antes.

Vamos a empezar por especificar el estado inicial. Redux va a llamar a nuestros reducers con `undefined` como valor del estado la primera vez. Esta es nuestra oportunidad de devolver el estado inicial de nuestra applicación.

```
import { VisibilityFilters } from './actions'

const initialState = {
  visibilityFilter: VisibilityFilters.SHOW_ALL,
  todos: []
}

function todoApp(state, action) {
  if (typeof state === 'undefined') {
    return initialState
  }

  // Por ahora, no maneja ninguna acción
  // y solo devuelve el estado que recibimos.
  return state
}
```

Un estupendo truco es usar la sintáxis de parámetros por defecto de ES6 para hacer lo anterior de forma más compacta:

```
function todoApp(state = initialState, action) {
  // Por ahora, no maneja ninguna acción
  // y solo devuelve el estado que recibimos.
  return state
}
```

Ahora vamos a manejar `SET_VISIBILITY_FILTER`. Todo lo que necesitamos hacer es cambiar la propiedad `visibilityFilter` en el estado. Fácil:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Nota que:

1. **No modificamos el `state`.** Creamos una copia con `Object.assign()`. `Object.assign(state, { visibilityFilter: action.filter })` también estaría mal: esto modificaría el primero argumento. **Debes** mandar un objeto vacío como primer parámetro. También puedes activar la propuesta del operador spread para escribir `{ ...state, ...newState }`.

2. **Devolvemos el anterior `state` en el caso `default`.** Es importarte devolver el anterior `state` por cualquier acción desconocida.

> **Nota sobre `Object.assign`**
>
> `Object.assign()` es parte de ES6, pero no esta implementado en la mayoría de los navegadores todavía. Vas a necesitar usar ya sea un polyfill, el plugin de Babel, o alguna otra función como `_.assign()`.
>
> **Nota sobre `switch` y Boilerplate**
>
> La sentencia `switch` no es verdadero boilerplate. El verdadero boilerplate de Flux es conceptual: la necesidad de emitir una actualización, la necesidad de registrar el Store con el Dispatcher, la necesidad de que el Store sea un objeto (y las complicaciones

> que existen para hacer aplicaciones universales). Redux resuelve estos problemas usando reducers puros en vez de emisores de eventos.
>
> Desafortunadamente muchos todavía eligen un framework basados en si usan `switch` en su documentación. Si no te gusta `switch`, puedes usar alguna función `createReducer` personalizada que acepte un mapa, como se ve en "Reduciendo el Boilerplate".

Manejando más acciones

¡Todavía tenemos dos acciones más que manejar! Vamos a extender nuestro reducer para manejar `ADD_TODO`.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    default:
      return state
  }
}
```

Tal cual como antes, nunca modificamos directamente `state` o sus propiedades, en cambio devolvemos un nuevo objeto. El nuevo `todos` es igual al viejo `todos` agregándole un único objeto nuevo al final. La tarea más nueva es creada usando los datos de la acción.

Finalmente, la implementación de `COMPLETE_TODO` no va a venir con ninguna una sorpresa:

```
case COMPLETE_TODO:
  return Object.assign({}, state, {
    todos: state.todos.map((todo, index) => {
      if (index === action.index) {
        return Object.assign({}, todo, {
          completed: true
        })
      }
      return todo
```

```
    })
  })
```

Debido a que queremos actualizar un objeto específico del array sin recurrir a modificaciones, necesitamos crear un nuevo array con los mismo objetos menos el objeto en la posición. Si te encuentras realizando mucho estas operaciones, es una buena idea usar utilidades como react-addons-update, updeep, o incluso una librería como Immutable que tienen soporte nativo a actualizaciones profundas. Solo recuerda nunca asignar nada a algo dentro de `state` antes de clonarlo primero.

Separando Reducers

Este es nuestro código hasta ahora. Es algo Here is our code so far. It is rather verboso:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    case COMPLETE_TODO:
      return Object.assign({}, state, {
        todos: state.todos.map((todo, index) => {
          if(index === action.index) {
            return Object.assign({}, todo, {
              completed: true
            })
          }
          return todo
        })
      })
    default:
      return state
  }
}
```

¿Hay alguna forma de hacerlo más fácil de entender? Parece que `todos` y `visibilityFilter` se actualizan de forma completamente separadas. Algunas veces campos del estado dependen uno de otro y hay que tener en cuenta más cosas, pero en

nuestro caso podemos facilmente actualizar `todos` en una función separada:

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case COMPLETE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    default:
      return state
  }
}

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
    case COMPLETE_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      })
    default:
      return state
  }
}
```

Fijate que `todos` acepta `state` —¡Pero es un array! Ahora `todoApp` solo le manda una parte del estado para que la maneje, y `todos` sabe como actualizar esa parte. **Esto es llamado *composición de reducers*, y es un patrón fundamental al construir aplicaciones de Redux.**

Vamos a explorar la composición de reducers un poco más. ¿Podemos extraer a otro reducer el control de `visibilityFilter` ? Podemos:

```
function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
  case SET_VISIBILITY_FILTER:
```

```
    return action.filter
  default:
    return state
  }
}
```

Ahora podemos reescribir el reducer principal como una función que llama a los reducers que controlan distintas partes del estado, y los combina en un solo objeto. Ni siquiera necesita saber el estado inicial. Es suficiente con que los reducers hijos devuelvan su estado inicial cuando reciben `undefined` la primera vez.

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case COMPLETE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    default:
      return state
  }
}

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

Nota que cada uno de estos reducers esta manejando su propia parte del estado global. El parámetro `state` es diferente por cada reducer, y corresponde con la parte

del estado que controla.

¡Esto ya se está viendo mejor! Cuando una aplicación es muy grande, podemos dividir nuestros reducers en archivos separados y mantenerlos completamente independientes y controlando datos específicos.

Por último, Redux viene con una utilidad llamada `combineReducers()` que realiza la misma lógica que usa `todoApp` arriba. Con su ayuda, podemos reescribir `todoApp` de esta forma.

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Fíjate que esto es exactamente lo mismo que:

```
export default function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

Además puedes darles diferentes nombres, o llamar funciones diferentes. Estas dos formas de combinar reducers son exactamente lo mismo:

```
const reducer = combineReducers({
  a: doSomethingWithA,
  b: processB,
  c: c
})
```

```
function reducer(state = {}, action) {
  return {
    a: doSomethingWithA(state.a, action),
    b: processB(state.b, action),
    c: c(state.c, action)
  }
}
```

Todo lo que `combineReducers()` hace es generar una función que llama a tus reducers **con la parte del estado seleccionada de acuerdo a su propiedad**, y combinar sus resultados en un único objeto. It's not magic..

> **Nota para expertos de ES6**
>
> Ya que `combineReducers` espera un objeto, podemos poner todos nuestros reducers en un archivo separado, `export` cada función reductora, y usar `import * as reducers` para obtenerlos todos juntos como objetos con sus nombres como propiedades.
>
> ```
> import { combineReducers } from 'redux'
> import * as reducers from './reducers'
>
> const todoApp = combineReducers(reducers)
> ```
>
> Ya que `import *` es todavía una sintaxis nueva, no la usamos más en la documentación para evitar confusiones, pero probalemente te encuentro con algunos ejemplos en la comunidad.

