ReactJS. Базовый курс

Погружение в react-redux

[React 17.0.1]

На этом уроке

1. Продолжим изучать react-redux
2. Узнаем об action creators и научимся передавать в стор данные из компонентов
3. Научимся хранить данные в нескольких редьюсерах и использовать combineReducers
4. Познакомимся с connect.

Оглавление

Теория урока

Action creators. Передача данных из компонента в стор

На текущий момент в нашем приложении стор используется только компонентом Profile. При этом компонент не передает в стор никаких данных - для изменения состояния используются только данные самого стора:

```
const initialState = {
```

```
  showName: false,
  name: 'Default'
}

const profileReducer = (state = initialState, action) => {
  switch (action.type) {
    case TOGGLE_SHOW_NAME:
      return {
        ...state,
        showName: !state.showName
      }
      default:
        return state
  }
}
```

Добавим в компоненте Profile возможность вводить и сохранять в сторе имя пользователя.

Сперва уберем чекбокс и добавим поле для ввода и кнопку для отправки данных:

```
export default function Profile() {
 const { name } = useSelector((state) => state.profile);
 const dispatch = useDispatch();
const [value, setValue] = useState('');

 const setShowName = useCallback(() => {
   dispatch(toggleShowName);
 }, [dispatch]);

 const handleChange = useCallback((e) => {
    setValue(e.target.value);
 }, []);

 const setName = () => {};

 return (
   <>
     <div>
       <h4>Profile</h4>
     </div>
     <div>
       <input type="text" value={value} onChange={handleChange} />
     </div>
     <div>
       <button onClick={setName}>Change Name</button>
     </div>
   </>
 );
}
```

Теперь необходимо отправить введенное пользователем имя в стор. Для этого добавим новый тип экшена:

```
export const CHANGE_NAME = "PROFILE::CHANGE_NAME";
```

И будем диспатчить экшен следующим образом:

```
 const setName = useCallback(() => {
   dispatch({ type: CHANGE_NAME, payload: value })
 }, [dispatch, value]);
```

А в редьюсере необходимо добавить его обработку:

```
const profileReducer = (state = initialState, action) => {
  switch (action.type) {
    case CHANGE_NAME:
      return {
        ...state,
        name: action.payload
      }
    default: return state;
  }
}
```

Как видно из примеров, для передачи данных из компонента в объект экшена добавляется новое поле (обычно его обозначают payload). Экшен с данными попадает в редьюсер, который может соответствующим образом изменить стор - к примеру, просто переписать новыми данными старые.

Поскольку экшены зачастую используются во многих местах и с различными данными, диспатчить их, передавая литерал объекта, не представляется удобным. Для того, чтобы этого избежать, создают функции, называемые action creators - “создатели экшенов”. Как понятно из названия, такая функция должна вернуть новый объект экшена с переданными ей данными:

```
export const CHANGE_NAME = "PROFILE::CHANGE_NAME";

export const changeName = (newName) => ({
  type: CHANGE_NAME,
  payload: newName,
});
```

Теперь в компоненте мы можем задиспатчить наш экшен, вызвав функцию changeName:

```
  const setName = useCallback(() => {
    dispatch(changeName(value))
  }, [dispatch, value]);
```

Обратите внимание, что в dispatch передается **результат** вызова action creator, т.е. объект экшен с заранее определенным типом и данными, переданными через аргумент.

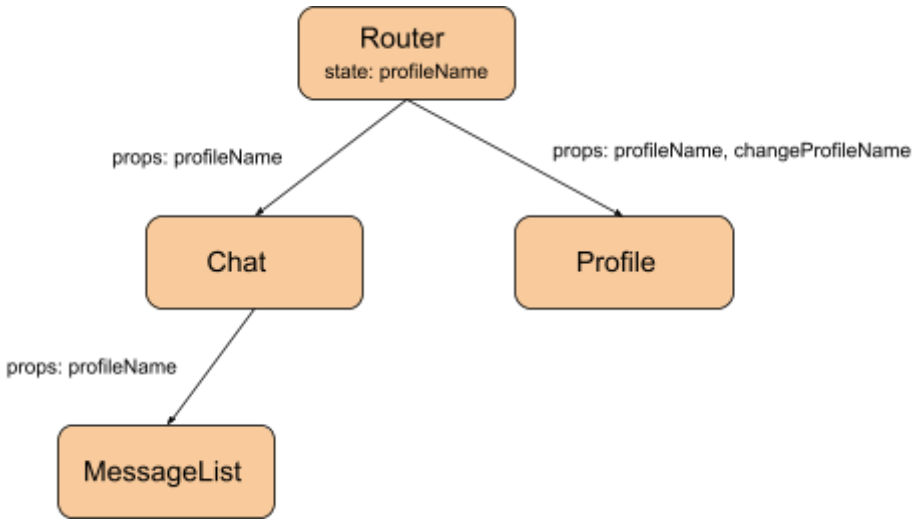
В дальнейшем для создания экшенов следует придерживаться только такого подхода - т.е. создавать их с помощью action creators.

