нетология

# REDUX, REDUX THUNK

АНТОН СТЕПАНОВ

# АНТОН СТЕПАНОВ

Ведущий фронтэнд разработчик в StepIntegrator

@anton_mesmer

# ПЛАН ЗАНЯТИЯ

1. Асинхронные действия
2. Middleware
3. Redux Thunk

# АСИНХРОННЫЕ ОПЕРАЦИИ

# ЗАДАЧА

Вспомним задачу с прошлой лекции: у нас есть панель управления услугами небольшого сайта по ремонту iPhone, где мы можем редактировать список услуг (для упрощения, мы рассмотрим только просмотр списка и добавление).

Мы начали с обычного CRUD и реализовали хранение всех данных локально.

# HTTP

Это всё хорошо, но данные мы обычно получаем по HTTP. И сохранение/удаление должны не только проходить по HTTP, но после них также должна следовать перезагрузка всего списка.

А, кроме того, нужно ещё отображать индикатор загрузки и отображать ошибки пользователю.

# API

```
npm init
npm install forever koa koa-router koa2-cors koa-body
```

# API

.foreverignore:

```
node_modules
```

scripts в `package.json`:

```
"scripts": {
  "prestart": "npm install",
  "start": "forever server.js",
  "watch": "forever -w server.js"
},
```

# API

```
const http = require('http');
const Koa = require('koa');
const Router = require('koa-router');
const cors = require('koa2-cors');
const koaBody = require('koa-body');

const app = new Koa();
app.use(cors());
app.use(koaBody({json: true}));

let nextId = 1;
const services = [
  { id: nextId++, name: 'Замена стекла', price: 21000, },
  { id: nextId++, name: 'Замена дисплея', price: 25000, },
  { id: nextId++, name: 'Замена аккумулятора', price: 4000, },
  {id: nextId++, name: 'Замена микрофона', price: 2500, },
];
```

```
const router = new Router();
router.get('/api/services', async (ctx, next) => {
  ctx.response.body = services;
});
router.post('/api/services', async (ctx, next) => {
  const id = nextId++;
  services.push({...ctx.request.body, id});
  ctx.response.status = 204;
});
router.delete('/api/services/:id', async (ctx, next) => {
  const id = Number(ctx.params.id);
  const index = services.findIndex(o => o.id === id);
  if (index === -1) {
    ctx.response.status = 404;
    return;
  }
  services.splice(index, 1);
  ctx.response.status = 204;
});

app.use(router.routes());
app.use(router.allowedMethods());

const port = process.env.PORT || 7070;
const server = http.createServer(app.callback());
server.listen(port);
```

# STATE

Давайте сначала поговорим о `state`.

Если отображение загрузки и ошибок должно быть не общим (делать один большой лоадер на всё приложение не очень хорошая идея), то нужно будет сделать флаги `loading`, `error` и для списка, и для формы:

```
{
  serviceList: {
    items: [], loading: false, error: null,
  },
  serviceAdd: {
    item: { name: '', price: '', }, loading: false, error: null,
  }
}
```

В соответствии с этим придётся поправить `initialState` у каждого `Reducer`'а, маппинги в компонентах.

# КОМПОНЕНТЫ

```
// ServiceAdd:
const {item, loading, error} = useSelector(state => state.serviceAdd);
// ServiceList
const {items, loading, error} = useSelector(state => state.serviceList);
// ServiceAddClassBased
ServiceAddClassBased.propTypes = {
  item: PropTypes.shape({
    name: PropTypes.string,
    price: PropTypes.string,
  }).isRequired,
  loading: PropTypes.bool.isRequired,
  error: PropTypes.object,
  onSave: PropTypes.func.isRequired,
  onChange: PropTypes.func.isRequired,
}

const mapStateToProps = (state, ownProps) => {
  const { serviceAdd: {item, loading, error} } = state;
  return { item, loading, error };
}
```

# ACTION TYPES

Именовать `Action`'ы при загрузке можно по следующей схеме:

```
export const FETCH_SERVICES_REQUEST = 'FETCH_SERVICES_REQUEST';
export const FETCH_SERVICES_FAILURE = 'FETCH_SERVICES_FAILURE';
export const FETCH_SERVICES_SUCCESS = 'FETCH_SERVICES_SUCCESS';
export const ADD_SERVICE_REQUEST    = 'ADD_SERVICE_REQUEST';
export const ADD_SERVICE_FAILURE    = 'ADD_SERVICE_FAILURE';
export const ADD_SERVICE_SUCCESS    = 'ADD_SERVICE_SUCCESS';
// специально оставили синхронным
export const REMOVE_SERVICE         = 'REMOVE_SERVICE';
export const CHANGE_SERVICE_FIELD   = 'CHANGE_SERVICE_FIELD';
```

