ASTON

Интенсив-курс по React JS

astondevs.ru

Занятие 5. Redux

1. Redux
2. Принципы Redux
3. Как работает Redux. Составляющие Redux
4. Подключение ReduxDevTools
5. Подключаем наш компонент к Redux
6. Redux Hooks
7. Redux middleware
8. Redux Thunk
9. Redux Toolkit

Redux

Redux – это стабильный (предсказуемый) контейнер для хранения состояния JavaScript приложений, основанный на паттерне проектирования Flux. Redux может использоваться с React и любой другой библиотекой. Он легкий (около 2 Кб) и не имеет зависимостей. Чтобы подключить Redux к нашему приложению необходимо установить 2 библиотеки – сам redux и react-redux.

```
npm install redux react-redux
```

Преимущества Redux

1. Хранилище позволяет любому компоненту получать состояние без передачи пропсов
2. Состояние сохраняется даже при размонтировании компонента
3. Предотвращает ненужные повторные рендеринги благодаря поверхностному сравнению нового и старого состояния
4. Разделение UI и управления данными облегчает тестирование
5. Сохраняется история изменения состояния, что позволяет легко повторять или отменять операции

Redux недостатки

1. Отсутствует инкапсуляция. Любой компонент имеет доступ к данным, что может привести к проблемам с безопасностью
2. Много шаблонного кода. Ограниченный дизайн
3. Поскольку состояние является иммутабельным, reducer обновляет его, каждый раз возвращая новое состояние, что влечет дополнительные расходы памяти

Ключевые принципы Redux

- Единственный источник истины – состояние всего приложения хранится в древовидном объекта - в одном хранилище. Единственное состояние-дерево облегчает наблюдение за изменениями и отладку или инспектирование приложения.

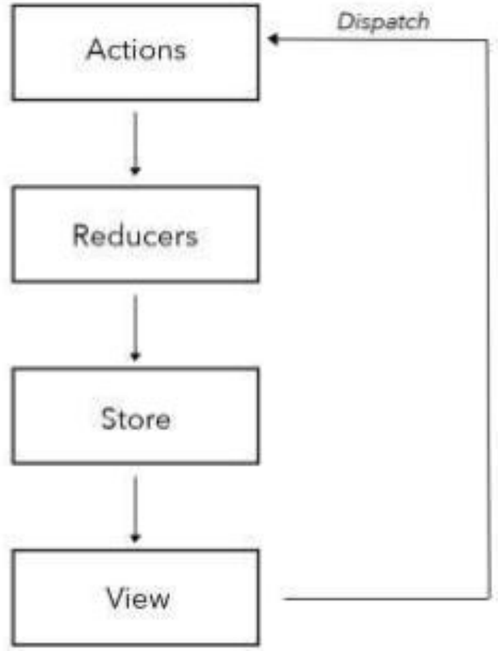
- Состояние доступно только для чтения – единственный способ изменить состояние заключается в запуске операции – объекте, описывающем произошедшее. Это позволяет гарантировать, что ни представления, ни сетевые коллбеки не буду иметь возможности изменять состояние напрямую.

- Изменения производятся с помощью "чистых" функций – для определения того, как изменяется состояние в зависимости от операции, создаются редукторы (reducers). Редукторы – это "чистые" функции, принимающие предыдущее состояние в качестве аргумента и возвращающие новое

Как работает Redux и его составляющие

Имеется центральное хранилище, содержащее состояние приложения.
Каждый компонент имеет доступ к этому хранилищу, так что нет необходимости передавать пропы из одного компонента в другой.

Существует три строительных блока: хранилище, редукторы (reducers) и операции (actions).



Actions

Action (операция) – это статическая информация о событии, инициализирующем изменение состояния. Обновление состояния в Redux всегда начинается с операции. Операции – это объекты, содержащие обязательное свойство type и опциональное свойство payload. Операции вызываются с помощью метода store.dispatch(). Операция создается с помощью "создателя операций" (action creator).

```
export const setProducts = products =>
  return {
    type: SET_PRODUCTS,
    payload: products,
  };
};
```

Создатели операций – функции, помогающие создавать операции. Создатель операции возвращает объект операции, который передается редуктору (reducer).