Código fuente

`reducers.js`

```
import { combineReducers } from 'redux'
import { ADD_TODO, COMPLETE_TODO, SET_VISIBILITY_FILTER, VisibilityFilters } from './acti
const { SHOW_ALL } = VisibilityFilters

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case COMPLETE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
```

```
      })
    default:
      return state
  }
}

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Siguientes pasos

A continuación, vamos a ver como crear un store de Redux que contenga todo el estado y se encargue de llamar a nuestro reducer cuando se despache una acción.

Avanzado

En la guía básica, exploramos cómo estructurar una aplicación de Redux simple. En esta guía, vamos a ver cómo encajan AJAX y ruteo.

Flujo asíncrono

Sin middlewares, Redux sólo soporta flujos de datos síncronos. Es lo que obtienes por defecto con `createStore()`.

Seguramente mejoraste `createStore()` con `applyMiddleware()`. No es necesario, pero te permite usar acciones asíncrona de forma más fácil.

Middlewares asíncronos como redux-thunk o redux-promise envuelven el método `dispatch()` y te permiten despachar otras cosas además de acciones, por ejemplo funciones o promesas. Cualquier middleware que uses entonces interpretan cualquier cosa que despaches, y entonces, pueden pasar acciones al siguiente middleware de la cadena. Por ejemplo, un middleware de promesas puede interceptar cualquier promesa y despachar acciones asíncronas en respuesta a estas.

Cuando el último middleware de la cadena despache una acción, tiene que ser un objeto plano. Acá es cómo el flujo de datos síncrono de Redux toma lugar.

Revisa el código fuente del ejemplo asíncrono.

Siguientes pasos

Ahora que viste un ejemplo de que puede hacer un middleware en Redux, es hora de aprender cómo funcionan, y cómo puedes crear uno propio. Ve a la siguiente sección detallada sobre Middlewares.

Recetas

Estos son algunos casos de uso y snippets de código para empezar a usar en aplicaciones de verdad con Redux. Se asume que leíste los temas básicos y avanzados.

- Migrando a Redux
- Reduciendo el Boilerplate
- Render en el Servidor
- Escribiendo pruebas
- Calculando Datos Obtenidos
- Implementando Deshacer

Migrando a Redux

Redux no es un framework monolítico, sino un conjunto de contratos y algunas funciones que hacen que todo funcione en conjunto. La mayor parte de tu "código de Redux" ni siquiera va a hacer uso de la API de Redux, ya que la mayor parte del tiempo vas a crear funciones.

Esto hace fácil migrar a o desde Redux.

¡No queremos encerrarte!

Desde Flux

Los reducers capturan "la esencia" de los Stores de Flux, así que es posible migrar gradualmente de un proyecto Flux existente a uno de Redux, ya sea que uses Flummox, Alt, Flux tradicional o cualquier otra librería de Flux.

También es posible hacer lo contrario y migrar de Redux a cualquier de estas siguiendo los siguiente pasos:

Tu proceso debería ser algo como esto:

- Crea una función llamada `createFluxStore(reducer)` que cree un store de Flux compatible con tu aplicación actual a partir de un reducer. Internamente debería ser similar a la implementación de `createStore` (código fuente) de Redux. Su función dispatch solo debería llamar al `reducer` por cada acción, guardar el siguiente estado y emitir el cambio.
- Esto te permite gradualmente reescribir cada Store de Flux de tu aplicación como un reducer, pero todavía exportar `createFluxStore(reducer)` así el resto de tu aplicación no se entera de que esto está ocurriendo con los Stores de Flux.
- Mientras reescribes tus Stores, vas a darte cuenta que deberías evitar algunos anti-patrones de Flux como peticiones a APIs dentro del Store, o ejecutar acciones desde el Store. Tu código de Flux va a ser más fácil de entender cuando lo modifiques para que funcione en base a reducers.
- Cuando hayas portado todos los Stores de Flux para que funcionen con reducers, puedes reemplazar tu librería de Flux con un único Store de Redux, y combinar estos reducers que ya tienes usando `combineReducers(reducers)`.
- Ahora lo único que falta es portar la UI para que use react-redux o alguno similar.
- Finalmente, seguramente quieras usar cosas como middlewares para simplificar tu

código asíncrono.

Desde Backbone

La capa de modelos de Backbone es bastante diferente de Redux, así que no recomendamos combinarlos. Si es posible, lo mejor es reescribir la capa de modelos de tu aplicación desde cero en vez de conectar Backbone a Redux. Igualmente, si no es posible reescribirlo, capaz debería usar backbone-redux para migrar gradualmente, y mantener tu Store de Redux sincronizado con los modelos y colecciones de Backbone.

Solución de problemas

Este un un lugar para compartir soluciones a problemas comunes. Los ejemplos usan React, pero deberías encontrarlos útiles incluso si usas otra cosa.

No ocurre nada el despachar una acción

Algunas veces, tratas de despachar una acción, pero la vista no se actualiza. ¿Por qué ocurre esto? Hay varias razons.

Nunca modifiques los argumentos del reducer

Si estas tratando de modificar el `state` o `action` que te da Redux. ¡No lo hagas!

Redux asume que nunca modificas el estado que te da en los reducers. **Siempre debes devolver un nuevo objeto con el estado**. Incluso si no usas librerías como https://facebook.github.io/immutable-js/, necesitas evitar completamente las modificaciones.

La inmutabilidad es lo que permite a react-redux suscribirse eficientemente a actualizaciones muy específicas de tu estado. También permite un gran experiencia de desarrollo gracias a características como time travel con redux-devtools.

Por ejemplo, un reducer como este está mal porque modifica el estado:

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      // Mal! Esto modifica el estado
      state.push({
        text: action.text,
        completed: false
      })
      return state
    case 'COMPLETE_TODO':
      // Mal! Esto modifica state[action.index]
      state[action.index].completed = true
      return state
    default:
      return state
  }
}
```

Debe ser re programado de esta forma:

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      // Devuelve un nuevo array
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      // Devuelve un nuevo array
      return [
        ...state.slice(0, action.index),
        // Copia el objeto antes de modificarlo
        Object.assign({}, state[action.index], {
          completed: true
        }),
        ...state.slice(action.index + 1)
      ]
    default:
      return state
  }
}
```

Es más código, pero es lo que permite a Redux ser predecible y eficiente. si quieres tener menos código puedes usar herramientas como `React.addons.update` para escribir transformaciones inmutables con una sintaxis pequeña:

```
// Antes:
return [
  ...state.slice(0, action.index),
  Object.assign({}, state[action.index], {
    completed: true,
  }),
  ...state.slice(action.index + 1),
];

// Después:
return update(state, {
  [action.index]: {
    completed: {
      $set: true,
    },
  },
});
```

Por último, para actualizar objeto, vas a necesitar algo como `_.extend` de Lodash o mejor, un polyfill de `Object.assign.

Asegurate de que usas `Object.assign` correctamente. Por ejemplo, en vez de devolver algo

como `Object.assign(state, newData)` en tus reducers, devuelve `Object.assign({}, state, newData)`. De estas forma no vas a modificar el `state` anterior.

También puedes habilitar ES7 object spread proposal con Babel stage 1:

```
// Antes:
return [
  ...state.slice(0, action.index),
  Object.assign({}, state[action.index], {
    completed: true,
  }),
  ...state.slice(action.index + 1),
];