Однако, особого смысла в использовании redux только в одном компоненте нет. Одна из основных причин использовать глобальный стор - это получение возможности доступа к одним и тем же данным из разных компонентов. К примеру, имя пользователя можно отображать как автора сообщений. Для этого в компоненте MessageList добавим useSelector:

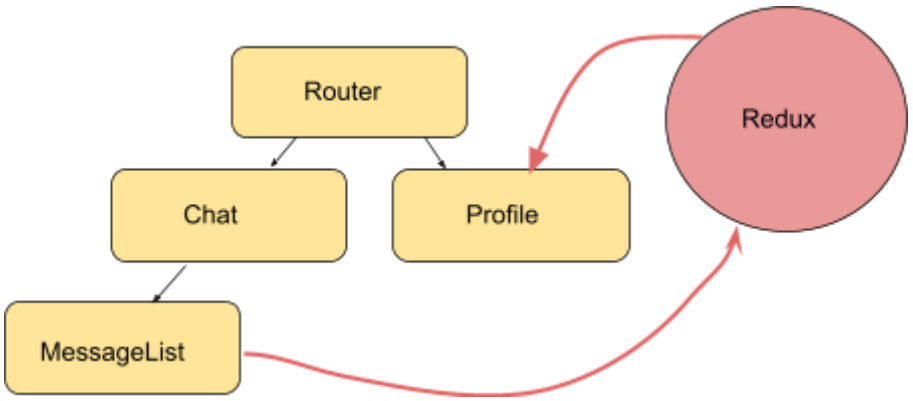
```
const profileName = useSelector(state => state.profile.name);

const renderMessage = useCallback((message, i) => (
  <div key={i}>
    <span>
      {message.author === AUTHORS.ME ? profileName : message.author}:
     </span>
    <span>{message.text}</span>
  </div>
), [profileName]);
```

Обратите внимание, насколько более сложной могла бы быть реализация такого функционала без единого хранилища:

Без redux (указаны только данные, относящиеся к профилю):



С redux:



combineReducers

Давайте перенесем теперь в стор и данные о чатах. Для начала создадим для них отдельные редьюсер и экшены:

store/chats/actions.js

```
export const ADD_CHAT = "CHATS::ADD_CHAT";

export const addChat = (name) => ({
  type: ADD_CHAT,
  name,
});
```

store/chats/reducer.js

```
import { ADD_CHAT } from "./actions";

const initialState = {
  chatList: [],
};

const chatsReducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_CHAT:
      return {
        ...state,
        chatList: [
          ...state.chatList,
          {
            id: `id${state.chatList.length}`,
            name: action.name,
          },
        ],
```

```
      };
    default:
      return state;
  }
};

export default chatsReducer;
```

store/messages/actions.js

```
export const ADD_MESSAGE = 'MESSAGES::ADD_MESSAGE';

export const addMessage = (chatId, message) => ({
  type: ADD_MESSAGE,
  chatId,
  message,
});
```

store/messages/reducer.js

```
import { ADD_MESSAGE } from "./actions";

const initialState = {
  // to be stored like this {[chatId]: [{id, text, author}]}
  messageList: {},
};

const chatsReducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_MESSAGE: {
      const currentList = state.messageList[action.chatId] || [];
      return {
        ...state,
        messageList: {
          ...state.messageList,
          [action.chatId]: [
            ...currentList,
            {
              ...action.message,
              id: `${action.chatId}${currentList.length}`,
            },
          ],
        },
      };
    }
    default:
      return state;
  }
};

export default chatsReducer;
```

`

Заметьте, что типы экшенов начинаются с названия редьюсера. Это, во-первых, обеспечит уникальность типов экшенов, а во-вторых, позволит легче отслеживать экшены в devtools.