Reducers

Reducers (редукторы) – это "чистые" функции, принимающие текущее состояние приложения, выполняющие над ним операцию и возвращающие новое состояние. Новое состояние – объект, описывающий изменения состояния, произошедшие в ответ на вызванную операцию.

Это похоже на функцию reduce() в JavaScript, когда значение вычисляется на основе нескольких значений после выполнения коллбека. В React Redux можно создавать несколько reducer’ов и преобразовать их в один с помощью функции combineReducers().

```
const initialState = {
  products: [],
};

export const productReducer = (
  state = initialState,
  { type, payload }
) => {
  switch (type) {
    case SET_PRODUCTS:
      return {
        products: [...payload],
      };
    default:
      return state;
  }
}
```

Reducers

В React Redux можно создавать несколько reducer’ов и преобразовать их в один с помощью функции combineReducers(). Создание нескольких редукторов иногда очень удобно когда у нас приложение состоит из нескольких семантически независимых частей.

Например у нас интернет-магазин в котором есть и каталог товаров и корзина и личный кабинет пользователя. Чтобы не смешивать все обработчики в одном редукторе, куда читаемый и проще в обслуживании кода будет разбить это на отдельные редукторы, а потом это все объединить и передать в наш createStore().

```
rootReducer = combineReducers({
  potato: potatoReducer,
  tomato: tomatoReducer,
})
// Это создаст следующий объект состояния
{
  potato: {
    // ... potatoes и другое состояние управляемое potatoReducer ...
  },
  tomato: {
    // ... tomatoes и другое состояние управляемое tomatoReducer,
    // возможно, какой-нибудь хороший соус? ...
  }
}
```

State

State (еще называют Store) – это объект, содержащий состояние приложения. При обновлении состояния, обновляются все подписанные на него компоненты.

Хранилище отвечает за запись, чтение и обновление состояния. Store в react-redux создается с помощью функции createStore(). Которая принимает в себя reducer.

store.js

```
1  import { createStore } from "redux";
2  import reducers from "./reducers/index";
3
4  const store = createStore(
5    reducers,
6  );
7
8  export default store;
```

State

После того как мы создали Store. Мы передаем его в Provider который должен оборачивать все наше приложение, либо ту часть которой необходимо дать доступ к нашему Redux.

Provider – это компонент, содержащий ссылку на хранилище и передающий данные из хранилища дочерним компонентам.

index.js

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

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById( elementId: 'root')
);
```

Подключение ReduxDevTools

Redux DevTools – это Redux-окружение для "путешествий во времени" и "живого“ редактирования кода с возможностью "горячей“ перезагрузки, повторения операций и "кастомизируемым" интерфейсом.

Для того чтобы его подключить, необходимо скачать соответствующее расширение для вашего браузера (Chrome или Firefox). А так же добавить одну строчку кода в наше приложение. Передав её как второй аргумент функции createStore().

Более подробнее об установке и настройке Redux DevTools по ссылке на документацию.

Подключение ReduxDevTools

Так же Redux DevTools можно подключить с помощью установки npm пакета redux-devtoolsextension.

В таком случае вместо добавления строки мы просто используем функцию devToolsEnhancer. Либо, если нам так же необходимо подключить middleware – функцию composeWithDevTools.

store.js

```
import { devToolsEnhancer } from "redux-devtools-extension";
import { createStore } from "redux";
import reducers from "./reducers";

const store = createStore(
  reducers,
  devToolsEnhancer(),
);

export default store;
```

store.js

```
import { composeWithDevTools } from "redux-devtools-extension";
import {applyMiddleware, createStore} from "redux";
import reducers from "./reducers";
import thunk from "redux-thunk";

const store = createStore(
  reducers,
  composeWithDevTools(
    applyMiddleware(thunk),