// Después:
return [
  ...state.slice(0, action.index),
  { ...state[action.index], completed: true },
  ...state.slice(action.index + 1),
];
```

Ten en cuenta que las funciones experimentales estás sujetas a cambios, y no es bueno depender de ellas en grandes cantidades de código.

No olvides llamar `dispatch(action)`

Sí defines un creador de acciones, ejecutándolo *no* va a despachar tus acciones automáticamente. Por ejemplo, este código no hace nada:

`TodoActions.js`

```
export function addTodo(text) {
  return { type: 'ADD_TODO', text };
};
```

`AddTodo.js`

```
import React, { Component } from 'react';
import { addTodo } from './TodoActions';

class AddTodo extends Component {
  handleClick() {
    // ¡No funciona!
    addTodo('Fix the issue');
  }

  render() {
    return (
      <button onClick={() => this.handleClick()}>
```

```
          Add
        </button>
      );
    }
  }
```

No funciona ya que el creador de acciones es solo una función que *devuelve* una acción. Pero depende de vos despacharla.No podemos conectar tu creador de acciones a una instancia específica del Store durante su definición ya que las aplicaciones que renderizas en el servidor necesitas un store de Redux para cada petición.

Esto se arreglar ejecutando el método `dispatch()` de la instancia del store:

```
handleClick() {
  // ¡Funciona! (pero debes obtener el store de alguna forma)
  store.dispatch(addTodo('Fix the issue'));
}
```

Si estas en algún punto profundo en la jerarquía de componentes, es incómodo pasar el store manualmente. Es por eso que react-redux te permite usar el componente de nivel superior `connect` que va a además de suscribirse al store de Redux, inyectar `dispatch` en los props de tus componentes.

El código arreglado quedaría así:

`AddTodo.js`

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { addTodo } from './TodoActions';

class AddTodo extends Component {
  handleClick() {
    // ¡Funciona!
    this.props.dispatch(addTodo('Fix the issue'));
  }

  render() {
    return (
      <button onClick={() => this.handleClick()}>
        Add
      </button>
    )
  }
}

// Además del state, `connect` agregar `dispatch` en nuestros props.
export default connect()(AddTodo);
```

Puedes pasar `dispatch` a otros componentes manualmente, si quieres.

Algo más no funciona

Pregunta en el canal #redux con Reactiflux, o crea un issue. Si lo descubre, modifica este documento como cortesía a la siguiente persona con el mismo problema.

Glosario

Este es un glosario de los términos principales en Redux, junto a su tipo de dato. Los tipos están documentados usando la notación Flow.

Estado

```
type State = any
```

Estado (también llamado árbol de estado) es un termino general, pero en la API de Redux normalmente se refiere al valor de estado único que es manejado por el Store y devuelto por `getState()`. Representa el estado de tu aplicación de Redux, normalmente es un objeto con muchas anidaciones.

Por convención, el estado a nivel superior es un objeto o algún tipo de colección llave-valor como un Map, pero técnicamente puede ser de cualquier tipo. Aun así, debes hacer tu mejor esfuerzo en mantener el estado serializable. No pongas nada dentro que no puedas fácilmente convertirlo a un JSON.

Acción

```
type Action = object
```

Una acción es un objeto plano (POJO - Plan Old JavaScript Object) que representa una intención de modificar el estado. Las acciones son la única forma en que los datos llegan al store. Cualquier dato, ya sean eventos de UI, callbacks de red, u otros recursos como WebSockets eventualmente van a ser despachados como acciones.

Las acciones deben tener un campo `type` que indica el tipo de acción a realizar. Los tipos pueden ser definidos como constantes e importados desde otro módulo Es mejor usar strings como tipos en vez de Symbols ya que los strings son serializables.

Aparte del `type`, la estructura de una acción depende de vos. Si estás interesado, revisa Flux Standard Action para recomendaciones de cómo deberías estar estructurado una acción.

Revisa acción asíncrona debajo.

Reducer

```
type Reducer<S, A> = (state: S, action: A) => S
```

Un *reducer* (también llamado *función reductora*) es una función que acepta una acumulación y una valor y devuelve una nueva acumulación. Son usados para reducir una colección de valores a un único valor.

Los reducers no son únicos de Redux — son un concepto principal de la programación funcional. Incluso muchos lenguajes no funcionales, como JavaScript, tienen una API para reducción. En JavaScript, es `Array.prototype.reduce()`.

En Redux, el valor acumulado es el árbol de estado, y los valores que están siendo acumulados son acciones. Los reducers calculan el nuevo estado con base al anterior estado y la acción. Deben ser *funciones puras* — funciones que devuelven el mismo valor dados los mismos argumentos. Deben estar libres de efectos secundarios. Esto es lo que permite características increíbles como hot reloading y time travel.

Los reducers son el concepto más importante en Redux.

No hagas peticiones a APIs en los reducers.

Función despachadora

```
type BaseDispatch = (a: Action) => Action
type Dispatch = (a: Action | AsyncAction) => any
```

La *función despachadora* (o simplemente *función dispatch*) es una función que acepta una acción o una acción asíncrona; entonces puede o no despachar una o más acciones al store.

Debemos distinguir entre una función despachadora en general y la función base `dispatch` provista por la instancia del store sin ningún middleware.

La función base dispatch *siempre* envía síncronamente acciones al reducer del store, junto al estado anterior devuelto por el store, para calcular el nuevo estado. Espera que las acciones sean objetos planos listos para ser consumidos por el reducer.

Los middlewares envuelven la función dispatch base. Le permiten a la función dispatch manejar acciones asíncronas además de las acciones. Un middleware puede transformar, retrasar, ignorar o interpretar de cualquier forma una acción o acción asíncrona antes de

pasarla al siguiente middleware. Lea más abajo para más información.

Creador de acciones

```
type ActionCreator = (...args: any) => Action | AsyncAction
```

Un *creador de acciones* es, simplemente, una función que devuelve una acción. No confunda los dos términos — otra vez, una acción es un pedazo de información, y los creadores de acciones son fabricas que crean esas acciones.

Llamar un creador de acciones solo produce una acción, no la despacha. Necesitas llamar al método `dispatch` del store para causar una modificación. Algunas veces decimos *creador de acciones conectado*, esto es una función que ejecuta un creador de acciones e inmediatamente despacha el resultado a una instancia del store específica.

Si un creador de acciones necesita leer el estado actual, hacer una llamada al API, o causar un efecto secundario, como una transición de rutas, debe retornas una acción asíncrona en vez de una acción.

Acción asíncrona

```
type AsyncAction = any
```

Una *acción asíncrona* es un valor que es enviado a una función despachadora, pero todavía no está listo para ser consumido por el reducer. Debe ser transformada por un middleware en una acción (o una serie de acciones) antes de ser enviada a la función `dispatch()` base. Las acciones asíncronas pueden ser de diferentes tipos, dependiendo del middleware que uses. Normalmente son primitivos asíncronos como una promesa o un thunk, que no son enviados inmediatamente a un reducer, pero despachan una acción cuando una operación se completa.