Теперь подключим наши редьюсеры к стору. Для того, чтобы использовать в одном сторе несколько редьюсеров, необходимо воспользоваться функцией combineReducers при создании стора:

store/index.js

```
import { createStore, combineReducers } from "redux";

import { chatsReducer } from "./chats/reducer";
import { profileReducer } from "./profile/reducer";
import { messagesReducer } from "./messages/reducer";

export const store = createStore(
combineReducers({
chats: chatsReducer,
profile: profileReducer,
messages: messagesReducer,
}),
window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

Взглянем на стор в devtools:

Теперь наш стор разделен на три составляющих - profile, chats и messages. Такой подход удобен для разделения данных - редьюсер профиля будет отвечать только за данные, связанные с профилем, и обрабатывать только связанные с ним экшены, редьюсер chats - работать только с чатами, а за добавление сообщений отвечает редьюсер messages.

Теперь заменим логику получения сообщений в компоненте MessageField - вместо пропсов будем получать их из стора через useSelector:

```
  const chats = useSelector((state) => state.chats.chatList);
  const messages = useSelector(state => state.messages.messageList);
```

А также изменим логику отправки сообщения - вместо вызова setMessages будем диспатчить соответствующий экшен:

```
  const dispatch = useDispatch();

  const onAddMessage = (message) => {
    dispatch(addMessage(chatId, message));
  }
```

После этого можно освободить стейт компонента от ненужных данных. Логику ответа робота также перенесем в MessageField.

Теперь данные, которые нужны нескольким компонентам приложения, находятся в сторе. Любой компонент может легко получить их или изменить.

Реализация добавления чатов

Однако, пока что у нас нет ни одного чата - так как стор инициализирован пустым массивом. Сделаем возможным добавление чата. Экшен для этого и его обработка в редьюсере у нас уже есть.

Добавим в компоненте ChatList кнопку для добавления нового чата. По нажатию на нее покажем диалоговое окно (используем компонент Dialog из @materil-ui).

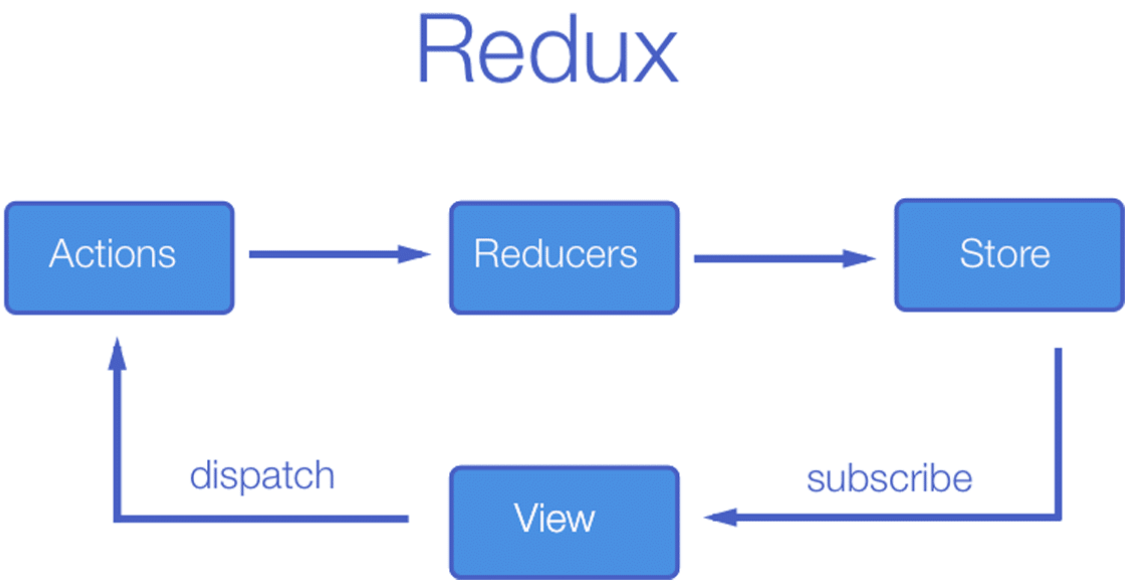
```
export default ({ chatId }) => {
  const [visible, setVisible] = useState(false);
  const [newChatName, setNewChatName] = useState("");

  const chats = useSelector((state) => state.chats.chatList);
  const dispatch = useDispatch();

  const handleClose = () => setVisible(false);
  const handleOpen = () => setVisible(true);
  const handleChange = (e) => setNewChatName(e.target.value);
```

