

4. Продвинутые концепты Redux

- Обзор
- Концепции функционального программирования
- Продвинутая иммутабельность
- Асинхронность в Redux
- Формы нормализации состояния
- Селекторы
- Итоги

Обзор

Здравствуйте, и добро пожаловать 🙌! Я надеюсь, что все предыдущие части Redux вас не напугали. Однако не волнуйтесь, даже если вы испуганы – цель нашего урока копнуть глубже в эти функциональные штуки, на которые опирается Redux для упрощения их понимания 🙌.

Помимо этого, мы рассмотрим:

- Какие преимущества обеспечивают immutable использование данных, И как их использовать более эффективным способом;
- Как управлять одной из ведущих характеристик языка JavaScript в рамках Redux;
- Как превратить de-normalized данные в normalized и как сохранить state опрятным, хорошо структурированными в легкой для работы форме;
- Как написать и использовать специальные функции selectors – которые сделают нашу работу с state более легкой и веселой.

Давайте, запрыгивайте 🙌!

Концепции функционального программирования

Redux берет вдохновение из концепций функционального программирования. К примеру, Elm – является языком функционального программирования который использует Дан Абрамов. Для обновления данных Elm использует следующую подпись:

```
(state, action) => state;
```

Это же вам знакомо? Конечно, это ведь синтаксис функции Redux reducer 🙌!

👉 Совет:

Мы советуем прочитать некоторую дополнительную информацию про `Elm` и другие функциональные языки программирования. Это поможет вам понять идею спрятанную за `functional` подходом и работать с `functional-inspired` технологиями с более высокой эффективностью 🧐.

Давайте поразмышляем над следующим примером:

```
const square = operand => operand * operand;
```

Эти функции берут `один входящий` и создают `один исходящий`. Не имеет значения как много раз вы вызываете эти функции с тем же аргументом, вы будете получать тот же результат. Технически мы можем рассмотреть такую характеристику как хорошую, потому что она наделяет нашу программу бесценным значением – `предсказуемость` 🧐.

Однако есть и другая сторона монеты: иногда функция может взять больше, чем `один входящий`, и произвести больше, чем `один исходящий` 😞.

Ты 🐱:

– Пойдите, функция не может иметь больше чем `один` ввод и вывод. Функции получают ввод как `аргументы` и производят вывод с помощью ключевого слова `return`.

Автор 🧑:

– Да, так и происходит, но входящими параметрами для функции могут служить не только `arguments`, и вывод может быть произведен не только с помощью ключевого слова `return`. `Оскар` не был бы ты так любезен помочь нам разъяснить этот вопрос 🙋?

`Оскар` 🧑:

– С удовольствием! Давайте зажигать 🔫!

Рассмотрим следующее:

```
const process = () => {
  const message = messages.pop();

  if (!message) {
    process(message)
  }
};
```

Функция `process()` не принимает никаких **входящих аргументов** и не производит никаких **возвращенных значений**. Тем не менее, она зависит от **чего-то**, что имеет **некоторое состояние** и делает **что-то** 🤔. Вы это чувствуете? Уже **непонятно**, что происходит в такой простой функции.

Нам **непонятна** входящее состояние структуры данных `messages` перед вызовом `pop()` и скрытые выводы: будет ли запускаться `process(message)` или нет (мы даже не знаем что делает `process()`), и состояние `messages` после того как функция закончит своё выполнение.

Поведение функции `process()` нельзя предсказать без знания `value`, `state` и `behavior` чужеродных сущностей внутри тела функции `process()`. Функция `process()` берет **непонятный** ввод, и создает **непонятный** вывод. Оно требует **чего-то**, и вызывает **что-то**, но вы никогда не сможете догадаться что оно делает, просто смотря на API.

Скрытые вводы и выводы называются **side-effects**. **side-effect** – это то что приносит **неясность** или **непредсказуемость** в поведение функции.

Это приводит нас к важной **концепции функционального программирования** – **pure functions** 💡.

🔍 Заметка:

Pure functions так же иногда может упоминаться как **idempotent functions**.

Почему **pure**? А **pure function** всегда производит тот же результат, получая тот же аргумент и не производит **side effects**. Просто **pure**.

И напротив, если функция не соответствует данным критериям – она является **impure** функцией.

❗ Важно:

Не забудьте, что **reducers** в **Redux** – являются **pure functions**! **reducers** берет текущее **state** и **action** как аргументы и возвращает новое **state**. **Reducer** не должен производить никаких **side effects**.

Functional programming — полностью про написание **pure functions**, про удаление спрятанных вводов и выводов на столько, на сколько мы можем, так что бы как можно больше нашего кода описывало взаимоотношения между вводами и выводами.