# ACTION CREATORS

Тогда `Action Creator`'ы:

```
export const fetchServicesRequest = () => ({
  {type: FETCH_SERVICES_REQUEST};
});

export const fetchServicesFailure = message => ({
  {type: FETCH_SERVICES_FAILURE, payload: {message}};
});

export const fetchServicesSuccess = items => ({
  {type: FETCH_SERVICES_SUCCESS, payload: {items}};
});

// далее - по аналогии (кроме Remove и Change)
export const changeServiceField = (name, value) => ({
  type: CHANGE_SERVICE_FIELD, payload: { name, value, },
});

export const removeService = id => ({
  type: REMOVE_SERVICE, payload: {id}
});
```

# SERVICELISTREDUCER

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

export default function serviceListReducer(state = initialState, action) {
  switch (action.type) {
    case FETCH_SERVICES_REQUEST:
      return {...state, loading: true, error: null};
    case FETCH_SERVICES_FAILURE:
      const {error} = action.payload;
      return {...state, loading: false, error};
    case FETCH_SERVICES_SUCCESS:
      const {items} = action.payload;
      return {...state, items, loading: false, error: null};
    // на сервере ничего не удаляем
    case REMOVE_SERVICE:
      const {id} = action.payload;
      return {...state, items: state.items.filter(o => o.id !== id)};
    default:
      return state;
  }
}
```

# SERVICEADDREDUCER

```
const initialState = {
  item: { name: '', price: '', },
  loading: false,
  error: null,
};

export default function serviceAddReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_SERVICE_REQUEST:
      return { ...state, loading: true, error: null };
    case ADD_SERVICE_FAILURE:
      const {error} = action.payload;
      return { ...state, loading: false, error };
    case ADD_SERVICE_SUCCESS:
      return { ...initialState };
    case CHANGE_SERVICE_FIELD:
      const { name, value } = action.payload;
      const { item } = state;
      return { ...state, item: {...item, [name]: value }};
    default:
      return state;
  }
}
```

# ГДЕ ДЕЛАТЬ FETCH?

Остался самый главный вопрос - где мы должны делать `fetch` и `async/await`?

В `Reducer`'е нельзя, т.к. это чистая функция.

Можно, конечно, написать эту логику в каждом компоненте и из компонентов `dispatch`'ить `Action`'ы.

Это вполне рабочий механизм, но если мы закладываемся на то, что эту логику будет использовать ещё кто-то, почему бы не вынести её в отдельную функцию?

Но тогда этой функции нужен будет `dispatch`.

# ГДЕ ДЕЛАТЬ FETCH?

```
// actionCreators.js
export const fetchServices = async dispatch => {
  dispatch(fetchServicesRequest());
  try {
    const response = await fetch(`${process.env.REACT_APP_API_URL}`)
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    const data = await response.json();
    console.log(data);
    dispatch(fetchServicesSuccess(data));
  } catch (e) {
    dispatch(fetchServicesFailure(e.message));
  }
}
```

# ГДЕ ДЕЛАТЬ FETCH?

```
// actionCreators.js
export const addService = async (dispatch, name, price) => {
  dispatch(addServiceRequest());
  try {
    const response = await fetch(`${process.env.REACT_APP_API_URL}`, {
      method: 'POST',
      headers: {'Content-Type': 'application/json'},
      body: JSON.stringify({name, price}),
    })
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    dispatch(addServiceSuccess());
  } catch (e) {
    dispatch(addServiceFailure(e.message));
  }
  fetchServices(dispatch);
}
```

# FUNCTIONAL COMPONENT

```
function ServiceList(props) {
  const {items, loading, error} = useSelector(state => state.serviceList);
  const dispatch = useDispatch();

  useEffect(() => {
    fetchServices(dispatch)
  }, [dispatch])

  const handleRemove = id => {
    dispatch(removeService(id));
  }

  return (...)
}
```

Работает-то, оно, конечно, работает, но хотелось бы, чтобы вызовы выглядели одинаково.

Можно, конечно, переделать все вызовы из формата `dispatch(actionCreator())` в `actionCreator(dispatch)`, но это не лучшая идея.

# CLASS BASED COMPONENT