  )
);

export default store;
```

Подключаем наш компонент к Redux

Есть несколько способов подключения компонента к хранилищу Redux. С помощью функции connect() или с помощью хука useSelector(). Для начала рассмотрим метод connect().

connect

connect – используется для создания компонентов контейнеров, которые подключены к хранилищу Redux. Хранилище, к которому осуществляется подключение, получают от самого верхнего предка компонента с использованием механизма контекста React.
Функция connect возвращает компонент высшего порядка (HOC) который оборачивает наш компонент и таким образом мы получаем компонент уже подключенный к нашему Redux.

```
function connect(mapStateToProps, mapDispatchToProps) {
  return function (WrappedComponent) {
      // здесь что-то происходит
  }
}
```

В качестве параметров функция connect принимает два аргумента:

- mapStateToProps()
- mapDispatchToProps()

mapStateToProps

Аргумент mapStateToProps является функцией, которая возвращает либо обычный объект, либо другую функцию. Передача этого аргумента connect() приводит к подписке компонента контейнера на обновления хранилища Redux.

Это означает, что функция mapStateToProps будет вызываться каждый раз, когда состояние хранилища изменяется.

Если вам слежение за обновлениями состояния не интересно, передайте connect() в качестве значения этого аргумента null.

```
function mapStateToProps(state) {
  const { todos } = state
  return { todoList: todos.allIds }
}

export default connect(mapStateToProps)(TodoList)
```

mapStateToProps

Функция mapStateToProps объявляется с двумя параметрами, второй из которых является необязательным. Первый параметр представляет собой текущее состояние хранилища Redux.
Второй параметр, если его передают, представляет собой объект свойств, переданных компоненту.
Если из mapStateToProps будет возвращён обычный объект, то возвращённый объект stateProps объединяется со свойствами компонента.
Если же mapStateToProps возвращает функцию, то эта функция используется как mapStateToProps для каждого экземпляра компонента. Это может пригодиться для улучшения производительности рендеринга и для мемоизации.

```
36  const mapStateToProps = (store, props) => ({
37    products: store.allProducts.products,
38  })
```

```
41  function mapStateToPropsFactory(initialState, ownProps) {
42    // a closure for ownProps is created
43    // this factory is not invoked everytime the component
44    // changes it's props
45    return function mapStateToProps(state) {
46      return {
47        blogs:
48          state.blogs.filter(blog => blog.author === ownProps.user)
49      };
50    };
51  }
```

mapDispatchToProps

Если обьяснить простыми словами то mapDispatchToProps “подключает” наши action’ы к пропсам React-компонента.

Каждая функция переданная в объект mapDispatchToProps будет воспринята в качестве генератора действий Redux и обёрнута в вызов метода store’a dispatch().

```
55  const mapDispatchToProps = {
56    setAllProducts,
57  }
```

После этого нашему компоненту из пропсов будут доступны эти функции как функции изменения состояния Redux.

Redux Hooks

Начиная с версии 7.1.0 Redux предоставляет нам такие хуки как:

- useSelector(selector: Function, equalityFn?: Function)
- useDispatch()

useSelector

useSelector приблизительно эквивалентен mapStateToProps. В качестве аргумента селектор будет передавать Redux state и будет вызываться когда компонент перерендеривается, так же он подписывается на store и вызывается каждый раз при изменении.

```
import React from 'react'
import { useSelector } from 'react-redux'

export const CounterComponent = () => {
  const counter = useSelector((state) => state.counter)
  return <div>{counter}</div>
}
```

Однако селектор будет производить сравнение (по умолчанию является строгим ===) предыдущего значения результата селектора и текущего значения результата. Если они отличаются, компонент будет вынужден повторно выполнить рендеринг. С useSelector возвращение нового объекта каждый раз по умолчанию будет вызывать повторный рендеринг.
Одним из вариантов возможно использование shallowEqual функции из React-Redux в качестве второго аргумента useSelector().

useDispatch

Этот хук возвращает ссылку на dispatch функцию из Redux. Вы можете использовать его для отправки действий.