Middleware

```
type MiddlewareAPI = { dispatch: Dispatch, getState: () => State }
type Middleware = (api: MiddlewareAPI) => (next: Dispatch) => Dispatch
```

Un middleware es una función de orden superior que toma una función despachadora y devuelve una nueva función despachadora. A menudo convierten acciones asíncronas en

acciones.

Los middlewares son combinables usando funciones. Son útiles para registrar acciones, realizar efectos secundarios como ruteo, o convertir una llamada asíncrona a una API en una serie de acciones síncronas.

Revisa `applyMiddleware(...middlewares)` para más detalles de los middlewares.

Store

```
type Store = {
  dispatch: Dispatch
  getState: () => State
  subscribe: (listener: () => void) => () => void
  replaceReducer: (reducer: Reducer) => void
}
```

Un store es un objeto que mantiene el árbol de estado de la aplicación.

Solo debe haber un único store en una aplicación de Redux, ya que la composición ocurre en los reducers.

- dispatch(action) es la función dispatch base descrita arriba.
- getState() devuelve el estado actual de la aplicación.
- subscribe(listener) registra una función para que se ejecute en cada cambio de estado.
- replaceReducer(nextReducer) puede ser usada para implementar hot reloading y code splitting. Normalmente no la vas a usar.

Revisa la referencia API del Store completa para más detalles.

Creador de store

```
type StoreCreator = (reducer: Reducer, initialState: ?State) => Store
```

Un creador de store es una función que crea un store de Redux. Al igual que la función despachante, debemos separar un creador de stores base, `createStore(reducer, initialState)` exportado por Redux, por los creadores de store devueltos por los potenciadores de store.

Potenciador de store

```
type StoreEnhancer = (next: StoreCreator) => StoreCreator
```

Un potenciador de store es una función de orden superior que toma un creador de store y devuelve una versión potenciada del creador de store. Es similar a un middleware ya que te permite alterar la interfaz de un store de manera combinable.

Los potenciadores de store son casi el mismo concepto que los componentes de orden superior de React, que ocasionalmente se los llamada "potenciadores de componentes".

Debido a que el store no es una instancia, sino una colección de funciones en un objeto plano, es posible crear copias fácilmente y modificarlas sin modificar el store original. Hay un ejemplo en la documentación de `compose` demostrándolo.

Normalmente nunca vas a escribir un potenciador de store, pero capaz uses el que provee las herramientas de desarrollo. Es lo que permite que el time travel sea posible sin que la aplicación se entere de que está ocurriendo. Curiosamente, la forma de implementar middleware en Redux es un potenciador de store.

Referencia API

La API de Redux es diminuta. Redux define una serie de contratos que debes implementar (como los reducers y provee una pocas funciones de ayuda para unir todo.

Esta sección documenta completamente la API de Redux. Recuerda que Redux solo se preocupa con mantener el estado de tu aplicación. En una aplicación de verdad, seguramente quieras usar conexiones como react-redux.

Exportaciones de nivel superior

- createStore(reducer, [initialState])
- combineReducers(reducers)
- applyMiddleware(...middlewares)
- bindActionCreators(actionCreators, dispatch)
- compose(...functions)

API del Store

- Store
  - getState()
  - dispatch(action)
  - subscribe(listener)
  - replaceReducer(nextReducer)

Importando

Cada función descrita arriba ex una exportación de nivel superior. Puedes importar cualquiera de estas de esta forma:

ES6

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

ES5 (CommonJS)

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

ES5 (UMD build)

```
var createStore = Redux.createStore;
```

`createStore(reducer, [initialState], [enhancer])`

Crea un store de Redux que mantiene el árbol de estado de tu aplicación. Solo debe haber un único store en tu aplicación.

Argumentos

1. `reducer` *(Función)*: Una función reductora que devuelve el siguiente árbol de estado, dado el árbol de estado actual y el una acción.

2. [`initialState`] *(cualquier cosa)*: El estado inicial. Puedes opcionalmente especificarlo para hidratar la aplicación con el estado del servidor en aplicaciones universales, o restaurar una sesión anterior serializada. Si crear el `reducer` con `combineReducers`, este debe ser un objeto plano con la misma forma usada ahí. De otra forma, eres libre de pasar cualquier cosa que el `reducer` pueda entender.

3. [`enhancer`] *(Función)*: El potenciador del store. Puedes opcionalmente especificarlo para mejorar el store con funcionalidad de terceros como los middlewares, time travel, persistencia, etc. El único potenciador de store que viene con Redux es `applyMiddleware()`.

Regresa

(*`Store`*): Un objeto que contiene el estado actual de la aplicación. La única forma de cambiar su estado es despachando acciones. Además deberías suscribirte a los cambios de estado para actualizar la UI.

Ejemplo

```
import { createStore } from 'redux'

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([ action.text ])
    default:
      return state
  }
}

let store = createStore(todos, [ 'Use Redux' ])
```

```
store.dispatch({
  type: 'ADD_TODO',
  text: 'Read the docs'
})

console.log(store.getState())
// [ 'Use Redux', 'Read the docs' ]
```

Consejos

- ¡No crees más de un Store en la aplicación! En cambio, usa `combineReducers` un solo reducer principal a partir de varios..
- Es tu decisión la forma del estado. Puedes usar objetos planos o algo como Immutable. Si no estas seguro, usa objetos planos.
- Si tu estado es un objeto plano, ¡Asegúrate de nunca modificarlo! Por ejemplo, en vez de regresar algo como `Object.assign(state, newData)` en tus reducers, regresa `Object.assign({}, state, newData)`. De esta forma no vas a sobreescribir el `state` anterior. Además puedes hacer `return { ...state, ...newData }` si habilitas la propuesta del operador spread.
- En aplicaciones universales que corren en el servidor, crea una instancia del store en cada petición de forma que estén aisladas. Despacha unas pocas acciones para obtener datos a la instancia del store y espera que se completen antes de renderizar la aplicación en el servidor.
- Cuando un store es creado, Redux despacha una acción falsa para que tu reducer llene el store con su estado inicial. No se supone que manejes esa acción directamente. solo recuerda que tu reducer debe devolver alguna clase de estado inicial si el estado provisto como primer argumento es `undefined`, y ya esta listo.
- Para aplicar multiples potenciadores de stores, probablemente quieras usar `compose()`.

Store

El store contiene todo el árbol de estado de tu aplicación. La única forma de cambiar el estado que contiene es despachando una acción.

El store no es una clase. Es solo un objeto con unos pócos métodos. Para crearlo, pasa tu principal función reductora a `createStore`.

> **Una nota para usuarios de Flux**
>
> Si vienes de Flux, hay una importante diferencias que necesitas entender. Redux no tiene un Dispatcher o soporta múltiples stores. **En cambio, hay un único store con una única función reductora.** Mientras tu aplicación crece, en vez de añadir stores, puedes dividir tu reducer en varios reducer independientes que operan en diferentes partes del árbolde estado. Puedes usar función como `combineReducers` para combinarlo. Esto es similar a como hay un único componente raíz en aplicaciones de React, pero esta compuesto de muchos componentes pequeños.

Métodos del Store(#metodos-store)

- `getState()`
- `dispatch(action)`
- `subscribe(listener)`
- `replaceReducer(nextReducer)`

Métodos del Store

`getState()`

Regresa el actual árbol de estado de tu aplicación. Es igual al último valor regresado por los reducers del store.

