Redux.

Redux Actions

**1.Экшены** — это структуры, которые передают данные из вашего приложения в стор. Они являются единственными источниками информации для стора. Вы отправляете их в стор, используя метод [store.dispatch()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/Store.html" \l "dispatch).

2. Экшены — это обычные JavaScript-объекты. Экшены должны иметь поле type, которое указывает на тип исполняемого экшена. Типы должны быть, как правило, заданы, как строковые константы. После того, как ваше приложение разрастется, вы можете захотеть переместить их в отдельный модуль.

Пример:

const ADD\_TODO = 'ADD\_TODO'

{

type: ADD\_TODO,

text: 'Build my first Redux app'

}

3. Кроме type, структуру объекта экшенов вы можете строить на ваше усмотрение. Если вам интересно, изучите [Flux Standard Action](https://github.com/acdlite/flux-standard-action" \t "_blank) для рекомендаций о том, как могут строится экшены.

Мы добавим еще один тип экшена, который будет отмечать задачу, как выполненную. Мы обращаемся к конкретному todo по index, потому что мы храним их в виде массива. В реальном приложении разумнее генерировать уникальный ID каждый раз, когда что-то новое создается.

{

type: TOGGLE\_TODO,

index: 5

}

4. Это хорошая идея, передавать как можно меньше данных в каждом экшене. Например, лучше отправить index, чем весь объект todo.

Наконец, мы добавим еще один тип экшена для изменения видимых, в данный момент, задач.

{

type: SET\_VISIBILITY\_FILTER,

filter: SHOW\_COMPLETED

}

|| || || ||

Генераторы экшенов.Action Creators.

1. **Генераторы экшенов (Action Creators)** — не что иное, как функции, которые создают экшены. Довольно просто путать термины “action” и “action creator,” поэтому постарайтесь использовать правильный термин.

В Redux генераторы экшенов (action creators) просто возвращают action:

function addTodo(text) {

return {

type: ADD\_TODO,

text

}

}

2.Это делает их более переносимыми и легкими для тестирования.

В [традиционной реализации Flux](http://facebook.github.io/flux), генераторы экшенов (action creators) при выполнении часто вызывают dispatch, примерно так:

function addTodoWithDispatch(text) {

const action = {

type: ADD\_TODO,

text

}

dispatch(action)

}

3. В Redux это *не* так. Вместо того чтобы на самом деле начать отправку, передайте результат в функцию dispatch():

dispatch(addTodo(text))

dispatch(completeTodo(index))

4. Кроме того, вы можете создать **связанный генератор экшена (bound action creator)**, который автоматически запускает отправку экшена:

const boundAddTodo = (text) => dispatch(addTodo(text))

const boundCompleteTodo = (index) => dispatch(completeTodo(index))

5. Теперь вы можете вызвать его напрямую:

boundAddTodo(text)

boundCompleteTodo(index)

6. Доступ к функции dispatch() может быть получен непосредственно из стора (store) [store.dispatch()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/Store.html" \l "dispatch), но, что более вероятно, вы будете получать доступ к ней при помощи чего-то типа connect() из [react-redux](http://github.com/gaearon/react-redux" \t "_blank). Вы можете использовать функцию [bindActionCreators()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/bindActionCreators.html) для автоматического привязывания большого количества генераторов экшенов (action creators) к функции dispatch().

Генератор экшены так же могут быть асинхронными и иметь сайд-эффекты. Вы можете почитать про [асинхронные экшены](https://rajdee.gitbooks.io/redux-in-russian/content/docs/advanced/AsyncActions.html) в [расширенном руководстве](https://rajdee.gitbooks.io/redux-in-russian/content/docs/advanced/), чтобы узнать, как обрабатывать ответы AJAX и создавать генераторы действий в асинхронном потоке управления. Не переходите к асинхронным экшенам до тех пор, пока вы не завершите базовое руководство, так как оно охватывает другие важные концепции, которые необходимы для продвинутого руководства и асинхронных экшенов.