```
import React, { useCallback } from 'react'
import { useDispatch } from 'react-redux'

export const CounterComponent = ({ value }) => {
  const dispatch = useDispatch()
  const incrementCounter = useCallback(
    () => dispatch({ type: 'increment-counter' }),
    [dispatch]
  )

  return (
    <div>
      <span>{value}</span>
      <MyIncrementButton onIncrement={incrementCounter} />
    </div>
  )
}

export const MyIncrementButton = React.memo(({ onIncrement }) => (
  <button onClick={onIncrement}>Increment counter</button>
))
```

Redux middlewares

Middleware (посредники) – это функции, которые вызываются последовательно в процессе обновления store.

Они встраиваются в хранилище при его создании и потом при диспатчинге данные проходят через них, до попадания в reducer.

Посредники используются для отправки асинхронных action’ов.

Для подключения middleware’ов используется функция applyMiddleware. В которую мы передаем наши middleware’ы и потом передаем вторым аргументом в функцию createStore.

store.js

```
import {applyMiddleware, compose, createStore} from "redux";
import reducers from "./reducers";
import thunk from "redux-thunk";

const store = createStore(
  reducers,
  compose(
    applyMiddleware(thunk),
    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__(),
  )
);

export default store;
```

Redux-Thunk

Redux Thunk – это middleware библиотека, которая позволяет вам вызвать action creator, возвращая при этом функцию вместо объекта.

Это позволяет нам выполнять внутри action creator’ов асинхронные операции. Возвращаемая функция принимает метод dispatch и getState как аргументы.

После того, как асинхронная операция завершится, мы можем использовать dispatch для диспатчинга обычного синхронного экшена.

Второй аргумент getState(), при вызове, возвращает нам актуальный объект хранилища Redux.

products.js

```
import {SET_PRODUCTS} from "../../constants/actionTypes";
import {getAllProducts, getProductById} from "../../api/products";

export const setAllProducts = () => (dispatch, getState) =>  {
  getAllProducts()
    .then(products => {
      dispatch({
        type: SET_PRODUCTS,
        payload: products,
      })

      console.log(getState());
    });
}
```

Redux-Thunk

После чего используем наш асинхронный экшн в компоненте так же как обычный экшн.

```jsx
import {useEffect} from "react";
import {connect} from "react-redux";
import {setAllProducts} from "../../redux/actions/products";
import ProductItem from "../ProductItem/ProductItem";
import "./styles.css";

const ProductSPage = ({ products, setAllProducts }) => {
  useEffect( effect: () => {
    setAllProducts();
  }, deps: []);

  return (
    <div>
      <h2>Product List</h2>
      <div className="container">
        {Boolean(products.length) && products.map(({ id, image, price, title }) => (
          <ProductItem
            id={id}
            key={id}
            image={image}
            price={price}
            title={title}
          />
        ))}
      </div>
    </div>
  );
}

const mapStateToProps = store => ({
  products: store.allProducts.products,
})

const mapDispatchToProps = {
  setAllProducts,
}

export default connect(mapStateToProps, mapDispatchToProps)(ProductSPage);
```

Redux-Toolkit

Redux Toolkit - это набор утилит, облегчающих работу с Redux - инструментом для управления состоянием приложений.

Он был разработан для решения трех главных проблем:

1. Слишком сложная настройка хранилища (store)
2. Для того, чтобы заставить Redux делать что-то полезное, приходится использовать дополнительные пакеты
3. Слишком много шаблонного кода (boilerplate)

```
# app-name - это название приложения
yarn create react-app app-name --template redux
# или
npx create-react-app app-name --template redux
```

```
yarn create react-app app-name --template redux-typescript
# или
npx create-react-app app-name --template redux-typescript
```