Regresa

(any): El actual árbol de estado de tu aplicación.

`dispatch(action)`

Despacha una acción. Esta es la única forma de realizar un cambio de estado.

La función función reductora es ejecutada por el resultado de `getState()` y el `action` indicado de forma síncrona. El valor devuelto es considerado el siguiente estado. Va a ser devuelto por `getState()` desde ahora, y las funciones escuchando los cambios van a ser inmediatamente notificadas.

> **Una nota para usuarios de Flux**
>
> Si intentas llamar a `dispatch` desde dentro de un reducer, va a tirar un error diciendo "Los reducers no deben despachar acciones." Esto es similar al error de Flux de "No se puede despachar en medio de un despacheo", pero no causa los problemas asociados con este. En Flux, despachar esta prohibido mientras los Store están manejando las acciones y emitiendo las actualizaciones. Esto es una lástima porque hace imposible despachar acciones desde el ciclo de vida del componente u otras partes benignas.
>
> En Redux, los suscriptores se ejecutan luego de que el reducer principal devuelve el nuevo estado, así *quizás* quieras despachar dentro de ellos. Solo no puedes despachar dentro de reducers porque no pueden tener efectos secundarios. Si quieres causar un efecto secundario en respuesta a una acción, el mejor lugar para hacer eso es creadores de acciones asíncronas.

Argumentos

1. `action` (*Objeto*†): Un objeto plano describiendo las cambios necesarios para tu aplicación. Las acciones son la única forma de enviar datos al store, así que cualquier dato, ya sea de eventos de la UI, callbacks de peticiones, o cualquier otra fuente como WebSocket va a necesitar eventualmente ser despachada como acciones. Las acciones deben tener un campo `type` que indique el tipo de acción que se esta realizando. Los tipos pueden ser definidos como constantes e importados desde otros módulos. Es mejor usar strings para el `type` en vez de Symbols porque los strings son serializables. Aparte de `type`, la estructura de los objetos de acciones son controlador por tí. Si estás interesado, revisa Flux Standard Action para recomendaciones de como armas acciones.

Regresa

(Objeto†): La acción despachada (revisa las notas).

Notas

† La implementación "vainilla" del store que obtienes al llamar `createStore` solo soporta

objetos planos como acciones y las maneja inmediatamento en los reducers.

De todas formas, si envuelves `createStore` con `applyMiddleware`, el middleware puede interpretar acciones de otra forma, y proveer soporta para despachar acciones asíncronas. Las acciones asíncronas normalmente son Promises, Observables, o thunks.

Los middlewares son creador por la comunidad y no vienen incluidos en Redux por defecto. Necesitas explícitamente instalar paquetes como redux-thunk o redux-promise para usarlos. Probablemente también crees tus propios middlewares.

Para aprender como describir llamadas asíncronas a un API, leer el estado actual dentro de creadores de acciones o realizar efectos secundarios, mira los ejemplos de `applyMiddleware`.

Ejemplo

```
import { createStore } from 'redux'
let store = createStore(todos, [ 'Use Redux' ])

function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}

store.dispatch(addTodo('Read the docs'))
store.dispatch(addTodo('Read about the middleware'))
```

`subscribe(listener)`

Agregar una función que escucha los cambios. Va a ser ejecutada cada vez que una acción es despachada y algunas partes del árbol de estado puedan potencialmente haber cambiado. Probablemente quieras llamar a `getState()` para leer el árbol de estado actual dentro del callback.

Probablemente quieras llamar a `dispatch()` desde el callback, siguiendo las siguientes advertencias:

1. Las suscripciones son ejecutadas justo después de cada llamada de `dispatch()`. Si te suscribes o desuscribes mientras un listener esta siendo invocado, no va a tener ningún efecto en el `dispatch()` actualmente en progreso. De todas formas, la siguiente llamada a `dispatch()`, ya sea anidada o no, va a usar la más reciente versión de la lista de suscripciones.

2. Los listeners no deben esperar ver todos los cambios de estado, ya que el estado puede haber sido actualizado multiples veces durante los `dispatch()` anidados antes de que los listeners sean llamados. De todas formas, es seguro que todos los listeners registrados antes de que `dispatch()` sea ejecutado van a ser llamados con el último estado existente en ese momento.

Es una API de bajo nivel. Comunmente, en vez de usarla directamente, vas a usar conexiones para React (u otros). Si sientes que necesitas que el callback sea invocado con el estado actual, probablemente quieras convertir el store en un Obersavle o escribir un `observeStore` personalizado

Para desuscribir un listener invoca la función devuelta por `subscribe`.

Arguments

1. `listener` (*Función*): El callback que va a ser invocado cada vez que una acción es despachada y el árbol de estado puede haber cambiado. Probablemente quieras usar `getState()` dentro de este callback para obtener el árbol de estado actual.

Regresa

(*Funciǿn*): Una función para desuscribir el listener.

Ejemplo

```
function select(state) {
  return state.some.deep.property
}

let currentValue
function handleChange() {
  let previousValue = currentValue
  currentValue = select(store.getState())

  if (previousValue !== currentValue) {
    console.log('Some deep nested property changed from', previousValue, 'to', currentVal
  }
}

let unsubscribe = store.subscribe(handleChange)
handleChange()
```

`replaceReducer(nextReducer)`

Reemplaza el reducer actualmente usado en el store para calcular el estado.

Es una API avanzada. Probablemente lo necesitas si tu aplicación implementa separación de código y quieres cargas algunos reducers dinamicamente. Además necesitas esto si implementas mecanismos como hot-reloading en Redux.

Argumentos

1. `reducer` (*Función*) El nuevo reducer a ser usado en el store.

`combineReducers(reducers)`

Mientras tu aplicación se vuelve más compleja, vas a querer separar tus funciones reductoras en funciones separadas, cada una manejando partes independientes del estado.

La función `combineReducers` devuelve un objeto cuyos valores son diferentes funciones reductoras en una única función reductora que puedes enviar a `createStore`.

El reducer resultante llama cada reducer interno, y junta sus resultados en un único objeto de estado. **La forma del objeto de estado es igual a las llaves enviadas a `reducers`.**

Consecuentemente, el objeto de estado luciría así:

```
{
  reducer1: ...
  reducer2: ...
}
```

Puedes controlar los nombres de llaves usando diferentes llaves para los reducers pasados a los objetos. Por ejemplo, podrías usar `combineReducers({ todos: myTodosReducer, counter: myCounterReducer })` para crear un estado con la forma `{ todos, counter }`.

Una convención popular es nombrar los reducers con el pedazo de estado que controlan, así puedes usar forma abreviada de definir propiedades: `combineReducers({ counter, todos })`. Esto es lo mismo que hacer `combineReducers({ counter: counter, todos: todos })`.

> **Una nota para usuarios de Flux**
>
> Esta función te ayuda a organizar tus reducers para que manejen su propia parte del estado, similar a si tuvieras diferentes Stores de Flux para manejar diferentes estados. Con Redux, hay un solo store, pero `combineReducers` te ayuda a mantener la misma lógica de separación entre reducers.

Argumentos

1. `reducers` (*Objeto*): Un objeto cuyos valores corresponden a diferentes funciones reductoras que necesitas combinar en uno solo. Revisa las notas debajo para ver algunas reglas que cada reducer debe seguir.

