**Конспект лекции - Управление состоянием приложения**

**Цель:** понять, что такое состояние приложения

**Задачи:**

1. изучить виды состояний;
2. обзор архитектурных паттернов;
3. обзор сложившихся практик по решению проблем управления состояний.

**План:**

1. состояния приложения;
2. рассмотрим виды состояний;
3. посмотрим на популярные архитектурные паттерны;
4. разберем основные практики: Redux, Mobx, React Context API.

**Состояния приложения**

**Состояние приложения** - информация, к которой программа имеет доступ и с которой может работать в определенный момент времени. Сюда входят данные, хранящиеся в памяти, порты ввода/вывода, базы данных и так далее.

**Приложения с состоянием.** Программы, приложения или компоненты с состоянием хранят в памяти данные о текущем состоянии. Они могут изменять состояние, а также имеют доступ к его истории. Следующий пример демонстрирует это:

var number = 1;

function increment() {

return number++;

}

// глобальная переменная изменяется: number = 2

increment();

**Приложения без состояния.** Функции или компоненты без состояния выполняют задачи, словно каждый раз их запускают впервые. Они не ссылаются или не используют в своем исполнении раннее созданные данные. Отсутствие состояния обеспечивает ссылочную прозрачность. Функции зависят только от их аргументов и не имеют доступа, не нуждаются в знании чего-либо вне их области видимости. [Чистые функции](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8) не имеют состояния. Пример:

var number = 1;

function increment(n) {

return n + 1;

}

// глобальная переменная НЕ изменяется: возвращает 2

increment(number);

Управление состоянием важно для любого сложного приложения. Функции или компоненты с состоянием изменяют состояние и его историю, их труднее тестировать и отлаживать. Функции без состояния полагаются только на свои входные данные для создания данных выходных. Программа без состояния возвращает новое состояние, а не модифицирует существующее состояние.

**Виды состояний**

* Состояние сервера
* Постоянное состояние
* Состояние URL и маршрутизации
* Состояние клиента
* Временное состояние клиента
* Локальное состояние интерфейса

Теперь рассмотрим более подробно.

**Состояние сервера** хранится на сервере.

**Постоянное состояние** сервера – это подмножество состояний сервера, хранящееся на клиенте в памяти. Можно рассматривать постоянное состояние как кэш состояния сервера. Но в реальных приложениях данное состояние почти не используется т.к. применяются частые обновления интерфейса пользователя для лучшего восприятия.

**Состояние клиента** не хранятся на сервере. Как пример состоянием клиента являются фильтры товаров для отображения пользователю. При этом элементы фильтрации хранятся в некой базе данных на сервере, а вот значения фильтров нет.

Good Practiceсохранять постоянное и клиентское состояние в **URL**.

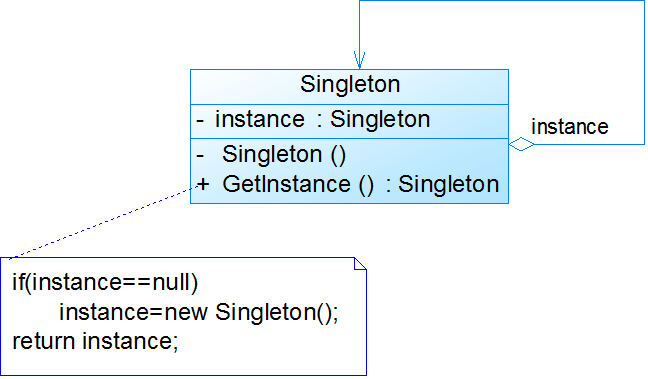
Приложения часто имеют состояние, которое хранится на клиенте, но не представлено в URL. Например, YouTube помнит, где остановлено видео. Поэтому в следующий раз, когда начать его смотреть, видео возобновится с сохраненного момента. Поскольку информация о времени не хранится в URL, передав ссылку с видео кому-то другому, он начнет смотреть видео с самого начала. Это **временное состояние клиента**.

