React
асинхронный REDUX

Комбинирование reducer-ов
Асинхронные запросы Redux
MiddleWare Redux
Redux Thunk
Redux Saga

combineReducers

В Redux обычно используется один store для управления состоянием приложения. Однако это не значит, что нужно ограничиваться одним редьюсером. Вы можете объединить несколько редьюсеров с помощью функции **combineReducers**, которая создает один главный редьюсер, управляющий несколькими частями состояния.

```
import { combineReducers } from 'redux';
import usersReducer from './usersReducer';
import todosReducer from './todosReducer';

const rootReducer = combineReducers({
  users: usersReducer,
  todos: todosReducer,
});

export default rootReducer;
```

```
{
  "users": {
    "users": []
  },
  "todos": {
    "todos": []
  }
}
```

```
import { createStore } from 'redux';
import rootReducer from './reducers'; // импорт rootReducer

const store = createStore(rootReducer);
```

Асинхронные запросы в REDUX

Асинхронные запросы часто используются для работы с данными, такими как получение, отправка или обновление информации на сервере (например, через fetch или библиотеки вроде Axios). Как только сервер отвечает, веб-приложение может обновить данные и перерисовать интерфейс на основе полученной информации.

State manager (например, Redux, MobX или контекстный API в React) управляет состоянием приложения, что включает в себя отслеживание данных, которые изменяются во время работы приложения. Асинхронные запросы связаны с изменением состояния, потому что данные, полученные от сервера, должны быть сохранены в состоянии, чтобы интерфейс мог обновиться и показать пользователю актуальную информацию.

Таким образом, связывание асинхронных запросов с state manager делает приложение более структурированным и управляемым, особенно когда нужно координировать несколько запросов и состояний.

Пример асинхронных запросов через REDUX

```
// actions.js
export const fetchDataRequest = () => ({
  type: 'FETCH_DATA_REQUEST',
});

export const fetchDataSuccess = (data) => ({
  type: 'FETCH_DATA_SUCCESS',
  payload: data,
});

export const fetchDataFailure = (error) => ({
  type: 'FETCH_DATA_FAILURE',
  payload: error,
});
```

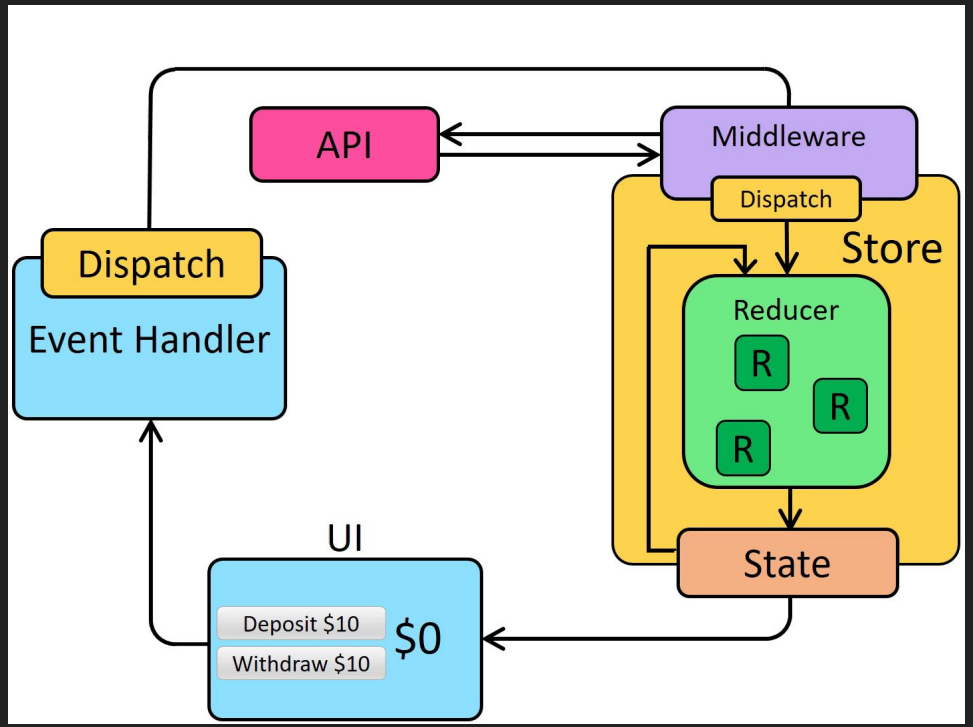
```
// reducer.js
const initialState = {
  loading: false,
  data: null,
  error: null,
};

const dataReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'FETCH_DATA_REQUEST':
      return { ...state, loading: true, error: null };
    case 'FETCH_DATA_SUCCESS':
      return { ...state, loading: false, data: action.payload };
    case 'FETCH_DATA_FAILURE':
      return { ...state, loading: false, error: action.payload };
    default:
      return state;
  }
};

export default dataReducer;
```

Механизм асинхронных запросов в REDUX

Механизм асинхронных запросов в Redux через middleware часто используется для выполнения побочных эффектов, таких как HTTP-запросы, которые не должны происходить непосредственно в редьюсерах, поскольку они должны оставаться чистыми функциями. Middleware предоставляет возможность перехватывать экшены, обрабатывать асинхронные операции и диспатчить новые экшены в зависимости от результатов этих операций.