> La primers documentación sugería usar la sintaxis de ES `import * as reducers` para obtener el objeto reducer. Esto fue una fuente de confusión, razón por la cual ahora recomendados exportar un único reducer obtenido usando `combineReducers()` en

> `reducers/index.js`. Se incluye un ejemplo debajo.

Regresa

(*Función*): Un reducer que invoca cada reducer dentro del objeto `reducers`, y arma el objeto de estado con la misma forma.

Notas

Esta función es medianamente opinionada y este hecha para evitar algunos error de principiantes. Es por eso que te fuerza a seguir ciertas reglas que no tendrías que seguir si escribieses tu reducer manualmente.

Cualquier reducer enviado a `combineReducers` debe satisfaces las siguientes reglas:

- Si no reconoce una acción, debe regresarel `state` recibido como primer argumento.
- Nunca debe devolver `undefined`. Es muy fácil hacer eso por error con un `return`, por eso `combineReducers` tira un error en vez de dejar que el error se manifieste en otra parte.
- Si el `state` proporcionado es `undefined`, debe devolver el estado inicial para ese reducer específico. De acuerdo con las reglas anteriores, el estado inicial tampoco puede ser `undefined`. Es fácil especificarlo usando la sintaxis de parámetros opcionales de ES^, pero además debes específicamente verificar que el primer argumento no sea `undefined`.

Mientras que `combineReducers` intenta validar tus reducers conforme a algunas de estas reglas, debes recordarlas y hacer lo posible para seguirlas.

Ejemplo

`reducers/todos.js`

```
export default function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([ action.text ])
    default:
      return state
  }
}
```

`reducers/counter.js`

```
export default function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

`reducers/index.js`

```
import { combineReducers } from 'redux'
import todos from './todos'
import counter from './counter'

export default combineReducers({
  todos,
  counter
})
```

`App.js`

```
import { createStore } from 'redux'
import reducer from './reducers/index'

let store = createStore(reducer)
console.log(store.getState())
// {
//   counter: 0,
//   todos: []
// }

store.dispatch({
  type: 'ADD_TODO',
  text: 'Use Redux'
})
console.log(store.getState())
// {
//   counter: 0,
//   todos: [ 'Use Redux' ]
// }
```

Consejos

- ¡Esta función para por conveniencia! Puedes crear tu propio `combineReducers` que funcione diferente, o incluso armar el objeto del estado en base a los reducers hijos manualmente y crear la función reductora principal explicitamente, como harías con

cualquier otra función.

- Puedes llamar `combineReducers` en cualquier punto de la jerarquía de reducers. No tiene que ocurrir al nivel superior. De hecho puedes usarlo para separar los reducers hijos que se volvieron muy complicados en reducers nietos independientes.

`applyMiddleware(...middlewares)`

Los middleware son la forma sugerida de extender Redux con funcionalidades personalizadas. Los middlewares te dajan envolver el método `dispatch` del Store. La característica principal de los middlewares es que son combinables. Multiples middlewares se pueden combinarjunto, donde ninguno necesita saber cual vino antes o viene después.

El uso más común de los middlewares es soportar acciones asíncronas sin demasiado código o dependiendo de librerías como Rx. Logre eso gracias a que te permite despachar acciones asíncronas además de las normales.

Por ejemplo, redux-thunk permite a los creadores de acciones invertír el control despachando funciones. Van a recibir `dispatch` como argumento y capaz llamarlo asíncronamente. Estas funciones son llamadas *thunks*. Otro ejemplo de middleware es redux-promise. Este te deja despachar una Promesa como una acción asíncrona, y despachar una acción normal cuando la Promesa se resuelve.

Los middlewares no vienen incluidos en `createStore` y no son una parte fundamental de la arquitectua de Redux, pero los consideramos suficientemente útiles para soportarlos directamente. De esta forma, hay una única forma estandarizada de extender `dispatch` y diferentes middlewares probablemente compitan en expresividad y utilidad.

Argumentos

- `...middlewares` (*argumentos*): Funciones que se ajustan la *API de middlewares* de Redux. Cada middleware recibe `dispatch` y el `getState` del `Store` como argumentos, y regresa una función. Esa función va a recibir el método para despachar el siguiente middleware, y se espera que devuelva una función que recibe `action` y llame `next(action)`. El último middleware de la cadena va a recibir el verdadero método `dispatch` del store como parámetro `next`, terminando la cadena. Así, la forma de un middleware sería `({ getState, dispatch }) => next => action`.

Regresa

(*Función*) Un potenciador de store que aplican los middlewares. El potenciador de store tiene el siguiente formato `createStore => createStore`, pero es más fácil de aplicar si lo envias a `createStore()` como el último argumento `enhancer`.

Ejemplo: Middleware de Logging Personalizado

```
import { createStore, applyMiddleware } from 'redux'
import todos from './reducers'

function logger({ getState }) {
  return (next) => (action) => {
    console.log('will dispatch', action)

    // Llama al siguiente método dispatch en la cadena de middlewares
    let returnValue = next(action)

    console.log('state after dispatch', getState())

    // Este seguramente sera la acción, excepto
    // que un middleware anterior la haya modificado.
    return returnValue
  }
}

let store = createStore(
  todos,
  [ 'Use Redux' ],
  applyMiddleware(logger)
)

store.dispatch({
  type: 'ADD_TODO',
  text: 'Understand the middleware'
})
// (Esta lineas son registradas por el middleware:)
// will dispatch: { type: 'ADD_TODO', text: 'Understand the middleware' }
// state after dispatch: [ 'Use Redux', 'Understand the middleware' ]
```

Ejemplo: Usando el Middleware Thunk para Acciones Asíncronas

```
import { createStore, combineReducers, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'
import * as reducers from './reducers'

let reducer = combineReducers(reducers)
// applyMiddleware sobrecarga createStore con middlewares:
let store = createStore(reducer, applyMiddleware(thunk))

function fetchSecretSauce() {
  return fetch('https://www.google.com/search?q=secret+sauce')
}

// Estos son los creadores normales que haz visto hasta ahora.
// Las acciones que devuelven pueden ser despachadas sin middlewares.
// De todas formas, ellos expresan "hechos" y no "flujos asíncronos".

function makeASandwich(forPerson, secretSauce) {
  return {
    type: 'MAKE_SANDWICH',
    forPerson,
```

```
    secretSauce
  }
}

function apologize(fromPerson, toPerson, error) {
  return {
    type: 'APOLOGIZE',
    fromPerson,
    toPerson,
    error
  }
}

function withdrawMoney(amount) {
  return {
    type: 'WITHDRAW',
    amount
  }
}

// Incluso sin middlewares, puedes despachar una acción:
store.dispatch(withdrawMoney(100))

// ¿Pero que haces cuando quieres iniciar una acción asíncrona,
// como un llamado a un API, o una transición de rutas?