Отдельные компоненты могут иметь локальное состояние, управляющими интерфейсом пользователя. На пример цвет кнопки, какая закладка открыта и другие варианты. Это **локальное состояние клиента.**

**Архитектурные паттерны**

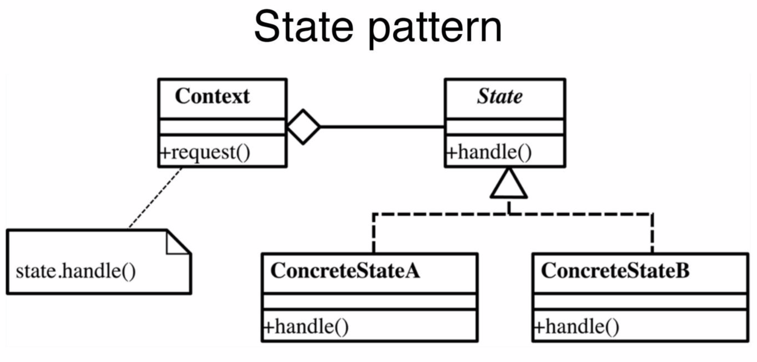
**Одиночка (Singleton)** – это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Это не просто глобальная переменная, через которую можно достучаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

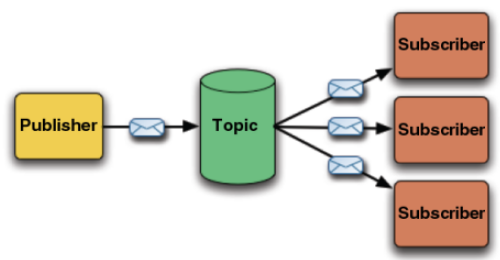


**Состояние (State) — это** поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

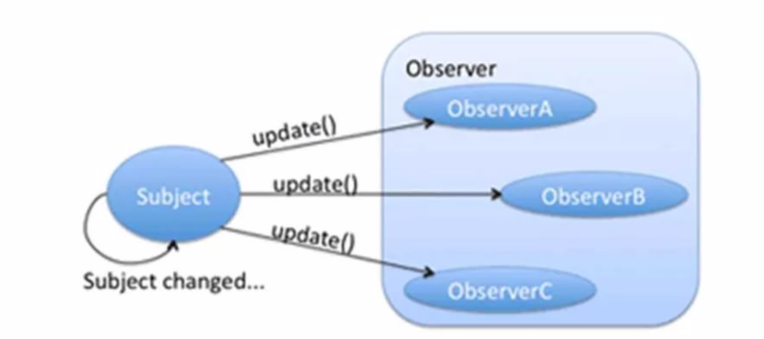
Основная идея в том, что программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и конечен. Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.



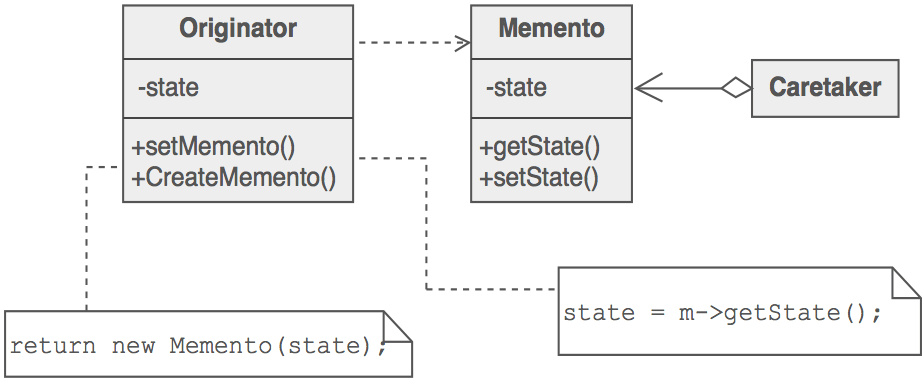
**Издатель-подписчик (****Subscriber-publisher)** - поведенческий шаблон проектирования передачи сообщений, в котором отправители сообщений, именуемые издателями (англ. publishers), напрямую не привязаны программным кодом отправки сообщений к подписчикам (англ. subscribers). Вместо этого сообщения делятся на классы и не содержат сведений о своих подписчиках, если таковые есть. Аналогичным образом подписчики имеют дело с одним или несколькими классами сообщений, абстрагируясь от конкретных издателей.