**Исходный код**

**actions.js**

/\*

\* типы экшенов

\*/

export const ADD\_TODO = 'ADD\_TODO'

export const TOGGLE\_TODO = 'TOGGLE\_TODO'

export const SET\_VISIBILITY\_FILTER = 'SET\_VISIBILITY\_FILTER'

/\*

\* другие константы

\*/

export const VisibilityFilters = {

SHOW\_ALL: 'SHOW\_ALL',

SHOW\_COMPLETED: 'SHOW\_COMPLETED',

SHOW\_ACTIVE: 'SHOW\_ACTIVE'

}

/\*

\* генераторы экшенов

\*/

export function addTodo(text) {

return { type: ADD\_TODO, text }

}

export function toggleTodo(index) {

return { type: TOGGLE\_TODO, index }

}

export function setVisibilityFilter(filter) {

return { type: SET\_VISIBILITY\_FILTER, filter }

}

**Редюсеры (Reducers)**

**7. Редюсеры** определяют, как состояние приложения изменяется в ответ на [экшены](https://rajdee.gitbooks.io/redux-in-russian/content/docs/basics/Actions.html), отправленные в стор. Помните, что экшены только описывают, \_что произошло, но не описывают, как изменяется состояние приложения.

**Проектирование структуры состояния (State)**

1.В Redux все состояние приложения хранится в виде единственного объекта. Подумать о его структуре перед написанием кода — довольно неплохая идея. Каково минимальное представление состояния Вашего приложения в виде объекта?

Для нашего todo-приложения, мы хотим хранить две разные сущности:

* Состояние фильтра видимости;
* Актуальный список todo-задач.

Часто вы будете понимать, что вам нужно хранить некоторые данные, а также некоторые состояния пользовательского интерфейса в дереве состояний. Это нормально, только старайтесь такие данные не смешивать с данными, которые описывают состояние UI.

{

visibilityFilter: 'SHOW\_ALL',

todos: [

{

text: 'Consider using Redux',

completed: true,

},

{

text: 'Keep all state in a single tree',

completed: false

}

]

}

В более сложном приложении вы, скорее всего, будете иметь разные сущности, которые будут ссылаться друг на друга. Мы советуем поддерживать состояние (state) в настолько упорядоченном виде, насколько это возможно. Старайтесь не допускать никакой вложенности. Держите каждую сущность в объекте, который хранится с ID в качестве ключа. Используйте этот ID в качестве ссылки из других сущностей или списков. Думайте о состоянии приложения (app state), как о базе данных. Этот подход детально описан в [документации к normalizr](https://github.com/gaearon/normalizr). Например, в реальном приложении хранение хеша todo-сущностей todosById: { id -> todo } и массива их ID todos: array<id> в состоянии (state) было бы лучшей идеей, но мы оставим пример простым.

## 2. Обработка экшенов

Теперь, когда мы определились с тем, как должны выглядеть наши объекты состояния (state objects), мы готовы написать редюсер для них. Редюсер (reducer) — это чистая функция, которая принимает предыдущее состояние и экшен (state и action) и возвращает следующее состояние (новую версию предыдущего).

(previousState, action) => newState

Функция называется редюсером (reducer) потому, что ее можно передать в [Array.prototype.reduce(reducer, ?initialValue)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce" \t "_blank). Очень важно, чтобы редюсеры оставались чистыми функциями. Вот список того, чего **никогда** нельзя делать в редюсере:

* Непосредственно изменять то, что пришло в аргументах функции;
* Выполнять какие-либо сайд-эффекты: обращаться к API или осуществлять переход по роутам;
* Вызывать не чистые функции, например Date.now() или Math.random().