```
  const onAddChat = () => {
    dispatch(addChat(newChatName));
    setNewChatName("");
    handleClose();
  };

  return (
    <>
      <div>
        {Object.keys(chats).map((id, i) => (
          <div key={i}>
            <Link to={`/chats/${id}`}>
              <b style={{ color: id === chatId ? "#000000" : "grey" }}>
                {chats[id].name}
              </b>
            </Link>
          </div>
        ))}
        <span className="add-chat" onClick={handleOpen}>
          Add Chat
        </span>
      </div>
      <Dialog open={visible} onClose={handleClose}>
        <DialogTitle>Please enter a name for new chat</DialogTitle>
        <TextField value={newChatName} onChange={handleChange} />
        <Button onClick={onAddChat} disabled={!newChatName}>
          Submit
        </Button>
      </Dialog>
    </>
  );
};
```

Здесь мы добавили два поля стейта - одно для отображения или скрытия диалогового окна (visible, передаем в проп open компонента DIalog), а второй - для хранения данных из TextField. Кроме того, список чатов теперь получаем не из пропсов, а через селектор. При нажатии на кнопку диспатчим экшен для добавления чата в стор, закрываем диалог, и очищаем поле ввода.

Заметьте, что данные, необходимые только внутри одного компонента (к примеру, данные в поле ввода компонента Input) по-прежнему хранятся в стейте. Эти данные не нужны другим компонентам, и они логически связаны только с тем компонентом, в котором хранятся, а потому переносить их в глобальное хранилище не имеет смысла.

Взглянем еще раз схему работы redux:



Заметьте, что экшен может проходит через несколько редьюсеров последовательно.

connect

Для доступа к redux помимо хуков useSelector и useDispatch можно использовать другой подход, основанный на HOC. Для этого react-redux предоставляет специальную функцию connect. Ее использование выглядит следующим образом (на примере ChatList):

```
const mapStateToProps = state => ({
  chats: state.chats.chatList,
});

const mapDispatchToProps = {
  addNewChat: addChat
}

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

Разберем происходящее в этом примере. Connect - функция высшего порядка, она принимает до 4-х аргументов и возвращает другую функцию. Эта возвращенная функция является HOC - она принимает компонент и возвращает другой компонент.

Несмотря на то, что connect может принимать 4 аргумента, все они являются необязательными. В большинстве случаев используются 1 или 2 аргумента. Как правило, их называют mapStateToProps и mapDispatchToProps. С помощью этих аргументов мы определяем, какие именно данные получит наш компонент из стора (mapStateToProps - аналог useSelector), а также какие экшены он сможет диспатчить (mapDispatchToProps - в некоторой степени аналог useDispatch).

mapStateToProps - функция. Она принимает аргументом state - данные стора - и возвращает объект. Ключи этого объекта станут названием пропсов в нашем (обернутом в connect) компоненте, а значения - соответственно, значениями этих пропсов. В примере выше mapStateToProps возвращает объект с ключом name и значением state.profile.name. значит, в нашем компоненте появится проп name, значением которого будет значение поля profile.name из стора. Причем, как и useSelector, connect обеспечивает автоматическое обновление компонента при обновлении данных в сторе.

Внимание!

MapStateToProps принимает второй аргумент, обычно обозначаемый ownProps - собственные пропсы - в котором содержатся пропсы обернутого компонента, переданные от родителя (или из других HOC).

В качестве mapDispatchToProps можно передавать как функцию, возвращающую объект (аргументом она принимает функцию dispatch), так и объект. Кроме того, ее можно создавать с помощью вспомогательной функции из redux **(не react-redux)** - bindActionCreators.

```
// вариант 1
const mapDispatchToProps = {
  addNewChat: addChat,
};
// вариант 2
const mapDispatchToProps = (dispatch) => ({
  addNewChat: (name) => addChat(name),
});

// вариант 3
const mapDispatchToProps = (dispatch) =>
  bindActionCreators(
    {
      addChat,
    },
    dispatch
  );
```

Такую запись можно встретить в легаси коде, однако документация не рекомендует использовать bindActionCreators вместе с react-redux.