**Наблюдатель (****Observer)** - представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

****

**Хранитель (****Memento)** - получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии. Является средством для инкапсуляции "контрольных точек" программы.Придает операциям "Отмена" (undo) или "Откат" (rollback) статус "полноценного объекта".



**Популярные архитектурные паттерны**

**Mobx**

Mobx сравнивают с другими библиотеками для управления состоянием, не предоставляет практически никаких удобств для работы с состоянием, за исключением вызова обновления компонентов react после того как меняется свойство помеченное @observable декоратором. Можно легко выбросить mobx убрав все @observable и @observer-декораторы и получить работающее приложение, добавив всего одну строчку update() в конце всех обработчиков событий где меняем данные состояния, которые выводятся в компонентах.

onCommentChange(e) {

const {comment} = this.props;

comment.text = e.target.value;

update(); //добавили одну строчку

}

Функция update() просто вызовет pre-render react-приложения и благодаря virtual DOM react в DOM применится только разница изменений.

function update(){

ReactDOM.render(<App>, document.getElementById('root');

}

Mobx - это целый state-менеджер потому, что позволяет сэкономить одну строчку update() в обработчиках как-то чересчур.

В отличии от него Redux позволяет удобно организовывать работу с состоянием через event-sourcing паттерн когда не обновляем состояние на месте, а dispatch-им объект изменения (action) и обрабатываем в совсем другом месте — в так называемых чистых функциях-reducers, а благодаря единой шине событий мы можем добавлять какую-то удобную работу с асинхронностью, перехватывая эти actions в конвейере middleware-ов и упростить отладку приложение через time-travel.

Mobx - это не та библиотека, которая упрощает работу с состоянием. Основная задача — это точечное обновление компонентов, а именно — вызывать обновление только тех компонентов, которые зависят от данных, которые поменялись.

В примере выше каждый раз когда меняется любые данные в приложении выполняем pre-render (сравнение virtual DOM) всего приложения, вызывая ReactDOM.render(<App>, document.getElementById('root')) в функции update() и это влияет на производительность, и на больших приложениях интерфейс неизбежно будет тормозить.

Несмотря на то что react изобрел virtual DOM со слоганом, что DOM медленный, а virtual DOM быстрый потому, что он сравнивает только деревья объектов в памяти, а в реальном DOM обновляет только измененные части, в реальности мы не можем при любом обновлении данных в приложении вызвать это сравнение virtual DOM для всего приложения потому что это медленно.

И тогда решением проблемы будет не полагаться на virtual DOM и обновлять компоненты вручную, вызывая this.forceUpdate() только тех компонентов в которых поменялись данные которые они выводят.

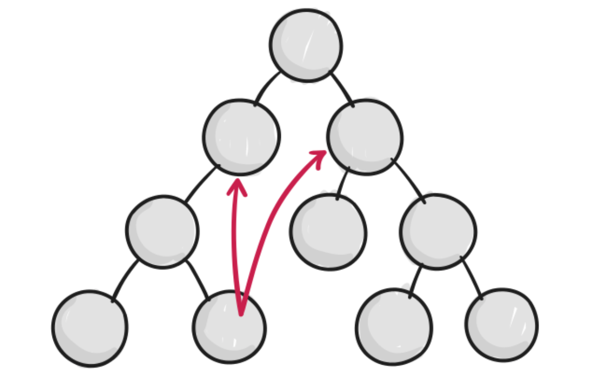
И вот эту проблему как раз и решает библиотека mobx.

Более подробно разобраться как работает mobx в Приложении 1.

**Redux**

Redux — это инструмент управления как состоянием данных, так и состоянием интерфейса в JavaScript-приложениях. Он подходит для одностраничных приложений, в которых управление состоянием может со временем становиться сложным. Redux не связан с каким-то определенным framework-ом, и хотя разрабатывался для React, может использоваться с Angular или jQuery.