Мы рассмотрим способы выполнения сайд-эффектов в [продвинутом руководстве](https://rajdee.gitbooks.io/redux-in-russian/content/docs/advanced/). На данный момент просто запомните, что редюсер должен быть чистым. **Получая аргументы одного типа, редюсер должен вычислять новую версию состояния и возвращать ее. Никаких сюрпризов. Никаких сайд-эффектов. Никаких обращений к стороннему API. Никаких изменений (mutations). Только вычисление новой версии состояния.**

Исходя из вышенаписанных принципов, давайте начнем писать редюсер, постепенно обучая его понимать [экшены (actions)](https://rajdee.gitbooks.io/redux-in-russian/content/docs/basics/Actions.html), которые мы описали чуть раньше.

Мы начнем с определения начального состояния (initial state). В первый раз Redux вызовет редюсер с неопределенным состоянием(state === undefined). Это наш шанс инициализировать начальное состояние приложения:

import { VisibilityFilters } from './act

3. import { VisibilityFilters } from './actions'

const initialState = {

visibilityFilter: VisibilityFilters.SHOW\_ALL,

todos: []

}

function todoApp(state, action) {

if (typeof state === 'undefined') {

return initialState

}

// Пока не обрабатываем никаких экшенов

// и просто возвращаем состояние, которое приняли в качестве параметра

return state

}

Использование [синтаксиса аргументов по умолчанию из ES6](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Default_parameters) для более компактного написания — просто аккуратный трюк:

function todoApp(state = initialState, action) {

// Пока не обрабатываем никаких экшенов

// и просто возвращаем состояние, которое приняли в качестве параметра

return state

}

4. Теперь давайте начнем обрабатывать экшен SET\_VISIBILITY\_FILTER. Все, что нужно сделать — это изменить visibilityFilter в состоянии приложения. Это просто:

import {

SET\_VISIBILITY\_FILTER,

VisibilityFilters

} from './actions'

...

function todoApp(state = initialState, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return Object.assign({}, state, {

visibilityFilter: action.filter

})

default:

return state

}

}

5. Обратите внимание:

**Мы не изменяем state**. Мы создаем копию с помощью [Object.assign()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign" \t "_blank). Object.assign(state, { visibilityFilter: action.filter }) тоже неверный вариант: в этом случае первый аргумент будет изменен. Вы **должны** передать первым аргументом пустой объект. Вы также можете подключить [object spread operator proposal](https://rajdee.gitbooks.io/redux-in-russian/content/docs/recipes/UsingObjectSpreadOperator.html), чтобы вместо этого писать { ...state, ...newState } .

**Мы возвращаем предыдущую версию состояния (state) в default ветке.** Очень важно возвращать предыдущую версию состояния (state) для любого неизвестного/необрабатываемого экшена (action).

##### 6. **Обратите внимание на Object.assign**

[Object.assign()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) это часть ES6, но этот метод не поддерживается старыми браузерами. Вам нужно будет использовать использовать полифилл, [плагин для Babel](https://www.npmjs.com/package/babel-plugin-transform-object-assign), либо хелпер из другой библиотеки, к примеру [\_.assign() из lodash](https://lodash.com/docs#assign).

**Обратите внимание на switch и шаблон (boilerplate)**

Конструкция switch *не является* реальным требованием. Реальный шаблон Flux определяется концепцией: необходимость инициировать обновление, необходимость зарегистрировать стор (Store) в Dispatcher'е, необходимость, чтобы Store был объектом (возникают осложнения, если вы хотите универсальное приложение (universal app)). Redux решает эти проблемы благодаря использованию чистых редюсеров вместо генераторов событий (event emitters)

Если вам не нравится конструкция switch, вы можете использовать собственную функцию createReducer, которая принимает объект обработчиков, как показано в [“упрощение шаблона (reducing boilerplate)”](https://rajdee.gitbooks.io/redux-in-russian/content/docs/recipes/ReducingBoilerplate.html#reducers).