👉 Совет:

В реальном мире мы принимаем то, что некоторые **side effects** неизбежны - большинство программ заботятся о том что они делают, а не о том что они возвращают, но мы должны каждый раз осуществлять жесткий контроль в программах. Нам так же необходимо устранить **side effects** как только будет возможность, и плотно контролировать их даже когда мы не можем.

В `математике` существует термин, который описывает `шаблон` когда функция возвращает другую функцию. Тот же шаблон считает, что функция `может` может взять другие функции как параметры. Этот `шаблон` композиции функций называется композицией `higher-order function` 🧠.

Так что `higher-order function` – это функция, которая возвращает функции и/или берет функцию как параметр.

`Functional programming` на самом деле использует подход `higher-order function` из-за его продвинутой способности составлять композиции из `pure functions`.

Перед этим мы говорили про `higher-order components` в рамках `React` и `Redux`. Идеи очень похожи. `Higher-order component` – функция, которая берёт компоненты `React` и неким образом улучшает их, а потом возвращает новым, расширенным компонентом. Это работает почти так же как и техника `higher-order function`. Единственное отличие в том, что функция возвращается вместо компонента `React` в сценарии `higher-order function` 📦.

Очень важно понять оба подхода. Таким образом вы должны их лучше узнать, и в результате стать еще лучшими экспертами 👑.

В предыдущем уроке мы так же коротко говорили про последнюю функцию из главного API `Redux` – функцию `compose()`. Так как мы погружаемся глубже в `концепции функционального программирования`, это подходящее время, чтобы разобраться с механизмом функции `compose()` 🧠.

Функция `compose()` – это `functional programming` функция, и может использоваться в вашей программе для удобства.

Функция `compose(functions...)` `компонует` `функции`, передаёт аргументы, `справа` на `лево`. Каждая `function` ожидает получения единичного параметра. Её возвращаемое значение будет предоставлено в качестве аргумента функции, стоящей с лева и так далее. Исключение самый правый аргумент, который может принять множественные параметры, поскольку это обеспечит сигнатуру для `composed function`.

К примеру, есть так же второй вариант внедрения `Redux DevTools enhancer` в ваше приложение с помощью `npm package` вместо `Google Chrome extension`:

```
// app/store/index.js

import { createStore, applyMiddleware, compose } from 'redux';
import DevTools from '../containers/DevTools';
import reducer from '../reducers';

const logger = (store) => (next) => (action) => {
  console.log('Previous state:', store.getState());
  next(action);
}
```

```
    console.log('Action:', action);
    console.log('Next state:', store.getState());
  };

  export default createStore(
    reducer,
    compose(
      applyMiddleware(logger),
      DevTools.instrument(),
    ),
  );
```

Более того, существует еще одна важная концепция, `functional programming` опирается на `immutability`.

`Immutability` – не изменяемость с течением времени 🗝️.

Однако, что означает `immutability` в программировании? Давайте возьмём `не изменяемый` объект. `Immutable` — это объект, который не может быть изменен после его создания. И наоборот, объект `mutable` — любой объект, который может быть изменен после его создания.


`Immutability` – является так же фундаментальной концепцией архитектуры `Redux`. Вы возможно интересуетесь как можно построить приложение, которое не изменяет свое состояние 🤔. Если я не могу изменить состояние, не значит ли это, что нельзя изменить данные 🤔? Вовсе нет. Это всего лишь значит, что вместо того чтобы непосредственно изменять состояние объекта, вы должны создать и вернуть новый объект, который предоставляет новое состояние.

Неизменные сущности JavaScript 😊	Переменные сущности JavaScript 🤔
Числа	Объекты
Строки	Массивы
Булевы значения	Функции
undefined	
null	

Каждый раз когда вы меняете значение типа `immutable`, создается `новая` копия. Каждый раз когда вы меняете свойства объекта или меняете значение индекса массива – вы `мутируете` его.

Без `immutability` потока данных в вашей программе, получают потери. История состояний становится заброшенной, и странные ошибки могут проникнуть в ваше программное обеспечение.


Давайте посмотрим на пример обновления состояния в традиционном приложении:

 Плохое применение (`mutation`):

```
const state = {
  name: 'Oscar Egilsson',
  role: 'author'
};



state.role = 'wizard';

return state;
```

 Хорошее применение (`immutability`):

```
const state = {
  name: 'Oscar Egilsson',
  role: 'author'
};


return {
  name: 'Oscar Egilsson',
  role: 'wizard'
}
```

В первом, плохом  примере мы просто `мутируем` состояние объекта путем присвоения нового значения свойства `role`. В сравнении со вторым, хорошим примером , мы вернули целиком новый объект `без мутации` 😊.