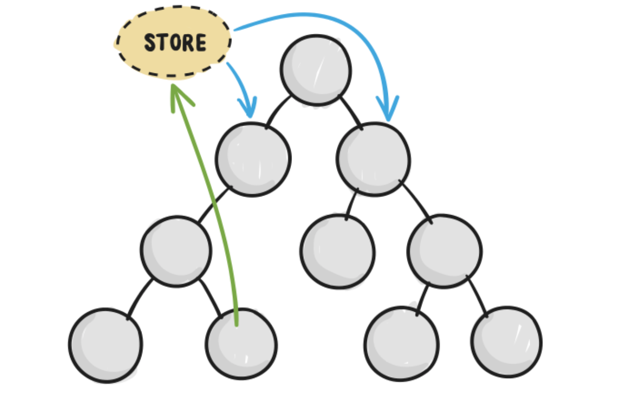
Данные в React текут через компоненты. Это называется однонаправленный поток данных — поток данных проходит в одном направлении, от родителя к ребенку. При этом не очевидно, как два компонента, не связанные отношением родитель-ребенок, будут взаимодействовать между собой:



В React не рекомендуется реализовывать прямое взаимодействие компонент-компонент. Это считается плохой практикой, приводит к ошибкам и спагетти-коду — старый термин для запутанного кода.

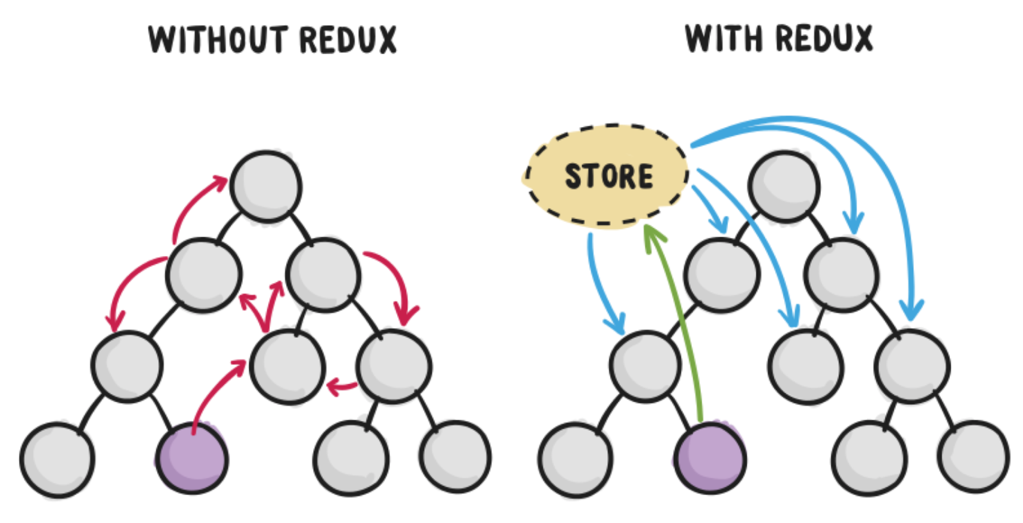
Как рекомендуют разработчики React – для взаимодействия между двумя компонентами, не имеющими связи, родитель-ребенок, можно настроить глобальную систему событий. Flux — это один из способов достижения этого (об этом чуть позже).

Вот здесь пригождается Redux. Redux предлагает хранить все состояние приложения в одном месте, называемом store (хранилище). Компоненты отправляют изменение состояния в хранилище, а не напрямую другим компонентам. Компоненты, которые должны быть в курсе этих изменений, подписываются на хранилище:



Хранилище может рассматриваться как посредник во всех изменениях состояния в приложении. С Redux компоненты не связываются друг с другом напрямую, все изменения должны пройти через единственный источник истины, через хранилище.

Это сильно отличается от других стратегий, где части приложения взаимодействуют непосредственно друг с другом:



C Redux все компоненты получают свое состояние из хранилища. Также ясно, куда компонент должен отправить информацию об изменении состояния — опять же в хранилище. Компонент только инициирует изменение и не заботится об остальных компонентах, которые должны получить это изменение. Таким образом, Redux делает поток данных более понятным.

Общая концепция использования хранилищ для координации состояния приложения — это шаблон, известный как Flux. Этот шаблон проектирования дополняет однонаправленный поток данных как в React. Redux напоминает Flux, но так ли они близки об этом далее.

**React Context API**

Более новые версии React (16.3+) включают встроенный способ совместного использования состояния, что означает отсутствие необходимости использовать внешнюю библиотеку. Данный способ известен как React Context API.