```
class ServiceListClassBased extends Component {
  componentDidMount = () => {
    this.props.fetchServices();
  }
  handleRemove = id => {
    this.props.removeService(id);
  }
  render() {
    const {items, loading, error} = this.props;
    return (...)
  }
}
const mapStateToProps = (state, ownProps) => {
  const {serviceList: {items, loading, error}} = state;
  return {items, loading, error};
};
const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    fetchServices: () => fetchServices(dispatch),
    removeService: id => dispatch(removeService(id))
  }
};
export default connect(mapStateToProps, mapDispatchToProps)(ServiceListClassBased);
```

# MIDDLEWARE

# MIDDLEWARE

Так же, как и во многих других фреймворках/библиотеках, Redux поддерживает концепцию Middleware - промежуточного ПО, позволяющего вклиниться в определённый момент обработки.

В случае Redux - это момент между отправкой `Action`'а и его попаданием в `Reducer`.

# MIDDLEWARE

Вспомним нашу проблему - мы можем отправлять только обычные `Action`'ы, а хотелось бы иметь возможность отправлять функции, как `Action`'ы.

Благодаря концепции Middleware мы можем написать функцию, которая бы принимала на вход наши функции и выполняла бы их.

# MIDDLEWARE

Эта идея настолько не нова, что уже есть готовые решения, чем мы и воспользуемся.

# REDUX THUNK

# REDUX THUNK

Redux Thunk - Middleware для Redux, который расширяет возможности `Store`, позволяя диспатчить функции (в том числе с асинхронными запросами).

```
npm install redux-thunk
```

# REDUX THUNK

```
// store/index.js
import { createStore, combineReducers, applyMiddleware } from "redux";
import serviceListReducer from '../reducers/serviceList';
import serviceAddReducer from '../reducers/serviceAdd';
import thunk from "redux-thunk";

const reducer = combineReducers({
  serviceList: serviceListReducer,
  serviceAdd: serviceAddReducer,
});

const store = createStore(reducer, applyMiddleware(thunk));

export default store;
```

# REDUX THUNK

Исходный код Redux Thunk:

```
function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => next => action => {
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }

    return next(action);
  };
}

const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;

export default thunk;
```

Т.е. с обычными `Action`'ами делать ничего не нужно.

https://github.com/reduxjs/redux-thunk/blob/master/src/index.js

# REDUX THUNK

Переделаем наши функции, чтобы они подходили для этого Middleware:

```
// общая схема:
const func => (наши аргументы) => (dispatch, getState) => {... наш код...}
// после чего можем делать:
disptach(func(args));
```

Redux Thunk будет выполнять код нашей функции, в котором может быть (а может и не быть - в зависимости от условий), `dispatch` `Action`'ов.

Аргумент `getState` позволяет получить доступ к текущему состоянию (мы говорили о этом, когда рассматривали пример с добавлением).

# REDUX THUNK

```
function ServiceList(props) {
  ...

  useEffect(() => {
    // было:
    // fetchServices(dispatch)
    // стало:
    dispatch(fetchServices())
  }, [dispatch])

  ...
}
```

# REDUX THUNK

```
class ServiceListClassBased extends Component {
  ...
}

ServiceListClassBased.propTypes = {
  ...
}

const mapStateToProps = (state, ownProps) => {
  ...
};

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    // было:
    // fetchServices: () => fetchServices(dispatch),
    // стало:
    fetchServices: () => dispatch(fetchServices()),
    removeService: id => dispatch(removeService(id)),
  }
};
```

# REDUX DEVTOOLS

```
import { createStore, combineReducers, applyMiddleware, compose } from 'redux';
import serviceListReducer from '../reducers/serviceList';
import serviceAddReducer from '../reducers/serviceAdd';
import thunk from 'redux-thunk';

const reducer = combineReducers({
  serviceList: serviceListReducer,
  serviceAdd: serviceAddReducer,
});

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const store = createStore(reducer, composeEnhancers(applyMiddleware(thunk)));

export default store;
```

# ИТОГИ

Сегодня мы рассмотрели достаточно сложную тему: использования побочных эффектов с Redux.

Итоговые исходники к материалам сегодняшней лекции будут размещены в репозитории с кодом к лекциям.

нетология

Задавайте вопросы и напишите отзыв о лекции!

# АНТОН СТЕПАНОВ

@anton_mesmer