MiddleWare

Middleware в Redux — это функции, которые перехватывают экшены между моментом их диспатча (вызова) и передачей в редьюсеры. Middleware предоставляет дополнительную функциональность, которая может быть полезной для решения ряда проблем, связанных с обработкой асинхронных операций, ведением логов, манипуляцией с экшенами и многим другим.

Middleware — это функции, которые имеют доступ к действию (action), диспатчу (dispatch) и состоянию (state), и могут:

- Изменять экшены.
- Выполнять побочные эффекты (например, сетевые запросы).
- Прерывать или повторять экшены.
- Добавлять дополнительную логику при обработке экшенов.

Асинхронные операции в Redux:

По умолчанию, Redux работает только с синхронными экшенами — обычными объектами. Но многие реальные приложения включают асинхронные задачи (например, запросы к API). Middleware, такие как redux-thunk или redux-saga, позволяют отправлять асинхронные экшены и управлять сложными операциями вроде API-запросов, таймеров и т.д.

```
const fetchData = () => {
  return (dispatch) => {
    dispatch({ type: 'FETCH_DATA_REQUEST' });

    fetch('https://api.example.com/data')
      .then((response) => response.json())
      .then((data) => dispatch({ type: 'FETCH_DATA_SUCCESS', payload: data }))
      .catch((error) => dispatch({ type: 'FETCH_DATA_FAILURE', payload: error }));
  };
};
```

Примеры популярных middleware:

redux-thunk: Позволяет диспатчить функции вместо объектов. Эти функции могут выполнять асинхронные операции, а затем диспатчить обычные экшены.

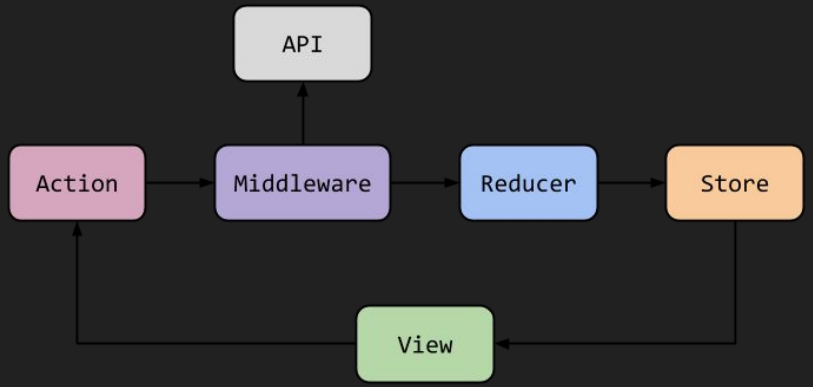
redux-saga: Позволяет работать с асинхронными операциями через генераторы и упрощает управление сложной асинхронной логикой.

redux-logger: Логирует все экшены и состояния до и после экшена, что полезно для отладки.

redux-promise: Позволяет диспатчить промисы в виде экшенов, которые автоматически разрешаются, как только промис выполнится.

Как подключить middleware?

Чтобы подключить middleware в Redux, используется функция applyMiddleware.




```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import logger from 'redux-logger';
import rootReducer from './reducers';

const store = createStore(
  rootReducer,
  applyMiddleware(thunk, logger) // подключаем несколько middleware
);
```

Свой кастомный middleWare асинхронных action-ов

Middleware — это функция, которая возвращает другую функцию, принимающую store (состояние и dispatch). Затем возвращается следующая функция, которая перехватывает каждый экшен, диспатчится через next(action) или обрабатывается каким-либо образом.

```
const customThunk = (store) => (next) => (action) => {
  if (typeof action === 'function') {
    // Если action — это функция, вызываем её и передаем dispatch и getState
    return action(store.dispatch, store.getState);
  }

  // Если action — это не функция (обычный объект), передаем его дальше
  return next(action);
};
```

```
import { createStore, applyMiddleware } from 'redux';
import rootReducer from './reducers';
import customThunk from './customThunk'; // наш middleware

const store = createStore(
  rootReducer,
  applyMiddleware(customThunk) // Подключаем наше middleware
);
```

Redux Thunk

redux-thunk — это middleware для Redux, которое позволяет вам писать асинхронные функции-диспатчи. Обычно в Redux вы можете диспатчить только объекты, но с помощью redux-thunk можно диспатчить функции. Это полезно для обработки асинхронных операций, таких как запросы к API, таймеры и т.д.