Рассмотрим Context API на сценарии панели инструментов, где пользователь может обновить свое имя. Имя пользователя отображается по всей панели инструментов, что означает, что имя пользователя будет сохранено в состоянии компонента, а затем передано другим компонентам через **props**.

Без Context API это выглядит так:

class Dashboard extends React.Component {

state = { username: '' }; render() {

return (

<div>

<welcomemessage username="{this.state.username}">

<settingsform

username="{this.state.username}"

updateusername="{

newUsername" ==""> {this.setState({ username: newUsername });

}}

/>

</settingsform></welcomemessage></div>

);

}

}

Имя пользователя сохраняется в состоянии компонента **Dashboard**, а затем передается с помощью this.state.usernameв оба компонента. Дополнительный параметр передается в форму для обновления состояния, которое затем повторно отображает информационную панель с новым именем пользователя.

Добавим на панель инструментов больше компонентов, которые глубоко вложены.

<dashboard>

<welcomemessage>

<messagelist>

<usermessage>

<p>Need to show username here...</p>

</usermessage>

</messagelist>

</welcomemessage>

</dashboard>

Чтобы передать ему имя пользователя, нам нужно сделать то, что называется **prop drilling**:

<dashboard>

<welcomemessage username="{this.state.username}">

<messagelist username="{this.props.username}">

<usermessage>

<p>Hello {this.props.username}!</p>

</usermessage>

</messagelist>

</welcomemessage>

</dashboard>

Это может стать чрезвычайно утомительным, так как мы добавляем больше состояний и вложенных компонентов. Кроме того, есть шанс, что нам нужно получить доступ к имени пользователя за пределами панели мониторинга.

С использованием встроенного React Context API можно избежать подробного пробрасывания параметров, что позволит компоненту иметь прямой доступ к состоянию имени пользователя, которое было изначально сохранено в компоненте.

Рассмотрим создание поставщиков и потребительских компонентов.

Начнем с создания файла для вашего контекста. Создаем **user-context.js**:

import React, { createContext } from 'react';

const UserContext = createContext({

username: '',

updateUsername: () => {},

});

export class UserProvider extends React.Component {

updateUsername = newUsername => {

this.setState({ username: newUsername });

};

state = {

username: 'user',

updateUsername: this.updateUsername,

};

render() {

return (

<usercontext.provider value="{this.state}">

{this.props.children}

</usercontext.provider>

);

}

}

export const UserConsumer = UserContext.Consumer;

Сначала пользовательский контекст создается с использованием **createContext()**. Здесь значения будут переопределены **UserProvider**.

Затем мы создаем компонент **UserProvider**, который будет служить родительским компонентом для хранения и управления общим состоянием. Это как о эквивалент компонента в нашем самом раннем примере.

Далее экспортируем компонент **UserConsumer**, который позволит компонентам получить доступ к общему состоянию.

Использование провайдера. Компонент должен охватывать все компоненты, которые совместно используют состояние. Самый простой способ - добавить его в основной компонент приложения, который обычно отображается в DOM от React.

import React from 'react';

import ReactDOM from 'react-dom';

import UserMessage from './UserMessage';

import SettingsForm from './SettingsForm';

import { UserProvider } from './user-context';

function App() {

return (

<userprovider>

<usermessage>

<settingsform>

</settingsform>

</usermessage>

</userprovider>

);

}

const rootElement = document.getElementById('root');

ReactDOM.render(<app>, rootElement);</app>

Импортируем два других компонента: **UserMessage** и **SettingsForm**. Эти два компонента будут иметь доступ к общему состоянию пользователя.

Использование потребителя для чтения состояния. Одним из вариантов использования общего состояния является его отображение. В этом случае отобразим текущее имя пользователя. Создайте файл с именем **UserMessage.js** и добавим в него следующее:

import React from 'react';

import { UserConsumer } from './user-context';

export default function UserMessage() {

return (

<userconsumer>

{({ username }) => <h1>Welcome {username}!</h1>}

</userconsumer>

);

}

В этом файле создали компонент **UserMessage**, который отображает сообщение Welcome username. Имя пользователя извлекается из компонента **UserConsumer**, который экспортируется из **user-context.js**.