// Conoce a thunks.
// Un thunk es una función que devuelve una función.
// Esto es un thnk

function makeASandwichWithSecretSauce(forPerson) {

  // ¡Control invertido!
  // Devuelve una función que acepta `dispatch` así podemos despacharlas luego.
  // El middleware thunk sabe como convertír acciones asíncronas con thunks en acciones.

  return function (dispatch) {
    return fetchSecretSauce().then(
      sauce => dispatch(makeASandwich(forPerson, sauce)),
      error => dispatch(apologize('The Sandwich Shop', forPerson, error))
    )
  }
}

// ¡El middleware thunk te deja despachar acciones asíncronas
// como si fuesen acciones!

store.dispatch(
  makeASandwichWithSecretSauce('Me')
)

// Incluso se asegura de regresas el valor de devuelve el thunk
// en el despacho, así puedo anidar promesas.

store.dispatch(
  makeASandwichWithSecretSauce('My wife')
).then(() => {
  console.log('Done!')
})
```

```
// De hecho, podría crear creadores de acciones que despachan
// acciones y acciones asíncronas desde otros creadores de acciones,
// y así crear mi propio flujo de control con promesas.

function makeSandwichesForEverybody() {
  return function (dispatch, getState) {
    if (!getState().sandwiches.isShopOpen) {

      // No necesitas devolver promesas, pero es una convención común
      // así siempre se puede llamar `.then()` como resultado.

      return Promise.resolve()
    }

    // Podemos despachar acciones tanto objetos planos como thunks,
    // lo que nos deja componer las acciones asíncronas en un único flujo.

    return dispatch(
      makeASandwichWithSecretSauce('My Grandma')
    ).then(() =>
      Promise.all([
        dispatch(makeASandwichWithSecretSauce('Me')),
        dispatch(makeASandwichWithSecretSauce('My wife'))
      ])
    ).then(() =>
      dispatch(makeASandwichWithSecretSauce('Our kids'))
    ).then(() =>
      dispatch(getState().myMoney > 42 ?
        withdrawMoney(42) :
        apologize('Me', 'The Sandwich Shop')
      )
    )
  }
}

// Esto es muy útil para renderizado en el servidor, ya que puedo esperar
// hasta que los datos estén disponibles, entonces síncronamente renderizar la app.

import { renderToString } from 'react-dom/server'

store.dispatch(
  makeSandwichesForEverybody()
).then(() =>
  response.send(renderToString(<MyApp store={store} />))
)

// También puedes despachar una acción asínrona desde un componente
// cada vez que los props cambian para cargar los datos faltantes.

import { connect } from 'react-redux'
import { Component } from 'react'

class SandwichShop extends Component {
  componentDidMount() {
    this.props.dispatch(
      makeASandwichWithSecretSauce(this.props.forPerson)
    )
  }
```

```
  componentWillReceiveProps(nextProps) {
    if (nextProps.forPerson !== this.props.forPerson) {
      this.props.dispatch(
        makeASandwichWithSecretSauce(nextProps.forPerson)
      )
    }
  }

  render() {
    return <p>{this.props.sandwiches.join('mustard')}</p>
  }
}

export default connect(
  state => ({
    sandwiches: state.sandwiches
  })
)(SandwichShop)
```

Consejos

- Los middlewares solo envuelvan la función `dispatch`. Técnicamente, cualquier cosa que podrías hacer un middleware, puede hacerse envolviendo manualmente la llamada a `dispatch`, pero es más fácil manejar esto en un solo lugar y definir las transformaciones de acciones a una escala de todo el proyecto.

- Si usas otros potenciadores de store además de `applyMiddleware`, asegurate de poner `applyMiddleware` antes de ellos, porque los middlewares son potenciamente asíncronos. Por ejemplo, si debería ir antes de redux-devtools porque de otra forma las DevTools no van a ver las acciones puras emitidas por un middleware de promesas.

- Si necesitas aplicar un middleware condicionalmente, asegurate de solo importarlo cuando sea necesario:

  ```
  let middleware = [ a, b ]
  if (process.env.NODE_ENV !== 'production') {
    let c = require('some-debug-middleware');
    let d = require('another-debug-middleware');
    middleware = [ ...middleware, c, d ];
  }

  const store = createStore(
    reducer,
    initialState,
    applyMiddleware(...middleware)
  )
  ```

  Esto hace más fácil para herramientas de empaquetado cortar los módulos no

necesarios y reducir el tamaño de los paquetes.

- ¿Algunas vez te preguntantes que es `applyMiddleware`? Debería ser un mecanismo de extensión más poderoso que los mismos middleware. En efecto, `applyMiddleware` es un ejemplo de el mecanismo más poderoso para extender Redux llamado potenciadores de store. Es muy poco probable que alguna vez quieras escribir tu propio potenciador de store. Otro ejemplo de un potenciador de store son las redux-devtools. Los middlewares son menos poderosos que un potenciador de store, pero son más fáciles de escribir.

- Los middlewares suenan mucho más complicados de lo que en realidad son. La única forma de entenderlos de verdad es ver como un middleware existente funciona, y tratar de escribir uno propio. La anidación de funciones parece intimidante, pero la mayoría de los middlewares que puedas encontrar son, de hecho, menos de 10 líneas y la anidación y combinanción es lo que hace al sistema de middlewares poderosos.

- Para aplicar múltiples potenciadores de store, probablemente quieras usar `compose()`.

`bindActionCreators(actionCreators, dispatch)`

Convierte un objeto cuyos valores son creadores de acciones, en un objeto, con las mismas claves, pero donde cada creador de acciones está envuelto en una llamada a `dispatch` para ser invocada inmediatamente.

Normalmente solo deberías llamar a `dispatch` directamente desde tu instancia del `Store`. Si usas Redux con React, react-redux te provee de la función `dispatch` para que las llames directamente.

En único caso de uso para `bindActionCreatores` es cuando quieres pasar un creador de acciones a un componente que no sabe nada de Redux y no quieres pasarle `dispatch` o el store de Redux.

Por conveniencia, también puedes pasar una sola función como primer argumento, y obtener una función devuelta.

Parametros

1. `actionCreators` (*Función* u *Objeto*): Un creador de acciones, o un objeto cuyos valores son creadores de acciones.
2. `dispatch` (*Función*): La función `dispatch` disponible en la instancia del `Store`.

Regresa

(*Función* u *Objeto*): Un objeto similar al original, pero donde cada función despacha inmediatamente la acción regresada por el creador de acciones correspondiente. Si pasas una función como `actionCreators`, el valor devuelto va a ser también una única función.

Ejemplo

`TodoActionCreators.js`

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}
```

```
export function removeTodo(id) {
  return {
    type: 'REMOVE_TODO',
    id
  }
}
```

SomeComponent.js

```
import { Component } from 'react'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

import * as TodoActionCreators from './TodoActionCreators'
console.log(TodoActionCreators)
// {
//   addTodo: Function,
//   removeTodo: Function
// }

class TodoListContainer extends Component {
  componentDidMount() {
    // Inyectado por react-redux:
    let { dispatch } = this.props

    // Nota: esto no funciona
    // TodoActionCreators.addTodo('Use Redux')

    // Solamente estas llamando una función que crea una acción
    // ¡También deberías despachar la acción!

    // Esto funcionaría
    let action = TodoActionCreators.addTodo('Use Redux')
    dispatch(action)
  }

  render() {
    // Inyectado por react-redux:
    let { todos, dispatch } = this.props

    // Acá hay un buen caso de uso para bindActionCreators:
    // Quieres un componente hijo  que este completamente desatendido de Redux

    let boundActionCreators = bindActionCreators(TodoActionCreators, dispatch)
    console.log(boundActionCreators)
    // {
    //   addTodo: Function,
    //   removeTodo: Function
    // }

    return (
      <TodoList todos={todos}
                {...boundActionCreators} />
    )

    // Una alternativa a bindActionCreators es enviar
    // la función dispatch hacia abajo , pero entonces tu componente
```

```
    // hijo necesita importar los creadores de acciones y saber sobre ellos

    // return <TodoList todos={todos} dispatch={dispatch} />
  }
}

export default connect(
  state => ({ todos: state.todos })
)(TodoListContainer)
```

Consejos

- Capaz te preguntes: ¿por qué no unimos los creadores de acciones a la instancia del store directamente, como en Flux? El problema es que eso no funciona en aplicaciones universales que deben renderizar en el servidor. Normalmente vas a querer tener una instancia del store por cada petición así puedes usarlas con diferentes datos, pero conectar los creadores de acciones durante su definición significa que estas atascado con un único instancia por petición.