## 7. Обрабатываем больше экшенов.

У нас есть еще два экшена, которые должны быть обработаны! Так же, как мы сделали с SET\_VISIBILITY\_FILTER мы имортируем ADD\_TODO и TOGGLE\_TODO экшены и затем допишем наш редюсер для обработки ADD\_TODO.

import {

ADD\_TODO,

TOGGLE\_TODO,

SET\_VISIBILITY\_FILTER,

VisibilityFilters

} from './actions'

...

function todoApp(state = initialState, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return Object.assign({}, state, {

visibilityFilter: action.filter

})

case ADD\_TODO:

return Object.assign({}, state, {

todos: [

...state.todos,

{

text: action.text,

completed: false

}

]

})

default:

return state

}

}

8.Как и раньше, мы никогда не изменяем непосредственно state или его поля. Вместо этого мы возвращаем новый объект. Новый todos равен старому todos, в конец которого добавлен новый элемент todo. Свежий tod был создан с использованием информации, полученной из action.

Ну и наконец, имплементация обработчика для экшена TOGGLE\_TODO не должна стать для Вас большим сюрпризом:

case TOGGLE\_TODO:

return Object.assign({}, state, {

todos: state.todos.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: !todo.completed

})

}

return todo

})

})

9. Поскольку мы хотим обновить конкретный элемент в массиве, не прибегая к мутациям, мы должны создать новый массив с теми же элементами, за исключением элемента по индексу. Если вы часто пишете такие операции, рекомендуется использовать хэлперы, такие как [immutability-helper] (<https://github.com/kolodny/immutability-helper>), [updeep] (<https://github.com/essential/updeep>) или даже такую библиотеку, как [Immutable] (<http://facebook.github.io/immutable-js/>), которая имеет встроенную поддержку для глубоких обновлений. Просто запомните, что нельзя присваивать ничего внутри state, пока вы его не склонировали.

**Разделение редюсеров**

Вот так выглядит наш код на данный момент. Выглядит излишне многословным:

function todoApp(state = initialState, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return Object.assign({}, state, {

visibilityFilter: action.filter

})

case ADD\_TODO:

return Object.assign({}, state, {

todos: [

...state.todos,

{

text: action.text,

completed: false

}

]

})

case TOGGLE\_TODO:

return Object.assign({}, state, {

todos: state.todos.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: !todo.completed

})

}

return todo

})

})

default:

return state

}

}

Есть ли способ облегчить понимание? Кажется, что todos и visibilityFilter обновляются совершенно независимо. Иногда поля состояния (state fields) зависят от других полей и требуется большая связанность, но в нашем случаем мы безболезненно можем вынести обновление todos в отдельную функцию:

function todos(state = [], action) {

switch (action.type) {

case ADD\_TODO:

return [

...state,

{

text: action.text,

completed: false

}

]

case TOGGLE\_TODO:

return state.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: !todo.completed

})

}

return todo

})

default:

return state

}

}

function todoApp(state = initialState, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return Object.assign({}, state, {

visibilityFilter: action.filter

})

case ADD\_TODO:

return Object.assign({}, state, {

todos: todos(state.todos, action)

})

case TOGGLE\_TODO:

return Object.assign({}, state, {

todos: todos(state.todos, action)

})

default:

return state

}

}

Обратите внимание, что функция todos также принимает state, но state — это массив! Теперь todoApp просто передает срез состояния в функцию todos, которая, свою очередь, точно знает, как обновить именно этот кусок состояния. **Это называется *композицией редюсеров* и является фундаментальным шаблоном построения Redux-приложений.**

Давайте рассмотрим композицию редюсеров подробнее. Можем ли мы извлечь редюсер, который будет управлять только visibilityFilter? Конечно можем:

Ниже нашего импорта давайте использовать [ES6 Object Destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment), чтобы объявить SHOW\_ALL:

const { SHOW\_ALL } = VisibilityFilters

Затем:

function visibilityFilter(state = SHOW\_ALL, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return action.filter

default:

return state

}

}

Теперь мы можем переписать наш главный редюсер в виде функции, которая вызывает другие редюсеры, обрабатывающие части состояния и собирает отдельно обработанные части состояния в один цельный объект. Также главному редюсеру больше нет необходимости знать полное начальное состояние. Достаточно того, что каждый дочерний редюсер возвращает свое начальное состояние, если при первом вызове получает undefined вместо state.