Вот другой пример `мутации` и другой пример сохранения `не изменности`:

 Плохое применение (`мутация`):

```
Object.assign(state, ...sources);
```

 Хорошее применение (`не изменности`):

```
Object.assign({}, state, ...sources);
```

В первом, плохом ❌ примере, происходит мутация объекта `state`, получением всех свойств объектов `sources`. В сравнении со вторым, хорошим примером ✅, новый пустой объект создается, и этот новый объект получает свойства объекта `state` и объектов `sources`, но главное здесь то, что объект `state` остается нетронутым.

Таким образом, практический результат такой хитрости – клонирование существующих объектов `state`, но со всей новой информацией из `sources`.

И вот еще другой пример мутации:

❌ Плохое применение (мутации):

```
const books = [{ /* book 1 */ }, { /* book 2 ... */ }];

books.push(newBook);

return books;
```

✅ Хорошее применение (не изменяемости):

```
const books = [{ /* book 1 */ }, { /* book 2 ... */ }];

const newBooks = [ ...books, newBook ];

return newBooks;
```

В первом, плохом ❌ примере, массив `books` изменился с помощью `push` новой книги. В сравнении, со вторым, хорошим примером ✅, массив `newBooks` создается используя оператор JavaScript `spread` чтобы сохранить предыдущий массив `books`, и добавить объект `newBook`.

! Важно:

Redux использует не изменяемость состояния для повышения производительности, поэтому крайне важно следовать принципу неизменности в процессе создания приложения Redux 🚔.

Продвинутая иммутабельность

Существует множество чудесных инструментов, которые упрощают процессы манипуляции `immutable` данными. Однако, если вы меня спросите о каком-либо – который будет для вас наилучшим, без сомнений – библиотека [Immerable.js](https://github.com/immerjs/immer) 📖.

`Immutable.js` была спроектирована и разработана компанией Facebook для решения проблем с `immutability`, свойственным JavaScript, обеспечивая все выгоды от `immutability` для повышения производительности, которую требует ваше приложение. Эта библиотека хорошо поддерживается, обеспечивая богатые API с широким набором `immutable` структур данных, структурированных как `Maps`, `Lists`, `Sets`, `Records` и многие другие 🍷.

`Immutable.js` `Map` к примеру – неупорядоченная коллекция таких пар как `key/value`.

Здесь представлен пример использования `Map` данных, структурированных в редьюсере `user` из приложения `Оскара` `book-reader`:

```
// app/reducers/user/index.js

import { UPDATE_USER, UPDATE_FIRST_NAME } from '../actions/profile/types';
import { Map } from 'immutable';

const initialState = Map({
  firstName: 'Oscar',
  lastName: 'Egilsson'
});

export default (state = initialState, { type, payload }) => {
  const { firstName, lastName } = payload;

  switch (type) {
    case UPDATE_USER:
      return state.merge({
        firstName,
        lastName
      });

    case UPDATE_FIRST_NAME:
      return state.set('firstName', firstName);

    default:
      return state;
  }
};
```

`merge()` метод структуры данных `Map` возвращает новую структуру `Map`, результат от слияния предоставленных сущностей данных в предыдущий `Map`.

Метод `set()` возвращает новый `Map`, а так же содержит новые пары `key/value`. Если эквивалент `key` уже существует в данной `Map`, он будет заменен.

Как вы видите, API довольно-таки интуитивен и удобен в использовании.

Давайте рассмотрим пример `mapStateToProps()` в компоненте `Profile`, в котором мы будем извлекать данные из `Immutable Map`:

```
// app/components/Profile/index.js

// ...the rest of the Profile implementation...

const mapStateToProps = ({ user }) => ({
  firstName: user.get('firstName'),
  lastName: user.get('lastName')
});

// ...the rest of the Profile implementation...
```

Да, это настолько просто. Как мы могли ожидать, метод `get()` возвращает `value` ассоциируя его с предоставленным `key`.

`Immutable.js` так же делает много работы за кулисами чтобы оптимизировать производительность. Это ключ к его силе, использование `immutable` структур данных может вовлечь множество дорогостоящих копирований. В частности, `immutablely` манипулирование, сложными наборами данных, такими как вложенные `Redux state` деревья, могут сгенерировать множество посредственных копий объектов, которые потребляют память и замедляют производительность так как сборщик мусора браузера, спешит чтобы все вычистить 🗑️.

👉 Совет:

`Immutable.js` высоко производительная `immutability-helper`, удобно сделанная с ❤️ инженерами Facebook. Имеет простой API, и очень легкая! Мы настойчиво рекомендуем, использовать её как часть вашего проекта 👍!

`Immutable.js` избегает этого умным образом распределяя структуры данных под поверхностью, тем самым минимизируя необходимость копирования данных. Она так же позволяет создавать сложные цепи операций без создания ненужных операций (и дорогих), клонирование промежуточных данных, которые будут быстро выброшены 🙌.

