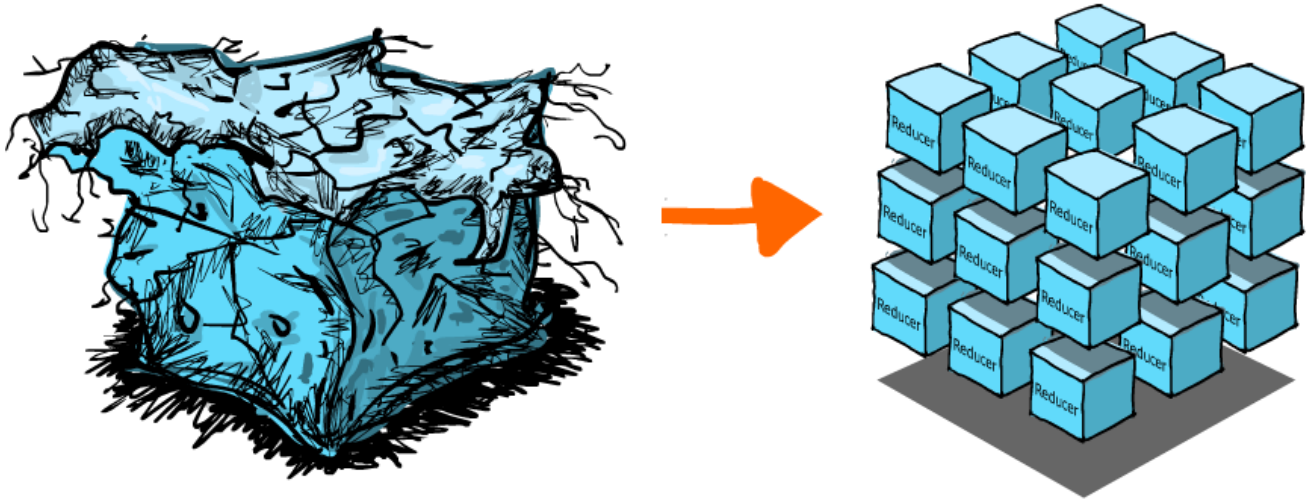
Основы Redux

Требования к функционалу приложений постоянно растут, в результате растет количество состояний интерфейса: переходы по раутам, лоадеры, асинхронная загрузка данных, бесконечное количество элементов интерфейса и т. п.