- Si usas ES5, en vez de usar la sintaxis `import * as` puedes simplemente pasar `require('./TodoActionCreators')` a `bindActionCreators` como primer argumento. Lo único que te tiene que importar es que los valores del argumento `actionCreators` sean funciones. El sistema de módulo no importa.

`compose(...functions)`

Combina funciones de derecha a izquierda.

Esta es una utilidad de programación funcional y es incluida en Redux por conveniencia. Probablemente la quieras usar para aplicar múltiples potenciadores de store en serie.

Argumentos

1. (*argumentos*): Las funciones a combinar. Se espera que cada función acepte un único parámetro. El valor de devuelva va a ser usado como argumento a la función que este a su izquierda, y así. La única excepción es la que esté más a la derecha la cual puede recibir múltiples argumentos.

Regresa

(*Función*): La función final obtenido de combinas las funciones indicadas de derecha a izquierda.

Ejemplo

Este ejemplo demuestra como usar `compose` para potenciar el store con `applyMiddleware` y algunas pocas herramientas de desarrollo de redux-devtools.

```
import { createStore, combineReducers, applyMiddleware, compose } from 'redux'
import thunk from 'redux-thunk'
import DevTools from './containers/DevTools'
import reducer from '../reducers/index'

const store = createStore(
  reducer,
  compose(
    applyMiddleware(thunk),
    DevTools.instrument()
  )
)
```

Consejos

- Todo lo que `compose` hace es permitirte escribir funciones transformadoras anidades fácilmente. ¡No le des mucho crédito!

Registro de cambios

Este proyecto se adhiere al Versionamiento Semántico.

Cada lanzamiento, junto a sus instrucciones de migración, están documentados en la página de Github de Lanzamientos.

Patrocinadores

El trabajo con Redux fue financiado por la comunidad.

Algunos de las compañías e individuos más destacadas que hicieron esto posible:

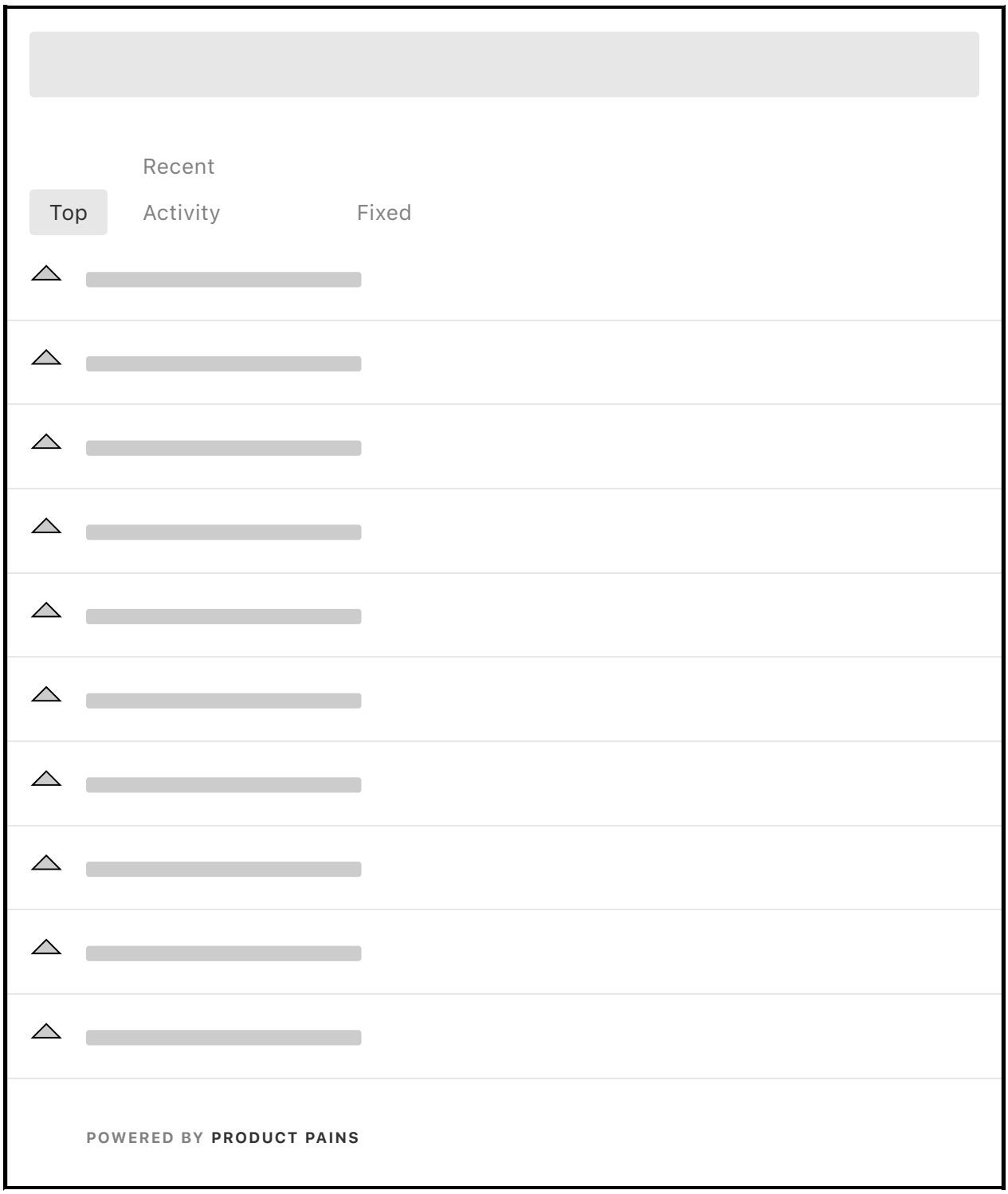
- Webflow
- Chess iX
- Herman J. Radtke III
- Ken Wheeler
- Chung Yen Li
- Sunil Pai
- Charlie Cheever
- Eugene G
- Matt Apperson
- Jed Watson
- Sasha Aickin
- Stefan Tennigkeit
- Sam Vincent
- Olegzandr Denman

Feedback

Apreciamos el feedback de la comunidad. Puedes publicar peticiones de características y reportes de bugs en Product Pains.

Traducción

Cualquier feedback específico de esta traducción lo puedes dejar en el repositorio de Github de la misma.


Recent
Top
Activity
Fixed
POWERED BY PRODUCT PAINS