Вы никогда не увидите этого, конечно - данные, которые вы передали объекту `Immutable.js` никогда не мутируют. Скорее всего, это промежуточные данные сгенерированные внутри `Immutable.js` из цепи последовательных вызовов методов, которые свободно могут быть подвержены мутации. Таким образом, вы получаете все преимущества неизменяемых структур данных без каких-либо (или очень маленьких) потерь производительности.

Вы так же можете рассмотреть аккуратное [расширение](#) `Google Chrome Web Store`, которое отформатирует `Immutable.js` структуры, которые вы передадите в конструкцию `console.log()` в красивом и легком для прочтения способе (не забывайте следовать четким инструкциям) 🙌.

Асинхронность в Redux

По дефолту `Redux` выполняет свой поток в `синхронном` режиме. Это означает, что `действия` являются `синхронными` и должны вернуть объект. Более того, только после того, как `действие` будет `запущено`, оно может быть передано к `reducers` что бы выполниться `синхронно`. Целый поток является `синхронным`.

Так что вопрос становится таким: как мне сделать `асинхронный` вызовы в `Redux` ?

😂 Шутка:

Эй, мы в стране JavaScript, так что возможно вы знаете ответ на вопрос: существует множество библиотек, чтобы управлять `асинхронностью` в `Redux` 🐱!

Главные игроки управления логикой `асинхронностью` в `Redux` являются:

- [redux-thunk](#)
- [redux-promise](#)
- [redux-saga](#)

`redux-thunk` достаточно популярен и был написан Деном Абрамовым. Это позволяет вам вернуть `функции` из вашего `создателя действия`, вместо `объектов`.

`redux-promise` это альтернативная библиотека, которая использует `FSA` для того, чтобы внести некоторые четкие конвенции в `асинхронные` вызовы. Она возвращает `promise`, который будет `запущен` что бы получить значения промиса. Ничего не будет `запущено`, в том случае если промис будет отвергнут. Это действительно популярный вариант для управления `асинхронным` потоком в `Redux`.

`redux-saga` использует совсем другой подход: она использует `функции генераторы` и предлагает впечатляющее количество возможностей справиться с `асинхронностью` используя богатый набор встроенных помощников в соединении с ключевым словом `yield`.

Мы покроем `redux-thunk` и `redux-saga` в этом уроке так как эти два наиболее эффективных варианта. Хотя, `redux-promise` тоже следует попробовать, однако, он не так популярен как `redux-thunk` или `redux-saga` 🌟.

Но все по порядку. Что такое `thunk`? `thunk` – является подпрограммой, используемой для введения дополнительного вычисления в другую подпрограмму.

Вы так же можете подумать про `thunk` как про функцию, которая оборачивает выражение, чтобы отложить его вычисление.

Учитывая следующее:

```
const result = 1 + 2; // <= not a thunk — immediate calculation
const thunk1 = () => 1 + 2; // <= 1-stage thunk
const thunk2 = a => b => (1 + 2) + (a + b); // <= 2-stage thunk
```

Результат выражения `1 + 2` привязывается к идентификатору `немедленно result`. Сравните откладывание выполнения выражения `1 + 2` в `thunk1`, с тем как еще больше откладывается `thunk2` 🚗!

Так как `reducers` является `idempotent`, `асинхронные` запросы должны сообщать `store`, когда запросы начинаются, и уведомить снова, когда они закончатся и только после этого вызов `dispatch` должен передать выполнение к `reducer` с чистыми данными, для продолжения синхронных вычислений.

Для использования `redux-thunk` в приложении `Redux`, нам нужно ввести его как `middleware`:

```
// app/store/index.js

import { createStore, applyMiddleware, compose } from 'redux';
import thunk from 'redux-thunk';
import reducer from '../reducers';

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ||
compose;

export default createStore(
  reducer,
  composeEnhancers(applyMiddleware(thunk))
);
```

Давайте создадим `thunk action`:

```
// app/actions/profile/types.js

export const START_FETCHING_BOOKS = 'START_FETCHING_BOOKS';
export const END_FETCHING_BOOKS = 'END_FETCHING_BOOKS';
export const GET_FAVORITE_BOOKS = 'GET_FAVORITE_BOOKS';
```

```
// app/actions/profile/index.js
```

```
import {
  START_FETCHING_BOOKS,
  END_FETCHING_BOOKS,
  GET_FAVORITE_BOOKS
} from './types';

export const getFavoriteBooks = () => async (dispatch) => {
  dispatch({ type: START_FETCHING_BOOKS });
  const result = await fetch(
    `https://data.book-reader.io/users/Oscar/favorite-books`
  );

  const favoriteBooks = await result.json();

  dispatch({
    type: GET_FAVORITE_BOOKS,
    payload: favoriteBooks
  });
  dispatch({ type: END_FETCHING_BOOKS });
};
```

Посмотрите ка поближе 🔍 на `getFavoriteBooks` `thunk action` – вы заметили несколько вызовов `dispatch`? Это все про `redux-thunk`.