function todos(state = [], action) {

switch (action.type) {

case ADD\_TODO:

return [

...state,

{

text: action.text,

completed: false

}

]

case TOGGLE\_TODO:

return state.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: !todo.completed

})

}

return todo

})

default:

return state

}

}

function visibilityFilter(state = SHOW\_ALL, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return action.filter

default:

return state

}

}

function todoApp(state = {}, action) {

return {

visibilityFilter: visibilityFilter(state.visibilityFilter, action),

todos: todos(state.todos, action)

}

}

**Обратите внимание на то, что каждый из этих дочерних редюсеров управляет только какой-то одной частью глобального состояния. Параметр state разный для каждого отдельного дочернего редюсера и соответствует той части глобального состояния, которой управляет этот дочерний редюсер.**

Уже выглядит лучше! Когда приложение разрастается, мы можем выносить редюсеры в отдельные файлы и поддерживать их совершенно независимыми, что дает нам возможность управлять различными разделами наших данных.

Наконец, Redux предоставляет утилиту, называемую [combineReducers()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/combineReducers.html), которая реализует точно такой же логический шаблон, который мы только что реализовали в todoApp. С ее помощью мы можем переписать todoApp следующим образом:

import { combineReducers } from 'redux'

const todoApp = combineReducers({

visibilityFilter,

todos

})

export default todoApp

Обратите внимание, что это полностью эквивалентно такому коду:

export default function todoApp(state = {}, action) {

return {

visibilityFilter: visibilityFilter(state.visibilityFilter, action),

todos: todos(state.todos, action)

}

}

Вы также можете назначать им разные ключи или вызывать функции по-разному. Есть два совершенно равноценных способа писать комбинированные редюсеры:

const reducer = combineReducers({

a: doSomethingWithA,

b: processB,

c: c

})

function reducer(state = {}, action) {

return {

a: doSomethingWithA(state.a, action),

b: processB(state.b, action),

c: c(state.c, action)

}

}

Все, что делает [combineReducers()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/combineReducers.html) — это генерирует функцию, которая вызывает ваши редюсеры c **частью глобального состояния, которая выбирается в соответствии с их ключами**, и затем снова собирает результаты всех вызовов в один объект. [Тут нет никакой магии.](https://github.com/reactjs/redux/issues/428#issuecomment-129223274) И, как и другие редюсеры, combineReducers() не создает новый объект, если все предоставленные ему редюсеры не изменяют состояние.

**Заметка для сообразительных пользователей синтаксиса ES6**

Т.к. combineReducers ожидает на входе объект, мы можем поместить все редюсеры верхнего уровня в разные файлы, экспортировать каждую функцию-редюсер и использовать import \* as reducers для получения их в формате объекта, ключами которого будут имена экспортируемых функций.

import { combineReducers } from 'redux'

import \* as reducers from './reducers'

const todoApp = combineReducers(reducers)

Поскольку import \* — это все еще новый синтаксис, мы не используем его нигде в документации во избежание [путаницы](https://github.com/reactjs/redux/issues/428#issuecomment-129223274), но вы можете случайно наткнуться на него в каких-нибудь примерах кода из сообщества.

**Исходный код**

**reducers.js**

import { combineReducers } from 'redux'

import {

ADD\_TODO,

TOGGLE\_TODO,

SET\_VISIBILITY\_FILTER,

VisibilityFilters

} from './actions'

const { SHOW\_ALL } = VisibilityFilters

function visibilityFilter(state = SHOW\_ALL, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return action.filter

default:

return state

}

}

function todos(state = [], action) {

switch (action.type) {

case ADD\_TODO:

return [

...state,

{

text: action.text,

completed: false

}

]

case TOGGLE\_TODO:

return state.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: !todo.completed

})

}

return todo

})

default:

return state

}

}

const todoApp = combineReducers({

visibilityFilter,

todos

})

export default todoApp

Redux-store

**Стор (Store)**

В предыдущих разделах мы определили [экшены](https://rajdee.gitbooks.io/redux-in-russian/content/docs/basics/Actions.html), которые представляют факт того, что "что-то случилось" и [редюсеры](https://rajdee.gitbooks.io/redux-in-russian/content/docs/basics/Reducers.html), которые обновляют состояние (state) в соответствии с этими экшенами.

