Redux

Экшены, генераторы экшенов

Экшены (Actions) — это структуры, которые передают данные из вашего приложения в стор. Например, вот экшен, которое представляет добавление нового пункта в список дел:

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

Экшены — это обычные JavaScript-объекты. Экшены должны иметь поле **type**, которое указывает на тип исполняемого экшена. Типы должны быть, как правило, заданы, как **строковые константы**. Кроме type, структуру объекта экшенов вы можете строить на ваше усмотрение.

Генераторы экшенов (Action Creators) — не что иное, как функции, которые создают экшены. В Redux генераторы экшенов (action creators) просто возвращают action:

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

Результат передавать в функцию dispatch(). Кроме того, вы можете создать связанный генератор экшена (bound action creator), который автоматически запускает отправку экшена:

```
const boundAddTodo = (text) => dispatch(addTodo(text));
```

Редьюсеры¶

Редьюсеры (Reducers) определяют, как состояние приложения изменяется в ответ на экшены, отправленные в стор. Помните, что экшены только описывают, что произошло, но не описывают, как изменяется состояние приложения. Редьюсер (reducer) — это чистая функция, которая принимает предыдущее состояние и экшен (**state и action**) и возвращает следующее состояние (новую версию предыдущего).

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

Вот список того, чего **никогда** нельзя делать в редьюсере:

- Непосредственно изменять то, что пришло в аргументах функции;
- Выполнять какие-либо сайд-эффекты: обращаться к API или осуществлять переход по роутам;
- Вызывать не чистые функции, например Date.now() или Math.random().

Мы начнем с определения начального состояния (initial state). В первый раз Redux вызовет редьюсер с неопределенным состоянием(**state === undefined**). Это наш шанс инициализировать начальное состояние приложения:

```
function todoApp(state = initialState, action) {
 // Пока не обрабатываем никаких экшенов и просто возвращаем состояние, которое приняли в качестве параметра
 return state;
}
```

Пример редьюсера:

```
// state - состояние, action - { type: "", payLoad: "?"}
const reducer = (state = defaultState, action) => {
    switch (action.type) {
        case "INCREASE":
        return { ...state, counter: state.counter + action.payLoad }; // возвращаем состояние. В данном случае ...state можно опустить
        case "DECREASE":
        return { ...state, counter: state.counter - action.payLoad };
        default:
        return state; // возвращаем начальное состояние
    }
};
```

Обратите внимание:

- **Мы не изменяем state**. Мы создаем копию с помощью **Object.assign()** или деструктуризации **{ ...state, ...newState }** .
- **Мы возвращаем предыдущую версию состояния (state) в default ветке.** Очень важно возвращать предыдущую версию состояния (state) для любого неизвестного/необрабатываемого экшена (action).
- Часто используют **switch**, но это не является обязательным требованием

Редьюсеров может быть и несколько. можно их разнести в отдельные файлы. Наконец, Redux предоставляет утилиту, называемую combineReducers()

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

const rootReducer = combineReducers({
 counter: counterReducer,
 users: usersReducer
}); // объединим reducer-ы
```

Все, что делает combineReducers() — это генерирует функцию, которая вызывает ваши редьюсеры с частью глобального состояния, которая выбирается в соответствии с их ключами, и затем снова собирает результаты всех вызовов в один объект. Т. к. combineReducers ожидает на входе объект, мы можем поместить все редьюсеры верхнего уровня в разные файлы, экспортировать каждую функцию-редьюсер и использовать **import * as reducers** для получения их в формате объекта, ключами которого будут имена экспортируемых функций.

```
import { combineReducers } from 'redux';
import * as reducers from './reducers';

const todoApp = combineReducers(reducers);
```

Состояние приложения

В Redux все состояние приложения хранится в виде единственного объекта. Мы советуем поддерживать состояние (state) в настолько упорядоченном виде, насколько это возможно. Старайтесь не допускать никакой вложенности. Держите каждую сущность в объекте, который хранится с ID в качестве ключа. Используйте этот ID в качестве ссылки из других сущностей или списков.

В предыдущих разделах мы определили экшены, которые представляют факт того, что "что-то случилось" и редьюсеры, которые обновляют состояние (state) в соответствии с этими экшенами. Стор (Store) — это объект, который соединяет эти части вместе. Стор берет на себя следующие задачи:

- содержит состояние приложения (application state);
- предоставляет доступ к состоянию с помощью getState();
- предоставляет возможность обновления состояния с помощью dispatch(action);
- Обрабатывает отмену регистрации слушателей с помощью функции, возвращаемой subscribe(listener).

Важно отметить, что у вас будет **только один стор** в Redux-приложении.

Очень легко создать стор (Store), если у Вас есть **редьюсер**. В предыдущем разделе мы использовали combineReducers() для комбинирования несколько редьюсеров в один глобальный редьюсер. Теперь мы их импортируем и передадим в createStore().