На линии кода 9 функция `async (dispatch)` возвращается вместо обычного объекта действия. Мы уже ввели `redux-thunk middleware` так что приложение уже знает этот прием. Возвращая функцию вместо объекта действия из создателя действия, мы инструктируем хранилище, о необходимости управления этим действием как `thunk action`.

Теперь у нас есть возможность сделать несколько вызовов `dispatch` из `thunk-action`. В нашем примере мы запускаем три обычных действия:

- `START_FETCHING_BOOKS` – дает понять нашему приложению, что `fetch` запрос начинается
- `GET_FAVORITE_BOOKS` – чтобы обновить `state` с полученной избранной книгой
- `END_FETCHING_BOOKS` – дает понять нашему приложению, что `fetch` запрос завершился

Таким образом, если асинхронный запрос займет пару секунд, мы можем отобразить запрос на UI в форме спинера, к примеру 🖥️.

Давайте рассмотрим реализацию обновленного `user reducer`:

```
// app/reducers/user/index.js
```

```

import {
  UPDATE_USER,
  START_FETCHING_BOOKS,
  GET_FAVORITE_BOOKS,
  END_FETCHING_BOOKS,
} from '../actions/profile/types';
import { Map } from 'immutable';

const initialState = Map({
  firstName: 'Oscar',
  lastName: 'Egilsson',
  favoriteBooks: [],
  isFavoriteBooksFetching: false
});

export default (state = initialState, { type, payload }) => {
  switch (type) {
    case UPDATE_USER:
      const { firstName, lastName } = payload;

      return state.merge({ firstName, lastName });

    case START_FETCHING_BOOKS:
      return state.set('isFavoriteBooksFetching', false);

    case GET_FAVORITE_BOOKS:
      return state.set('favoriteBooks', payload);

    case END_FETCHING_BOOKS:
      return state.set('isFavoriteBooksFetching', false);

    default:
      return state;
  }
};

```

Теперь мы можем обновлять `состояние` приложения правильно, когда бы запрос избранной книги не `начался`, когда массив избранных книг получен от сервера, и когда запрос на выборку `окончен`.

Мы можем сослаться на `getFavoriteBooks` `thunk action`, чтобы получить желаемый результат:

```
// app/components/Profile/index.js
```

```
import { getFavoriteBooks } from '../actions/profile';

// ...the rest of the Profile implementation...

componentDidMount () {
  this.props.actions.getFavoriteBooks()
}

// ...the rest of the Profile implementation...

const mapDispatchToProps = (dispatch) => ({
  actions: bindActionCreators({ push, getFavoriteBooks }, dispatch)
});

// ...the rest of the Profile implementation...
```

Обрабатывать `асинхронный` запрос с `redux-thunk` весело и радостно. Однако нестоящее "мясо" начнется, когда `redux-saga` выйдет на сцену 🙋.

Принцип `redux-saga` немного отличается. Он рассматривает каждую серию связанных между собой событий как цепочку. И эта цепь имеет свою нить. Это означает, что он может быть концептуализирован отдельно от любых других цепей событий. Для того чтобы это стало возможным, `redux-saga` использует функции генераторы из ES2015 🤖.

Функции генераторы одна из наиболее сложных частей JavaScript, которая делает `redux-saga` немного сложнее в использовании. Она использует ключевое слово `yield`, так что если вы пришли из языка `C#`, у которого есть функциональная `асинхронность`, вы узнаете, что имплементирование происходит почти таким же способом и вам возможно будет удобно использовать `redux-saga` 🧐.

Давайте напишем нашу `асинхронную` реализацию действия `getFavoriteBooks()` используя `redux-saga` 🧑.

Чтобы добавить `redux-saga` в проект, давайте введем его как `middleware`:

```
// app/store/index.js

import { createStore, applyMiddleware, compose } from 'redux';
import createSaga from 'redux-saga';
import reducer from '../reducers';
import { watchGetFavoriteBooks } from '../sagas';

const saga = createSaga();
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ||
compose;
```

```
export default createStore(
  reducer,
  composeEnhancers(applyMiddleware(saga))
);

saga.run(watchGetFavoriteBooks);
```

На этом этапе каждая строка кода должна быть вам знакома, за исключением следующего утверждения:

```
import { watchGetFavoriteBooks } from '../sagas';
```