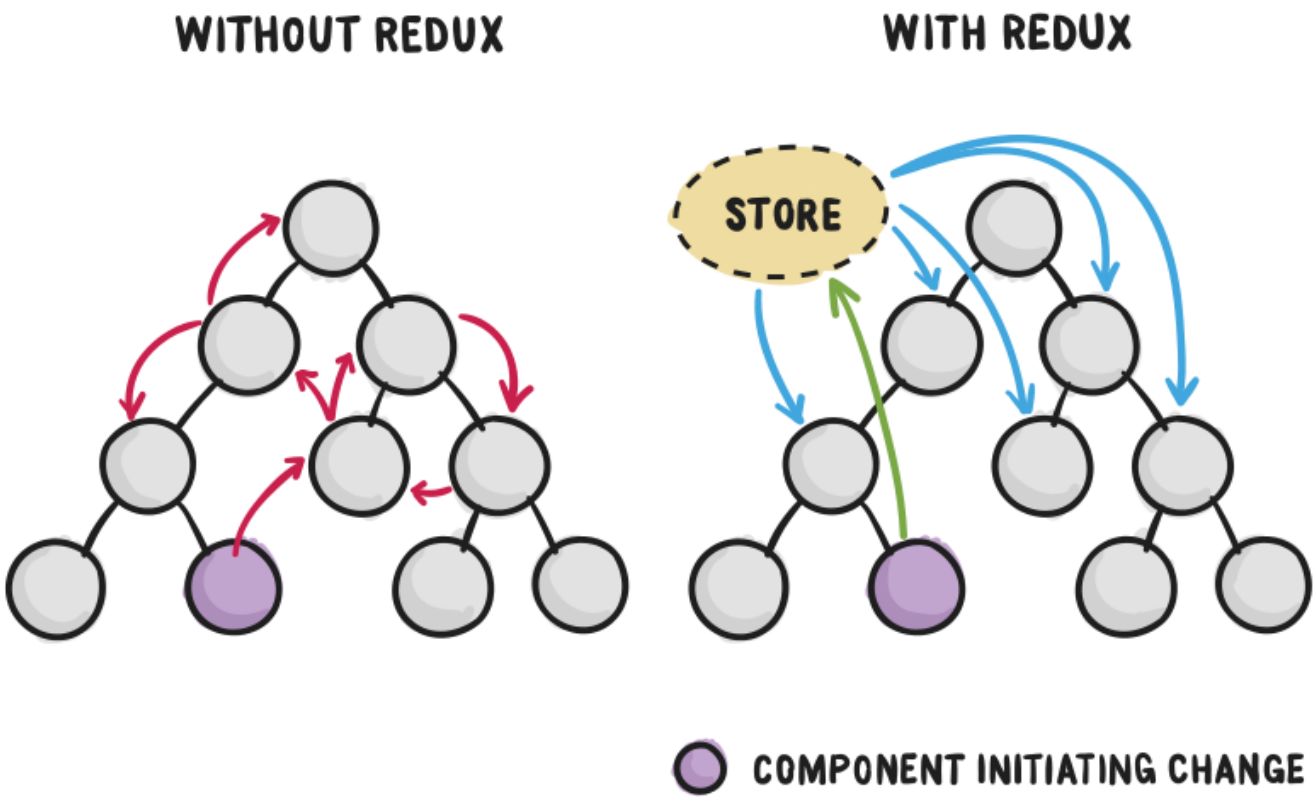
За всем эти необходимо следить и обрабатывать... это не просто. В один момент можно просто перестать улавливать связь между изменениями, так как контроль над тем, когда, почему и как изменилось состояние потерян из-за сложности самого состояния.

Идеальный вариант, это когда интерфейс вообще не знает о бизнес-логике. В этом нам помогают библиотеки управления состоянием, `Redux` и `Mobx` самые популярные. Для бекендов основаных на GraphQL есть `Apollo`.