Redux Thunk перехватывает любое действие, которое вы пытаетесь отправить в Redux. Если действие является функцией, то эта функция получает доступ к методам dispatch и getState, что позволяет вам управлять потоком асинхронных действий и диспатчить другие действия после завершения асинхронной операции.

npm install redux-thunk

Проблемы, которые решает Redux Thunk

Redux изначально поддерживает только синхронные действия, и без redux-thunk вы бы не могли легко интегрировать асинхронные операции. Например, вам нужно сделать запрос к серверу, дождаться ответа, а затем обновить состояние приложения — redux-thunk решает эту задачу, предоставляя возможность выполнения логики до отправки действия в редьюсер.

Плюсы redux-thunk

- **Асинхронные действия**: Позволяет вам работать с асинхронным кодом (запросы к API, таймеры и т.д.) через Redux.
- **Гибкость:** Вы можете управлять тем, когда и как будут диспатчены действия.
- **Простота интеграции:** Легко интегрируется с существующими Redux-приложениями.

Минусы redux-thunk

- **Смешивание логики:** Бизнес-логика (например, обработка API) может перемешиваться с кодом компонентов или действий.
- **Трудности с тестированием:** Асинхронные функции могут быть сложнее тестировать по сравнению с чистыми действиями и редьюсерами.
- **Повышение сложности:** Использование асинхронных действий через redux-thunk может усложнять код, особенно в крупных проектах.

Пример использования Redux Thunk

```
export const fetchTodosRequest = () => ({
  type: 'FETCH_TODOS_REQUEST'
});

export const fetchTodosSuccess = (todos) => ({
  type: 'FETCH_TODOS_SUCCESS',
  payload: todos
});

export const fetchTodosFailure = (error) => ({
  type: 'FETCH_TODOS_FAILURE',
  payload: error
});

export const fetchTodos = () => {
  return async (dispatch) => {
    dispatch(fetchTodosRequest());

    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/todos');
      const data = await response.json();
      dispatch(fetchTodosSuccess(data));
    } catch (error) {
      dispatch(fetchTodosFailure(error.message));
    }
  };
};
```

```
const initialState = {
  loading: false,
  todos: [],
  error: null
};

const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'FETCH_TODOS_REQUEST':
      return {
        ...state,
        loading: true
      };
    case 'FETCH_TODOS_SUCCESS':
      return {
        loading: false,
        todos: action.payload,
        error: null
      };
    case 'FETCH_TODOS_FAILURE':
      return {
        loading: false,
        todos: [],
        error: action.payload
      };
    default:
      return state;
  }
};

export default todoReducer;
```

```
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { fetchTodos } from '../actions/todoActions';

const TodoList = () => {
  const dispatch = useDispatch();
  const { loading, todos, error } = useSelector(state => state.todos);

  useEffect(() => {
    dispatch(fetchTodos());
  }, [dispatch]);

  if (loading) {
    return <p>Loading...</p>;
  }

  if (error) {
    return <p>Error: {error}</p>;
  }

  return (
    <ul>
      {todos.slice(0, 10).map(todo => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </ul>
  );
};

export default TodoList;
```

Redux Saga

redux-saga — это middleware для Redux, которое позволяет работать с асинхронными действиями с помощью "car" — генераторов. Основное отличие от redux-thunk заключается в том, что redux-saga использует ES6 генераторы для описания асинхронных потоков данных, что делает его более мощным и гибким инструментом для управления побочными эффектами в Redux-приложениях.

Redux-saga интерпретирует "саги" как процессы, которые могут запускать асинхронные задачи, ждать их завершения, слушать действия, отправлять другие действия и выполнять любые другие побочные эффекты. Это позволяет вам управлять сложными потоками данных, сохраняя код чистым и легко тестируемым.

Сага — это функция, которая может приостанавливаться и возобновляться (с помощью yield), что упрощает обработку асинхронных операций. Эти функции слушают действия в Redux и выполняют логику, которая требует побочных эффектов (запросы к API, задержки и т.д.).

npm install redux-saga

Проблемы, которые решает Redux Saga

- **Асинхронные запросы:** Подобно redux-thunk, redux-saga решает проблему выполнения асинхронных действий в Redux.
- **Обработка сложных сценариев:** Он отлично подходит для сложных потоков данных, таких как многократные последовательные запросы, параллельные запросы и управление таймерами.
- **Обработка побочных эффектов:** Redux-saga позволяет легко отслеживать побочные эффекты в приложении, делая код более предсказуемым и тестируемым.