Прежде чем мы продолжим давайте рассмотрим немного базовых вещей про `redux-saga` и рассмотрим обновления `profile` действий, они скоро нам потребуются:

```
// app/actions/profile/types.js

export const UPDATE_USER = 'UPDATE_USER';
export const START_FETCHING_BOOKS = 'START_FETCHING_BOOKS';
export const END_FETCHING_BOOKS = 'END_FETCHING_BOOKS';
export const GET_FAVORITE_BOOKS = 'GET_FAVORITE_BOOKS';
export const GET_FAVORITE_BOOKS_SUCCESS = 'GET_FAVORITE_BOOKS_SUCCESS';
```

```
// app/actions/profile/index.js

import { GET_FAVORITE_BOOKS } from './types';

export const getFavoriteBooks = () => ({
  type: GET_FAVORITE_BOOKS
});
```

Существует два типа `sagas`:

- Функции генераторы `watcher saga`;
- Функции генераторы `worker saga`.

`watcher saga` – это `saga`, которая отвечает за наблюдение за вызовами `dispatch` конкретных действий и вызывает `worker saga` соответственно. Так что название `watcher saga` говорит само за себя 🤪, так же как и для `worker sagas`.

`worker saga` – это `главная saga`, она выполняет главный поток `асинхронно`, который должен быть выполненным с набором вспомогательных помощников `redux-saga`, таких как `call`, `put`, `takeEvery` и многих других.

Помощник `call()` создает инструкцию для вызова специальной функции с указанными аргументами. Помощник `put()` сообщает `middleware` запустить действие для хранилища. Помощник `takeEvery()` вызывает `saga` на каждое действие, которые запускается к хранилищу.

Трюк с `import { watchGetFavoriteBooks } from '../sagas';` в том что, `redux-saga` довольно сложный инструмент, так что по-хорошему вам нужно будет расширить дополнительные ветки вашего проекта. Мы создадим новую директорию `sagas` под директорией `app`. На данный момент, все `sagas` будут жить здесь:

```
// app/sagas/index.js

import { call, put, takeEvery } from 'redux-saga/effects';
import {
  START_FETCHING_BOOKS,
  END_FETCHING_BOOKS,
  GET_FAVORITE_BOOKS,
  GET_FAVORITE_BOOKS_SUCCESS
} from '../actions/profile/types';

function* getFavoriteBooksWorker () {
  yield put({ type: START_FETCHING_BOOKS });

  const result = yield call(
    fetch,
    'https://data.book-reader.io/users/Oscar/favorite-books'
  );

  const favoriteBooks = yield result.json();

  yield put({
    type: GET_FAVORITE_BOOKS_SUCCESS,
    payload: favoriteBooks
  });

  yield put({ type: END_FETCHING_BOOKS });
}

export function* watchGetFavoriteBooks () {
  yield takeEvery(GET_FAVORITE_BOOKS, getFavoriteBooksWorker);
}
```


Большинство API возвращают глубоко вложенные данные JSON многократно продублированными сущностями. Сохраняя данные в `state` в той же форме получается множество сложнейших дубликатов. С ростом приложения становится сложнее работать с таким, `de-normalized` состояниями.

Обратите внимание на следующие `de-normalized` JSON `books` данные, которые возвращаются к нам из API:

```
const books = [{
  "id": "41",
  "title": "The best about pine trees.",
  "author": {
    "id": "1",
    "name": "Oscar Egilsson"
  }
}, {
  "id": "15",
  "title": "The Thing.",
  "author": {
    "id": "1",
    "name": "Oscar Egilsson"
  }
}
];
```

Вы наверное заметили, что сущность `author` дублируется в обеих книгах. Возможно это не кажется вам проблемой в таком простом примере, но представьте себе что ваш массив состоит из `1.000` книг и в нем каждая книга имеет заполненный дубликат `author` 🤔?

Более того, настоящие ответы API, содержат гораздо более разные сущности, с еще более глубоким вложением. Так что структура данных с `1.000` (или `10.000`) книг, где каждая запись содержит дублированные сущности, настоящая боль 😞.

Ситуация станет еще хуже, если ответы API вернут разные формы с каждым ответом 😬.

Оскар 🧑:

– Не бойся, мой дорогой друг. Де-нормализованная форма `состояния` испариться, как только мы узнаем что такое `normalization` 💪!

`Normalization` – это процесс упрощения `de-normalized` форм `состояния` до оптимизированной структуры данных для сокращения избыточности данных и повышения их целостности 🧑.

Удачно полученная `нормализованная структура данных` оптимальная структура, состоящая из атомарных элементов.

Существует красивый инструмент, который нам поможет `нормализовать` данные – он называется [normalizr](#).

`Normalizr` – это библиотечная утилита, которая помогает нормализовать ответы API, чтобы они имели одинаковую форму.