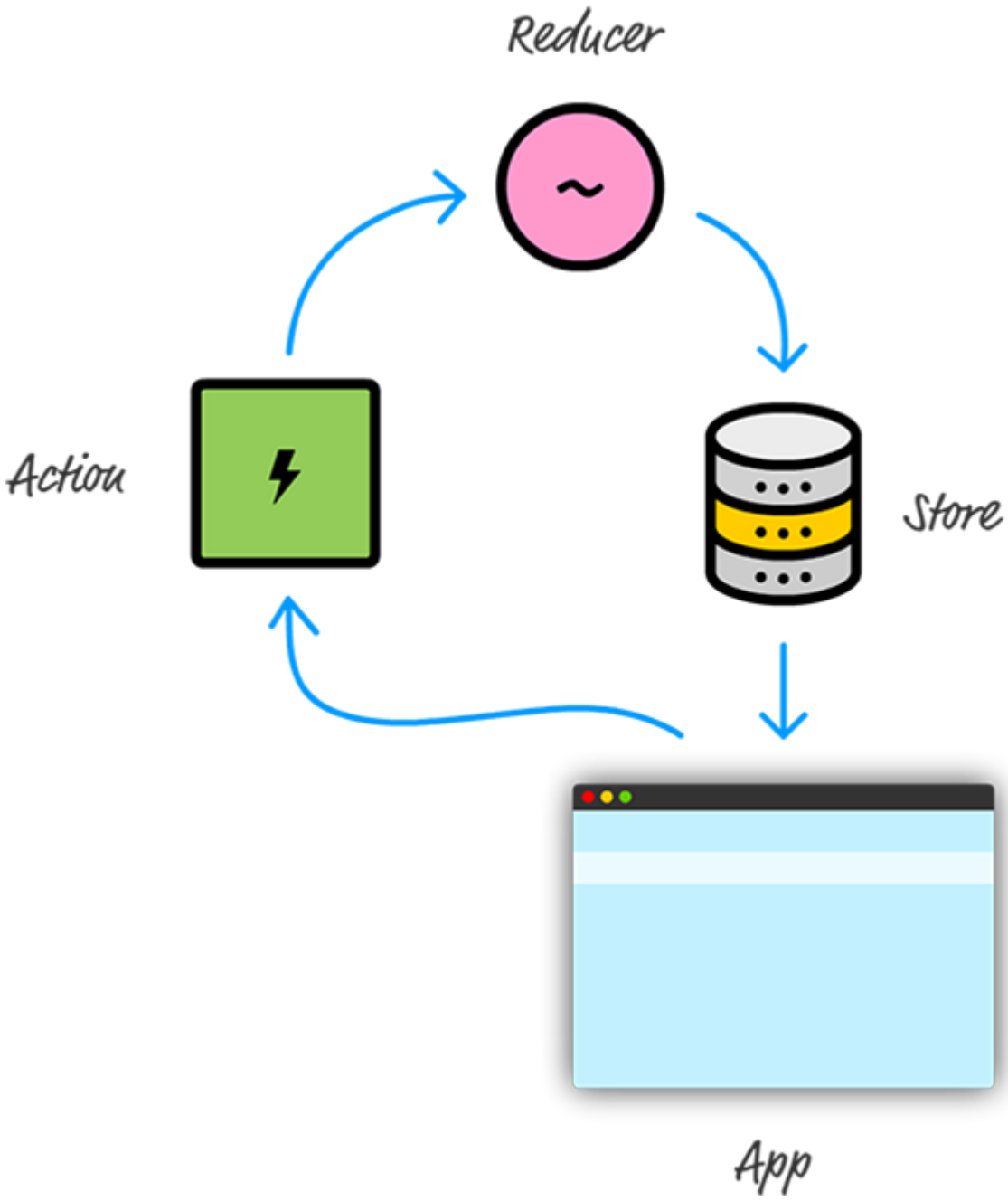
Основные принципы Redux

- **Предсказуемость результата** - существует всегда один источник правды, `store`, хранящий в себе состояние приложения и методы для работы с ним
- **Поддреживаемость** - Redux имеет набор правил и лучших практик о том, как должен быть структурирован код, что делает его более единообразным и понятным
- **Инструменты разработчика** - все происходящее можно отслеживать в режиме реального времени
- **Простота тестирования** - первое правило написания тестируемого кода - писать небольшие функции, которые выполняют только одну вещь и независимы. Redux - это в основном функции: маленькие, чистые и изолированные.
- Независим от React, но отлично работает с React



Поток данных

Поток данных в Redux всегда однонаправленный, и очень простой.



- UI (интерфейс) инициализирует отправку действий (`actions`)
- Хранилище (`store`) вызывает все объявленные редьюсеры (`reducers`), передавая им текущее состояние (`state`) и действие (`action`)
- Хранилище (`store`) сохраняет обновленное дерево состояния (`state`) возвращенное из редьюсеров (`reducers`)
- При обновлении состояния (`state`) вызываются все подписчики для обновления интерфейса

When do I know I'm ready for Redux?

Store

Хранилище (`store`) - js-объект, который содержит состояние приложения и методы доступа к нему, отправки действий и регистрации слушателей.

- Хранит состояние `state` всего приложения как один объект
- Позволяет получить доступ к состоянию через `getState()`
- Напрямую `state` доступен только для чтения
- Позволяет обновлять состояние посредством `dispatch(action)`
- Единственный способ изменить состояние (`state`) - отправить действие (`action`), объект, описывающий, что произошло
- Изменения производятся с использованием чистых функций (`reducers`), которые реагируют на действия (`actions`)
- Регистрирует слушателей используя `subscribe(listener)`

Так как все состояние приложения хранится как один объект, стоит подумать о форме состояния прежде чем писать какой-либо код. Продумайте минимальное представление состояния приложения в виде объекта.

```
const state = {
  notes: [],
  filter: '',
  session: {
    user: {
      name: null,
      avatar: null,
    },
    token: null,
  },
};
```