```
yarn add @reduxjs/toolkit
# или
npm i @reduxjs/toolkit
```

Redux-Toolkit

Redux Toolkit включает в себя следующие API:

- **configureStore**(): обертка для createStore(), упрощающая настройку хранилища с настройками по умолчанию.
- **createReducer**(): позволяет использовать таблицу поиска (lookup table) операций для редукторов случая (case reducers) вместо инструкций switch.
- **createAction**(): генерирует создателя операции (action creator) для переданного типа операции.
- **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

Redux-Toolkit настройка

Ручная настройка

Проблемы данного подхода:

- Аргументы (rootReducer, preloadedState, enhancer) должны быть переданы функции createStore() в правильном порядке
- Процесс настройки middlewares и enhancers (усилителей) может быть сложным, особенно, при попытке добавить несколько частей конфигурации
- Документация Redux DevTools Extension рекомендует использовать некоторый код для проверки глобального пространства имен для опеределения доступности расширения. Копирование/вставка предложенного сниппета усложняет последующее изучение кода

```
import { applyMiddleware, createStore } from 'redux'
import { composeWithDevTools } from 'redux-devtools-extension'
import thunkMiddleware from 'redux-thunk'

import monitorReducersEnhancer from './enhancers/monitorReducers'
import loggerMiddleware from './middleware/logger'
import rootReducer from './reducers'

export default function configureStore(preloadedState) {
  const middlewares = [thunkMiddleware, loggerMiddleware]
  const middlewareEnhacer = applyMiddleware(...middlewares)

  const enhancers = [monitorReducersEnhancer, middlewareEnhacer]
  const composedEnhancers = composeWithDevTools(...enhancers)

  const store = createStore(rootReducer, preloadedState, composedEnhancers)

  if (process.env.NODE_ENV !== 'production' && module.hot) {
    module.hot.accept('./reducers', () => store.replaceReducer(rootReducer))
  }

  return store
}
```

Redux-Toolkit настройка

Упрощение настройки с помощью configureStore()

configureStore() помогает решить названные проблемы следующим образом:

- Принимает объект с "именованными" параметрами, что облегчает изучение кода
- Позволяет передавать массив middlewares и enhancers, автоматически вызывая applyMiddleware() и compose()
- автоматически включает расширение Redux DevTools

```
import { configureStore } from '@reduxjs/toolkit'
import rootReducer from './reducers'

const store = configureStore({
  reducer: rootReducer
})

export default store
```

Redux-Toolkit Создание редукторов

Редукторы - это самая важная часть Redux.

- Определение характера ответа на основе поля type объекта операции
- Обновление состояния посредством копирования части состояния и модификации этой копии

Функция createReducer() похожа на функцию создания поисковой таблицы из документации по Redux. В ней используется библиотека immer, что позволяет писать "мутирующий" код, обновляющий состояние иммутабельно.

```
function todosReducer(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO': {
      return state.concat(action.payload)
    }
    case 'TOGGLE_TODO': {
      const { index } = action.payload
      return state.map((todo, i) => {
        if (i !== index) return todo

        return {
          ...todo,
          completed: !todo.completed
        }
      })
    }
    case 'REMOVE_TODO': {
      return state.filter((todo, i) => i !== action.payload.index)
    }
    default:
      return state
  }
}
```

Redux-Toolkit Определение создателей операции

С помощью **createReducer**() мы можем сократить приведенный пример

```
const todosReducer = createReducer([], builder => {
  builder
    .addCase('ADD_TODO', (state, action) => {
      // "мутируем" массив, вызывая push()
      state.push(action.payload)
    })
    .addCase('TOGGLE_TODO', (state, action) => {
      const todo = state[action.payload.index]
      // "мутируем" объект, перезаписывая его поле `completed`
      todo.completed = !todo.completed
    })
    .addCase('REMOVE_TODO', (state,action) => {
      // мы по-прежнему можем использовать иммутабельную логику обновления сос
      return state.filter((todo,i) => i !== action.payload.index)
    })
})
```

Redux-Toolkit Создание редукторов

Определение создателей операции с помощью **createAction**()

createAction() также принимает аргумент-колбек prepare, позволяющий кастомизировать результирующее поле payload и добавлять поле meta, при необходимости.