**Стор (Store)** — это объект, который соединяет эти части вместе. Стор берет на себя следующие задачи:

* содержит состояние приложения (application state);
* предоставляет доступ к состоянию с помощью [getState()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/Store.html" \l "getState);
* предоставляет возможность обновления состояния с помощью [dispatch(action)](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/Store.html" \l "dispatch);
* Обрабатывает отмену регистрации слушателей с помощью функции, возвращаемой [subscribe(listener)](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/Store.html" \l "subscribelistener).

Важно отметить, что у вас будет только один стор в Redux-приложении. Если Вы захотите разделить логику обработки данных, то нужно будет использовать [композицию редюсеров (reducer composition)](https://rajdee.gitbooks.io/redux-in-russian/content/docs/basics/Reducers.html#splitting-reducers) вместо использования множества сторов (stores).

Очень легко создать стор (Store), если у Вас есть редюсер. В [предыдущем разделе](https://rajdee.gitbooks.io/redux-in-russian/content/docs/basics/Reducers.html) мы использовали [combineReducers()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/combineReducers.html) для комбинирования несколько редюсеров в один глобальный редюсер. Теперь мы их импортируем и передадим в [createStore()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/createStore.html).

import { createStore } from 'redux'

import todoApp from './reducers'

let store = createStore(todoApp)

Вы можете объявить начальное состояние, передав его вторым аргументом в [createStore()](https://rajdee.gitbooks.io/redux-in-russian/content/docs/api/createStore.html). Это полезно для пробрасывания состояния на клиент из состояния приложения Redux, работающего на сервере.

const store = createStore(todoApp, window.STATE\_FROM\_SERVER)

**Отправка экшенов (Dispatching Actions)**

На текущий момент у нас есть созданный стор, давайте проверим, как работает наше приложение! Даже без любого UI мы уже можем проверить логику обновления состояния.

import {

addTodo,

toggleTodo,

setVisibilityFilter,

VisibilityFilters

} from './actions'

// Выведем в консоль начальное состояние

console.log(store.getState())

// Каждый раз при обновлении состояния - выводим его

// Отметим, что subscribe() возвращает функцию для отмены регистрации слушателя

const unsubscribe = store.subscribe(() => console.log(store.getState()))

// Отправим несколько экшенов

store.dispatch(addTodo('Learn about actions'))

store.dispatch(addTodo('Learn about reducers'))

store.dispatch(addTodo('Learn about store'))

store.dispatch(toggleTodo(0))

store.dispatch(toggleTodo(1))

store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW\_COMPLETED))

// Прекратим слушать обновление состояния

unsubscribe()

Вы можете видеть, как выполнение кода, приведенного выше, меняет состояние, содержащееся в сторе:



Мы смогли определить поведение нашего приложения даже до того, как начали создавать какой-то UI. Мы не будем делать этого в этом руководстве, но с этого момента Вы можете писать тесты для редюсеров и генераторов экшенов (action creators). Вам не нужно будет ничего "мокать", потому что редюсеры — это просто [чистые](https://rajdee.gitbooks.io/redux-in-russian/content/docs/introduction/ThreePrinciples.html#changes-are-made-with-pure-functions) функции. Вызывайте их и делайте проверки (make assertions) того, что они возвращают.

**Исходный код (Source Code)**

**index.js**

import { createStore } from 'redux'

import todoApp from './reducers'

let store = createStore(todoApp)