Плюсы redux-saga

- **Мощная обработка потоков данных:** Генераторы позволяют легко описывать сложные асинхронные процессы.
- **Тестируемость:** Генераторы можно тестировать шаг за шагом, что упрощает проверку бизнес-логики.
- **Гибкость:** Можно легко комбинировать параллельные задачи, обрабатывать ошибки, отменять запросы и т.д.
- **Изоляция побочных эффектов**: Логика побочных эффектов изолируется в саги, что делает код более чистым и поддерживаемым.

Минусы redux-saga

- **Сложность:** Код с генераторами может показаться сложнее для начинающих разработчиков.
- **Больше кода:** Саги иногда могут требовать больше кода для решения тех же задач, что и redux-thunk.
- **Порог вхождения:** ES6 генераторы требуют некоторого времени на изучение и понимание их работы.

Настройка reducer-а и action-ов

```
export const fetchTodosRequest = () => ({
  type: 'FETCH_TODOS_REQUEST'
});

export const fetchTodosSuccess = (todos) => ({
  type: 'FETCH_TODOS_SUCCESS',
  payload: todos
});

export const fetchTodosFailure = (error) => ({
  type: 'FETCH_TODOS_FAILURE',
  payload: error
});
```

```
const initialState = {
  loading: false,
  todos: [],
  error: null
};

const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'FETCH_TODOS_REQUEST':
      return { ...state, loading: true };
    case 'FETCH_TODOS_SUCCESS':
      return { loading: false, todos: action.payload, error: null };
    case 'FETCH_TODOS_FAILURE':
      return { loading: false, todos: [], error: action.payload };
    default:
      return state;
  }
};

export default todoReducer;
```

Создаем сагу для выполнения асинхронного запроса к API.

call — вызывает функцию (например, fetch для выполнения запроса).

put — диспатчит действия в Redux-стор.

takeEvery — прослушивает действия определенного типа и запускает указанную сагу (в данном случае fetchTodos).

```
import { call, put, takeEvery } from 'redux-saga/effects';
import { fetchTodosSuccess, fetchTodosFailure } from '../actions/todoActions';

function* fetchTodos() {
  try {
    const response = yield call(fetch, 'https://jsonplaceholder.typicode.com/todos');
    const data = yield response.json();
    yield put(fetchTodosSuccess(data));
  } catch (error) {
    yield put(fetchTodosFailure(error.message));
  }
}

export function* watchFetchTodos() {
  yield takeEvery('FETCH_TODOS_REQUEST', fetchTodos);
}
```

Комбинируем все саги (в нашем примере только одна).

Создаем Redux-стор и подключаем redux-saga.

```
import { all } from 'redux-saga/effects';
import { watchFetchTodos } from './todoSagas';

export default function* rootSaga() {
  yield all([watchFetchTodos()]);
}
```

```
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga';
import rootReducer from './reducers';
import rootSaga from './sagas/rootSaga';

const sagaMiddleware = createSagaMiddleware();

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

sagaMiddleware.run(rootSaga);

export default store;
```

Использование в компоненте

Компонент TodoList диспатчит действие fetchTodosRequest, которое запускает сагу fetchTodos.

Сага fetchTodos делает запрос к API, используя call, и при успешном ответе диспатчит действие fetchTodosSuccess с данными.

Если произошла ошибка, диспатчится fetchTodosFailure.

Редьюсер обновляет состояние в зависимости от типа действия, и список задач отображается в компоненте.

```jsx
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { fetchTodosRequest } from '../actions/todoActions';

const TodoList = () => {
  const dispatch = useDispatch();
  const { loading, todos, error } = useSelector(state => state.todos);

  useEffect(() => {
    dispatch(fetchTodosRequest());
  }, [dispatch]);

  if (loading) {
    return <p>Loading...</p>;
  }

  if (error) {
    return <p>Error: {error}</p>;
  }

  return (
    <ul>
      {todos.slice(0, 10).map(todo => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </ul>
  );
};

export default TodoList;
```

Ресурсы

Код с урока (git репозиторий) - ТЫК

Механизм асинхронных action-ов в REDUX - ТЫК

Продвинутый REDUX учебник - ТЫК

комбинирования редукторов в REDUX учебник - ТЫК

Документация комбинирования редукторов в REDUX - ТЫК

Документация REDUX THUNK - ТЫК

Документация REDUX SAGA - ТЫК