```
const addTodo = createAction('ADD_TODO')
addTodo({ text: 'Buy milk' })
// { type : "ADD_TODO", payload : {text : "Buy milk"}} )
```

Redux-Toolkit Создание частей состояния

createSlice() автоматически генерирует типы и создателей операции на основе переданного названия редуктора:

```
const postsSlice = createSlice({
  name: 'posts',
  initialState: [],
  reducers: {
    createPost(state, action) {},
    updatePost(state, action) {},
    deletePost(state, action) {}
  }
})

console.log(postsSlice)
/*
{
  name: 'posts',
  actions : {
      createPost,
      updatePost,
      deletePost,
  },
  reducer
}
*/

const { createPost } = postsSlice.actions

console.log(createPost({ id: 123, title: 'Привет, народ!' }))
// { type : 'posts/createPost', payload : { id : 123, title : 'Привет, народ!'
```

Redux-Toolkit асинхронные запросы

createAsyncThunk() упрощает процесс выполнения асинхронных запросов - мы передаем ему строку для префикса типа операции и колбек создателя полезной нагрузки (payload), выполняющего реальную асинхронную логику и возвращающего промис с результатом. createAsyncThunk() возвращает преобразователя, который заботится об отправке правильных операций на основе возвращенного промиса, и типы операции, которые можно обработать в редукторах:

```
import { createAsyncThunk, createSlice } from '@redux/toolkit'
import { userAPI } from './userAPI'

// Создаем преобразователя
const fetchUserById = createAsyncThunk(
  'user/fetchByIdStatus',
  async (userId, thunkAPI) => {
    const response = await userAPI.fetchById(userId)
    return response.data
  }
)

// Обрабатываем операции в редукторах
const usersSlice = createSlice({
  name: 'users',
  initialState: {entries: [], loading: 'idle'},
  reducers: {
    // стандартная логика редуктора с авто-генерируемыми типами операции дл
  },
  extraReducers: {
    [fetchUserById.fullfilled]: (state, action) => {
      // Добавляем пользователя в массив состояния
      state.entries.push(action.payload)
    }
  }
})

// Позже, отправляем `thunk`
dispatch(fetchUserById(123))
```

Redux-Toolkit нормализация

Нормализация с помощью **createEntityAdapter**()

createEntityAdapter() предоставляет стандартизированный способ хранения данных путем преобразования коллекции в форму { ids: [], entities: {} }.

```
import {
  createSlice,
  createAsyncThunk,
  createEntityAdapter
} from '@reduxjs/toolkit'
import userAPI from './userAPI'

export const fetchUsers = createAsyncThunk('users/fetchAll', async () => {
  const response = await userAPI.fetchAll()
  // В данном случае `response.data` будет выглядеть так:
  // [{id: 1, first_name: 'Example', last_name: 'User'}]
  return response.data
})

export const updateUser = createAsyncThunk('users/updateOne', async arg => {
  const response = await userAPI.updateUser(arg)
  // В данном случае `response.data` будет выглядеть так:
  // { id: 1, first_name: 'Example', last_name: 'UpdatedLastName'}
  return response.data
})

export const usersAdapter = createEntityAdapter()