Функция createStore

Для того чтобы создать хранилище, используется функция `createStore`. Она принимает набор параметров и возвращает созданное хранилище.

```
createStore(reducer, [preloadedState], [enhancer])
```

- **reducer** - функция, которая возвращает следующее дерево состояния, учитывая текущее дерево состояния и действие для обработки.
- **preloadedState** - начальное состояние, к примеру сериализованное состояние последнего пользовательского сеанса. Это должен быть объект той же формы, что и, как минимум, часть состояния.
- **enhancer** - расширяет возможности хранилища при помощи сторонних дополнений, к примеру программных прослоек (middleware).

```
import { createStore } from 'redux';

// Используем редьюсер-болванку
const reducer = (state = {}, action) => state;
```

```
const store = createStore(reducer);
```

Actions

Действия (`actions`) - это события, они доставляют данные из приложения в хранилище.

- Обычные JS-объекты.
- Несут в себе информацию для хранилища (`store`).
- Должны иметь свойство `type`, которое указывает тип выполняемого действия.
- Помимо поля `type`, структура действия может быть произвольной.
- Содержат минимально необходимый набор информации.
- Типы определяются как строковые константы.

```
const action = {
  type: 'ADD_NOTE',
  payload: {
    text: 'Redux is awesome!',
  },
};
```

Действия (`actions`) создаются функциями (`action creators`), которые могут быть асинхронными и иметь побочные эффекты. В базовом варианте они просто возвращают объект-дейтсвие.

```
const addNote = text => ({
  type: 'ADD_NOTE',
  payload: {
    id: Date.now(),
    text,
  },
});
```

Reducers

Редьюсеры (reducer) - это чистые функции , которые принимают предыдущее состояние приложения (`state`) и действие (`action`), а затем возвращают новое следующее состояние.

Они определяют, как изменяется состояние (`state`) приложения в ответ на действия (`actions`), отправленные в хранилище. Помните, что действия описывают только то, что произошло, а не как изменяется состояние приложения.

```
(previousState, action) => newState;
```

Вещи, которые нельзя делать внутри редьюсера:

- Мутировать аргументы
- Выполнять побочные эффекты, такие как API-запросы и т. п.
- Вызывать нечистые функции, например `Date.now()`

Как выполнять побочные эффекты мы рассмотрим далее, пока что просто помните - редьюсер должен быть чистым. Получая те же аргументы, он должен вычислить следующее состояние и вернуть его. Без сюрпризов. Никаких побочных эффектов. Никаких мутаций. Просто рассчет.

Вот редьюсер, который принимает текущее состояние и действие как аргументы, а затем возвращает следующее состояние:

```
const initialState = [];

function notesReducer(state = initialState, action) {
  switch (action.type) {
    case 'ADD_NOTE':
      return [...state, action.payload];
    default:
      return state;
  }
}
```

Обратите внимание:

- Мы создаем копию `state`, а не мутируем его.
- Мы возвращаем предыдущее состояние по умолчанию. Важно вернуть предыдущее состояние для любого неизвестного действия.

Why Redux need reducers to be "pure functions"

Оптимизация

При каждом изменении объект состояния (`state`) создается заново. Логично предположить что это очень трудозатратно с точки зрения производительности.

На самом деле если состояние (`state`) изменяет только некоторые значения, Redux создает новый объект, но значения которые не изменились, ссылаются на старый объект состояния, а созданы будут только новые значения.

React и Redux

Для того чтобы использовать React и Redux вместе, необходимо добавить пакет `react-redux`. Это набор компонентов связывающих React-компоненты и Redux-хранилище.

```
npm install react-redux
```

Репозиторий и документация react-redux

Provider

Компонент `Provider`, оборачивает все дерево компонентов приложения и, используя контекст, предоставляет `store` и его методы.

```
import { createStore } from 'redux';
import { Provider } from 'react-redux';

// Болванка под reducer
const reducer = (state = {}, action) => state;

const store = createStore(reducer);

<Provider store={store}>
  <App />
</Provider>;
```

Provider docs

connect()

Если какой-либо компонент хочет получить доступ к `store`, он должен быть обернут в функцию `connect()`, которая свяжет компонент и `store`. Предоставляет доступ к `state` и `disaptch()`.