Аналогично mapStateToProps, ключи объекта mapDispatchToProps (или объекта, возвращаемого функцией mapDispattchToProps) становятся именами пропсов в компоненте, а значения этих ключей - значениями соответствующих пропсов. Передав в connect такой mapDispatchToProps, мы можем в компоненте вызывать функцию addNewChat:

```
const ChatList = ({ chatId, addNewChat, chats }) => {
  const [visible, setVisible] = useState(false);
```

```
  const [newChatName, setNewChatName] = useState("");

  const handleClose = () => setVisible(false);
  const handleOpen = () => setVisible(true);
  const handleChange = (e) => setNewChatName(e.target.value);
  const onAddChat = () => {
    addNewChat(newChatName);
    setNewChatName("");
    handleClose();
  };

  return (
     // ...
  );
}
```

Передачу данных из стейта через connect чаще всего используют в классовых компонентах, т.к. в них нельзя пользоваться хуками, однако встречаются и случаи использования с функциональными. Вместе с тем, официальная документация react-redux указывает, что рекомендованным “дефолтным” подходом к использованию библиотеки является использование хуков.

Подробнее о useSelector

Оптимизация использования useSelector

Как уже было сказано, и useSelector, и connect отвечают за то, чтобы обновить компонент, когда обновляются данные в сторе. Для обеспечения такого поведения проводится сравнение полученных из стора данных - по умолчанию строгое ссылочное сравнение (prev === next). Это может приводить к лишним обновлениям компонента, особенно в случае, когда из селектора возвращается не примитив.

Одним из способов решения этой проблемы является использование второго аргумента useSelector. Этим аргументом данный хук принимает функцию, которая будет использоваться для сравнения старого и нового значения - в случае равенства этих значений обновление компонента вызвано не будет.

```
  const chats = useSelector(
    (state) => state.chats.chatList,
    (prev, next) => prev.length === next.length
);
```

react-redux также предоставляет функцию для поверхностного сравнения двух значений:

```
import { shallowEqual } from "react-redux";
// ...

const chats = useSelector((state) => state.chats.chatList, shallowEqual);
```

Подробнее о shallow equality можно прочитать здесь.

Другим способом решения указанной проблемы является использование отдельных библиотек, таких как reselect, для организации мемоизации данных, возвращаемых useSelector.

Селекторы

До сих пор в качестве селектора (т.е. первого аргумента useSelector) использовалась стрелочная функция, объявленная в самом компоненте. Такой подход имеет два важных минуса:

1. Повторение кода (селектор для имени профиля используется дважды в разных компонентах, и каждый раз создается новая стрелочная функция)
2. react-redux не может закэшировать результат вызова селектора, т.к. функция-селектор каждый раз создается заново

Исправим это, просто вынеся селекторы в отдельные функции:

```
export function getChatList(state) {
  return state.chats.chatList
}
```

Поместим их в файлы selectors.js в папке store - для каждого редьюсера будет свой файл с селекторами.

Импортируем их в компоненты и используем в useSelector:

```
import { getChatList } from "../../store/chats/selectors";

const chats = useSelector(getChatList, shallowEqual);
```

Или в connect:

```
const mapStateToProps = (state) => ({
  chats: getChatList(state),
});
```

В случае, если селектор зависит от пропсов, можно создать функцию-обертку:

store/chats/selectors.js

```
export function getChatById(chatId) {
  return (state) => state.chats.chatList[chatId]
}
```

MessageField.js

```
const getSelectedChat = useMemo(() => getChatById(chatId), [chatId]);
const selectedChat = useSelector(getSelectedChat);
```

Обратите внимание, что здесь используется useMemo для мемоизации результата вызова функции getChatById. Это помогает избегать ненужных пересозданий функции getSelectedChat.

Глоссарий

1. Селектор - функция, использующаяся для получения некоторой части данных из стора
2. Action creator - “создатель экшена” - функция, возвращающая экшен.
3. Shallow equality comparison - поверхностное сравнение - сравнение, при котором примитивы сравниваются по значению, а объекты по ссылкам.

Домашнее задание

1. Добавить редьюсеры чатов и сообщений. Сообщения хранить в объекте по ключу - id чата.
2. Подключить соответствующие компоненты к стору. Перенести сообщения и чаты из стейта в стор.
3. Перенести (если есть) или добавить логику удаления и добавления чатов в редьюсер.
4. Вынести селекторы в именованные функции, поместить их в соответствующие файлы (store/<reducerName>/selectors.js).

Дополнительные материалы

1. Статья о поверхностном сравнении
2. Статья о reselect

Используемые источники

1. Официальный сайт redux
2. Официальный сайт react-redux
3. Документация Реакт - HOC