// По умолчанию `createEntityAdapter()` возвращает `{ ids: [], entities: {} }`
// Для отслеживания 'loading' или других ключей, их необходимо инициализировать
// `getInitialState({ loading: false, activeRequestId: null })`
const initialState = usersAdapter.getInitialState()

export const slice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    removeUser: usersAdapter.removeOne
  },
  extraReducers: builder => {
    builder.addCase(fetchUsers.fulfilled, usersAdapter.upsertMany)
    builder.addCase(updateUser.fulfilled, (state, { payload }) => {
      const { id, ...changes } = payload
      usersAdapter.updateOne(state, { id, changes })
    })
  }
})

const reducer = slice.reducer
export default reducer

export const { removeUser } = slice.actions
```

Redux-Toolkit настройка хранилища

cofigureStore() - абстракция над стандартной функцией createStore(), добавляющая полезные "дефолтные" настройки хранилища для лучшего опыта разработки.

cofigureStore() принимает следующий объект:

```
333  import { createStore } from '@reduxjs/toolkit'
334
335  import rootReducer from './reducers'
336
337  const store = configureStore({ reducer: rootReducer })
338  // В хранилище автоматически добавляется `redux-thunk` и расширение `Redux DevTools`
339
```

```
291  {
292    /**
293     * Функция редуктора, используемая в качестве корневого редуктора,
294     * или объект с частичными редукторами, автоматически передаваемыми в `combineReducers()`
295     */
296    reducer: func | object,
297
298    /**
299     * Массив посредников, автоматически передаваемых в `applyMiddleware()`,
300     * или набор дефолтных посредников, возвращаемый `getDefaultMiddleware()`
301     */
302    middleware?: array,
303
304    /**
305      * Интеграция с инструментами разработчика `Redux`
306      *
307      * Допускается дополнительная настройка
308     */
309    devTools?: bool | object,
310
311    /**
312     * Начальное состояние, аналогичное начальному состоянию `createStore()`.
313     * Это можно использовать для гидратации состояния,
314     * полученного от сервера в универсальных приложениях,
315     * или для восстановления ранее сериализованной сессии пользователя.
316     * При использовании `combineReducers()` для получения корневого редуктора,
317     * `preloadedState` должен быть объектом аналогичной формы (с такими же ключами, что и редуктор)
318     */
319    preloadedState?: any,
320
321    /**
322     * Массив усилителей хранилища, автоматически передаваемых в `compose()`.
323     * Все усилители будут включены до расширения `DevTools`.
324     * Если вам требуется кастомизировать порядок усилителей,
325     * используйте функцию обратного вызова, получающую оригинальный массив (например, `[applyMiddleware]`),
326     * и возвращающую новый массив (например, `[applyMiddleware, offline]`)
327     * Для добавления посредника можно использовать параметр `middleware`
328     */
329    enhancers?: array | func
330  }
```

Redux-Toolkit builder callback

Параметры:

initialState - начальное состояние, используемое при первом вызове редуктора

builderCallback - колбек, принимающий объект builder для определения редуктора случая

путем builder.addCase(actionCreatorOrType, reducer)

Методы:

addCase() - добавляет редуктора случая для определенного типа операции.

addMacther() - позволяет осуществлять проверку входящей операции с помощью собственных фильтров в дополнение к проверке типа.

addDefaultCase() - добавляет дефолтный случай, когда для операции не были выполнены другие редукторы

```
import { createAction, createReducer } from '@reduxjs/toolkit'

const increment = createAction('increment')
const decrement = createAction('decrement')

function isActionWithNumberPayload(action) {
  return typeof action.payload === 'number'
}

createReducer(
  {
    counter: 0,
    sumOfNumberPayloads: 0,
    unhandledActions: 0,
  },
  (builder) => {
    builder
      .addCase(increment, (state, action) => {
        state.counter += action.payload
      })
      // Можно создавать цепочки из вызовов `addCase`
      // или каждый раз писать `builder.addCase()`
      .addCase(decrement, (state, action) => {
        state.counter -= action.payload
      })
      // Можно использовать "функцию поиска совпадений" для входящих операций
      .addMatcher(isActionWithNumberPayload, (state, action) => {})
      // и указывать случай по умолчанию, если ни один из обработчиков не сраб
      .addDefaultCase((state, action) => {})
  }
)
```