`connect` это HOC, он не изменяет переданный ему компонент, а возвращает новый компонент связанный с хранилищем.

```
connect(mapStateToProps, mapDispatchToProps, mergeProps, options)
(Component)
```

mapStateToProps(state, [ownProps]) - функция соединяющая часть состояния (`state`) с пропами компонента. Таким образом, связанный компонент будет иметь доступ к необходимой ему части состояния (`state`).

- Получает `state` как параметр и позволяет выбрать из всего `state` только необходимые компоненту слайсы (части).
- Возвращает объект, свойства которого дополнят `props` компонента.
- Вызывается каждый раз , когда хранилище обновляется.
- Если нет необходимости подписываться на обновления, передаем `null`.
- Если функция объявлена как принимающая два параметра, первым будет передана ссылка на `state`, а вторым ссылка на пропы самого компонента.

```
const mapStateToProps = (state, props) => ({
  notes: state.notes,
  currentFilter: state.filter,
});
```

mapDispatchToProps(dispatch, [ownProps]) - функция соединяющая отправку действий (`actions`) с пропами компонента. Таким образом, связанный компонент сможет отправлять действия посредством вызова методов указанных в возвращаемом объекте.

- Получает ссылку на `dispatch` как параметр и позволяет объявить методы для отправки действий `actions`.
- Возвращает объект, свойства которого дополнят `props` компонента.
- Если функция объявлена как принимающая два параметра, первым будет передана ссылка на `dispatch`, а вторым ссылка на пропы самого компонента.

```
const addNote = text => ({
  type: 'ADD_NOTE',
  text,
});

const mapDispatchToProps = (dispatch, props) => ({
  addNote: text => dispatch(addNote(text)),
});
```

В случае когда аргументы действия совпадают с параметрами объявляемого метода, можно вместо функции передать объект. В таком случае `connect` пройдет по ключам объекта и обернет их в `dispatch`.

```
const addNote = text => ({
  type: 'ADD_NOTE',
  text,
});

const mapDispatchToProps = { addNote };
```

Документация connect

Redux DevTools

Чтобы упростить работу с Redux существуют Redux DevTools. Они помогают отслеживать изменение состояния с течением времени, наблюдать за отправкой дейтсвий и т. п. Естественно, это предназначено только для разработки.

Репозиторий redux-devtools-extension

Организация хранилища

Необходимо хранить все состояние приложения в Redux? Можно ли использовать `setState()`?

Точного овета нет. Одни предпочитают хранить все данные в Redux. Другие хранят некритическое или пользовательское состояние, например состояние элемента интерфейса, в состоянии компонента.

Как разработчик, ваша задача - определить, из какого набора данных состоит приложение, и где их лучше хранить.

Некоторые общие правила для определения того, какие данные должны быть помещены в Redux:

- Необходимы ли эти данные другим частям приложения?
- Необходимо ли на основе этих данных создавать дополнительные производные?
- Используются ли одни и те же данные для управления несколькими компонентами?
- Есть ли необходимость в кешировании?

Материалы

- Where to Hold React Component Data: state, store, static, and this
- The 5 Types Of React Application State

Множественные редьюсеры

Еще одна полезная функция Redux - возможность делать композицию редьюсеров, то есть совмещать много в один. Это позволяет удобно поддерживать гораздо более сложные состояния в больших приложениях. Для этого используется функция `combineReducers()`.

```
import { combineReducers } from 'redux';
import notesReducer from './notes';
import filterReducer from './filter';

const rootReducer = combineReducers({
  notes: notesReducer,
  filter: filterReducer,
});

export default rootReducer;
```

Дополнительные материалы

- Документация Redux
- Документация Redux на русском
- A cartoon intro to Redux
- Redux: шаг за шагом
- Getting Started with Redux by Dan Abramov
- Building React Applications with Idiomatic Redux by Dan Abramov
- Using Redux-Actions — Why and How?
- React-Redux workflow - graphical cheat sheet

Licence