```
import { createStore } from 'redux';
import todoApp from './reducers';
let store = createStore(todoApp);   //1-ый аргумент - редьюсер
```

Вы можете объявить **начальное состояние**, передав его **вторым аргументом** в createStore(). Это полезно для пробрасывания состояния на клиент из состояния приложения Redux, работающего на сервере.

Для того чтобы использовать Redux вместе с React существует компонент <Provider />, который оборачивает всё наше приложение и передаёт хранилище **store** всем дочерним элементам.
index.js:

```
import React from "react";
import ReactDOM from "react-dom/client";
import { Provider } from "react-redux";
import App from "./App";
import { store } from "./toolkitSliceRedux";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
```

```
 <Provider store={store}>
   <App />
 </Provider>
);
```

Асинхронность

Миддлвар - просто некая программа, которая выполняет определенные промежуточные действия или вычисления и передает их другой программе.

Для асинхронности в Redux используется Redux Thunk middleware. Этот мидлвар (middleware) содержится в отдельном пакете, который называется redux-thunk. При использовании конкретно этого мидлвара, генератор экшенов может вернуть **функцию, вместо объекта экшена**.

Redux-middleware является функция, которая получает данные после отправки экшена. Далее она может их проверить или сделать что-то с ними и затем передать дальше по цепочке в редьюсер. Благодаря middleware в Redux имеется замечательная возможность производить нужные промежуточные действия в момент, когда пользователь взаимодействует с интерфейсом. Например, мы можем логировать данные, получать дополнительные данные, создавать нужные задержки и тд.

index.js

```
import thunk from 'redux-thunk';
import rootReducer from './reducers';
import App from './App';

// используй applyMiddleware, чтобы добавить thunk middleware к стору
const store = createStore(rootReducer, applyMiddleware(thunk));
```

App.js:

```
import { addManyUsersAction } from "./store/usersReducer";
...
 //добавили функцию для обработки сетевого запроса
 const fetchusers = () => {
      return function (dispatch) {
      fetch("https://jsonplaceholder.typicode.com/users")
      .then((response) => response.json())
      .then((data) => dispatch(addManyUsersAction(data)));
      };
 };
 return (
 <div className="App">
     //добавили кнопку и обработчик
     <button onClick={() => dispatch(fetchusers())}>Добавить пользователей из базы</button>
</div>
 );
```

Thunk middleware — это не единственный путь управления асинхронными экшенами в Redux.

- Вы можете использовать redux-promise или redux-promise-middleware для отправки Promises вместо функций.
- Вы можете использовать redux-observable для отправки Observables
- Вы можете использовать мидлвар redux-saga для создания более комплексных асинхронных экшенов
- Вы можете использовать мидлвар redux-pack для отправки асинхронных экшенов, базирующихся на промисах
- Вы даже можете писать **собственные мидлвары**, для описания вызовов вашего API.

Асинхронный мидлвар, типа redux-thunk или redux-promise, **оборачивает метод стора dispatch()** и позволяет вам вызывать что-то, что не является экшеном, например, функции или Промисы.

Redux Toolkit

Redux Toolkit - это пакет, облегчающий работу с Redux.

Установим @reduxjs/toolkit:
npm i @reduxjs/toolkit

Redux Toolkit включает в себя следующие API:

- configureStore(): обертка для createStore(), упрощающая настройку хранилища с настройками по умолчанию. Позволяет автоматически комбинировать отдельные частичные редукторы (slice reducers), добавлять промежуточные слои или посредников (middlewares), по умолчанию включает redux-thunk (преобразователя), позволяет использовать расширение Redux DevTools (инструменты разработчика Redux)

- createReducer(): позволяет использовать таблицу поиска (lookup table) операций для редукторов случая (case reducers) вместо инструкций switch. В данном API используется библиотека immer, позволяющая напрямую изменять иммутабельный код, например, так: state.todos[3].completed = true
- createAction(): генерирует создателя операции (action creator) для переданного типа операции. Функция имеет переопределенный метод toString(), что позволяет использовать ее вместо константы типа
- createSlice(): принимает объект, содержащий редуктор, название части состояния (state slice), начальное значение состояния, и автоматически генерирует частичный редуктор с соответствующими создателями и типами операции
- createAsyncThunk(): принимает тип операции и функцию, возвращающую промис, и генерирует thunk, отправляющий типы операции pending/fulfilled/rejected на основе промиса
- createEntityAdapter(): генерирует набор переиспользуемых редукторов и селекторов для управления нормализованными данными в хранилище
- утилита createSelector() из библиотеки Reselect