Вместо того чтобы управлять каждым новым API ответом по-разному, перед обработкой соответствующих данными редюсерами, мы хотим нормализовать ответы с `normalizr`, для того чтобы соответствующая форма была всегда такой же. Когда каждая форма запроса нормализована, вся форма состояния приложения также в результате нормализуется.

Давайте посмотрим как мы можем использовать `normalizr` для `нормализации` API в соответствии с `de-normalized` данными `books`:

```
import { normalize, schema } from 'normalizr';

const books = [{
  "id": "41",
  "title": "The best about pine trees.",
  "author": {
    "id": "1",
    "name": "Oscar Egilsson"
  }
}, {
  "id": "15",
  "title": "The Thing.",
  "author": {
    "id": "1",
    "name": "Oscar Egilsson"
  }
}];

const author = new schema.Entity('authors');
const book = new schema.Entity('books', {
  author
});

const normalized = normalize(books, [book]);
```

`Normalizr` использует `схему` для определения типа каждой сущности.

Функция `normalize()` берет два аргумента:

- `de-normalized data` как первый аргумент;

- И `схему` как второй аргумент.

`normalized` данные будут производиться после успешного вычисления `normalize()`.

Теперь `нормализованный` результат будет выглядеть так:

```
{
  entities: {
    authors: {
      1: {
        id: "1",
        name: "Oscar Egilsson"
      }
    },
    books: {
      15: {
        author: "1",
        id: "15",
        title: "The Thing."
      },
      41: {
        author: "1",
        id: "41",
        title: "The best about pine trees."
      }
    }
  },
  result: ["41", "15"]
};
```

`normalized` вывод состоит из свойств `entities`, которые описывают `данные` и свойств `result`, которые описывают `идентификаторы entities`.

После `нормализации`, данные складываются в эффективную форму, где каждая сущность имеет свою собственную `id` и определяется только один раз. Это позволяет нам `ссылаться` на запрошенные сущности с помощью их `ID`. Легко и красиво.


👉 Совет:

Нормализация данных не является обязательным, хотя это значительно снижает сложность приложения и улучшает опыт разработчиков, а так же, в результате, делает процесс разработки намного более эффективным. И поэтому мы настоятельно рекомендуем рассмотреть этот вариант, особенно потому, что нормализация данных тесно соединенна с запросом состояния, который мы вскоре обсудим.

Вот вам несколько причин, которые направят вас к использованию техники `нормализации данных`:

- Когда фрагмент данных дублируется в нескольких местах, становится все труднее убедиться, что он обновлен соответствующим образом;
- Вложенные данные означают, что соответствующая логика `reducer` должна быть более вложенной и, следовательно, более сложной. В частности, попытка обновить глубоко вложенное поле может стать очень уродливым.
- Так как `immutable` обновления данных требует, чтобы все предки в древе `state` были скопированы и так же обновлены, а обновление новых объектов приведет к повторному отображению соединённых компонентов UI, обновление глубоко вложенного объекта данных может заставить полностью не связанные компоненты пользовательского интерфейса повторно перерисоваться, даже если отображаемые данные фактически не изменились.

Селекторы

Важно отметить, при работе с `Redux` – каждый раз, когда обновляется компонент, `react-redux` вызывает функцию `mapStateToProps()`. Если вы делаете в ней что-то дорогое в плане производительности, вы возможно захотите воспользоваться такой библиотекой как [reselect](#) для `memoizing` .

 Заметка:

`Memoizing` – это отслеживание результатов каждого вызова функции, чтобы функция не запускалась снова, в том случае если она уже была запущена с теми же параметрами. Мы так же можем сказать, что `мемоизация` – это как `кеширование` для вызова функций.

Каждый раз при вызове функций, `reselect` всего лишь проверяет, была ли уже вызвана функция с указанными параметрами. И если была, то не вызывать функцию снова – а вместо этого вернуть `мемоизированное` значение 🦾.

Для повышения производительности полезно избегать ненужных, дорогостоящих операций, которые выполняются повторно.

 Совет:

Вы можете применить технику `memoization` не только в рамках `Redux` – но так же каждый раз когда вы выполняете дорогую операцию где-либо. К примеру: фильтрация списков, или делая дорогие вычисления, поэтому `memoization` может гарантировать, что эти дорогостоящие операции происходят только при необходимости.

Замечательная вещь про `reselect` заключается в том, что он отлично работает в паре с `normalizr`, там где данные состоят `нормализованы` в отношения между `ID` и `entities`.

Беря во внимание `нормализованный` массив книг, как часть `состояний` приложения `book-reader`:

```
const state = {
  user: { /* user data */},
  magicBook: { /* magic book data */},
  books: {
    entities: {
      authors: {
        1: {
          id: "1",
          name: "Oscar Egilsson"
        }
      },
      books: {
        15: {
          author: "1",
          id: "15",
          title: "The Thing."
        },
        41: {
          author: "1",
          id: "41",
          title: "The best about pine trees."
        }
      }
    },
    result: ["41", "15"]
  }
};
```

Давайте создадим подходящий `reducer` для `books` части `состояния`:

```
// app/reducers/books/index.js

import { fromJS } from 'immutable';

const initialState = fromJS({
  entities: {
    authors: {},
    books: {}
  },
  result: []
});
```

```
export default (state = initialState, { action, payload }) => {
  switch (action) {
    case 'GET_BOOKS':
      return fromJS(payload);

    default:
      return state;
  }
};
```

🔍 Заметка:

Метод `fromJS()` глубоко преобразует обычный JavaScript объект и массив в Immutable Maps и Lists.

Мы можем запросить `books` часть из `state` используя функцию `createSelector` из пакета `reselect`.

```
// app/selectors/index.js

import { createSelector } from 'reselect';

const booksById = (state) => state.books.getIn(['entities', 'books']);
const booksIds = (state) => state.books.get('result');

export const getBooks = createSelector(
  booksById,
  booksIds,
  (books, ids) => ids.map((id) => books.get(id))
);
```

🔍 Заметка:

Метод `getIn()` возвращает значение по указанному пути начиная с предоставленной коллекции.

Теперь мы можем безопасно использовать наш недавно созданный селектор `getBooks()` в компоненте `Books`, который представляет список книг, доступных для чтения:

```
// app/components/Books/index.js

import { getBooks } from '../../selectors';

// ...the rest of the Books implementation...
```

```

render () {
  const { books } = this.props;

  console.log(books); // Will output:

  // [{
  //   author: "1",
  //   id: "42",
  //   title: "The best about pine trees."
  // },
  // {
  //   author: "1",
  //   id: "15",
  //   title: "The Thing."
  // }]

  return /* returned JSX with list of books */
}

// ...the rest of the Books implementation...

const mapStateToProps = (state) => ({
  books: getBooks(state).toJS()
});

export default connect(mapStateToProps)(Books);

```

🔍 Заметка:

Метод `toJS()` глубоко преобразует коллекцию `immutable` в нативный эквивалент JavaScript.

По этому если вы делаете дорогие операции в своем `mapStateToProps` – рассмотрите возможность добавления библиотеки `reslect` в ваш проект 📖.

Итоги

`Functional programming` – это процесс создания программ путем составления `pure functions`, используя `immutable data`, и избегая `side-effects`.

`Functional programmin` так же рассматривается больше как `declarative` нежели `imperative`.

`Pure (idempotent) function` – это функция, которая производит одинаковый вывод, получая одинаковые входящие параметры и не производит никаких `side effects`.

`higher-order function` – функция, которая берет параметр и/или возвращает функцию.

`Immutable` сущности в JavaScript – значения сущностей, которые не могут быть изменены. Сохранение данных вашего приложения `immutable`, делает ваше приложение `предсказуемым`. Придерживаясь принципов `неизменности`, `новые` модифицированные сущности данных должны производятся каждый раз, вместо непосредственной `мутации` предыдущих.

JavaScript `immutable` типы:

- strings
- numbers
- booleans
- null, undefined, NaN

JavaScript `mutable` типы:

- Arrays
- Functions
- Objects (более важно)

`Immutable.js` - это удивительная библиотека `immutability-helper`, которая дает нам богатый набор `неизменяемых` структур данных наряду с удобным API, чтобы облегчить работу с `не изменяемыми` данными. Её производительность так же сильно оптимизирована, так что попробуйте взглянуть на неё!

`Redux` это `синхронный` по своей природе. Нам необходимо сделать, грациозно, некоторую дополнительную конфигурацию для обработки вызовов `асинхронно`. Наилучший вариант для этого универсальный и простой – `redux-thunk`, а немного сложнее, но более мощный – `redux-saga`, которая основана на функциях генераторах ES2015.

`redux-promise` это то на что следует взглянуть.

За сохранение `state` гибким и легким в доступе отвечает служебная библиотека `normalizr`, которая может использоваться для создания `нормализованных форм состояния` основываясь на `de-normalized` API, соответственно.

Если ваша библиотека включает в себя множество дорогостоящих операций – рассмотрите возможность добавления `reselect` в ваш проект, для того чтобы минимизировать ненужные расчеты с помощью `memoized selectors`. Дополнительный бонус в том, что `Immutable.js`, `normalizr`, и `reselect` замечательно работают вместе!

Спасибо за то, что остаетесь с нами! Увидимся на следующем уроке 😊!

Если у вас есть идеи как можно улучшить этот урок, пожалуйста, поделитесь вашими мыслями с нами `hello@lectrum.io`. Ваш отзыв очень важен для нас!