Внутри передаем то, что называется реквизитом рендера. В данном случае это функция, в которой можем захватить части состояния и что-то сделать с ним.

Использование потребителя для обновления состояния. Другой вариант использования общего состояния - обновить его. В этом случае предоставим пользователю форму для обновления своего имени пользователя. Создадим файл с именем **UserSettings.js** и добавьте в него следующее:

import React from 'react';

import { UserConsumer } from './user-context';

export default function UserSettings() {

return (

<userconsumer>

{({ updateUsername }) => (

<div>

<h2>Settings</h2>

<label htmlfor="username">Username: </label>

<input id="username" type="text" onchange="{event" ==""> {

updateUsername(event.target.value);

}}/>

</div>

)}

</userconsumer>

);

}

Это похоже на предыдущий пример, за исключением того, что вместо получения имени пользователя используем функцию **updateUsername** для его обновления.

**Приложение 1**

Попробуем решить задачу точечного обновления компонентов, не беря во внимания Mobx.

Тут можно придумать два подхода и оба они будут накладывать ограничения на то, как мы работаем с состоянием.

Первый подход — это воспользоваться иммутабельностью и двоичным поиском — если каждое обновление состояния будет возвращать новые объекты данных которые изменились и всех родительских объектов (для случая когда состояние имеет иерархическую структуру) то тогда мы можем добиться почти точечного обновления компонентов путем сравнения ссылок на предыдущее и новое состояние и пропускать все поддеревья компонентов данные которых не изменились (newSubtree === oldSubtree) и в результате мы обновим наше приложение вызвав pre-render только нужных компонентов сравнив при этом данные только O(log(n)) компонентов где n — это количество компонентов.

Так, например работает Angular, если выставить ему настройку ChangeDetectionStrategy.OnPush. Но у решения спуска сверху-вниз есть пара недостатков. Во первых — несмотря на эффективность O(log(n)), если какой-то компонент выводит список других компонентов, то мы вынуждены пробежаться по всему массиву компонентов, чтобы у них сравнить их пропсы, и, если каждый компонент списка render-ит еще один список, то количество сравнений еще больше возрастает.

Во-вторых — компонент должен зависеть только от своих props-ов, которые часто приходится прокидывать вложенным компонентам через промежуточные.

Также иммутабельный подход применяет и библиотека redux, но только слегка в измененном виде, решая недостаток с зависимостью только от props-ов. Помимо сравнения props-ов, redux сравнивает также и дополнительные данные которые вернула функция mapStateToProps() (в connect декораторе), в которой мы указываем зависимость от разных частей состояния и дальше они становятся дополнительными props-ми. Но для этого redux вынужден выполнить проверку всех n подключенных компонентов. Но даже это все равно быстрее чем делать обновление (ReactDOM.render(<App>, “root”)) всего приложения.

Но у иммутабельного подхода есть пара серьезных недостатков, которые накладывают ограничения на работу с состоянием.

Первый недостаток — это то, что мы не можем теперь просто взять и обновить любое свойство объекта данных в приложении. Из-за требования возвращать каждый раз новый иммутабельный объект целого состояния, нам нужно вернуть новый объект и также пересоздать все родительские объекты и массивы. Например, если объект состояния хранит массив проектов, каждый проект хранит массив задач, и каждая задача хранит массив комментариев:

let AppState = {

projects: [

{..},

{...},

{name: 'project3', tasts: [

{...},

{...},

{name: 'task3', comments: [

{...},

{...},

{text: 'comment3' }

]}

]}

]

}

Для того чтобы обновить текст у объекта комментария мы не можем просто выполнить comment.text = 'new text' — нам нужно выполнить сначала пересоздание объекта комментария (comment = {...comment, text: 'updated text'}), дальше нужно пересоздать объект задачи и скопировать у туда ссылки на другие комментарии (task = {...task, tasks: [...task.comments]}), дальше пересоздать объект проекта и скопировать туда ссылки на другие задачи (project = {...project, tasks: [...project.tasks]}) и в конце уже пересоздать объект состояние и также скопировать ссылки на другие проекты (AppStat = {...AppState, projects: [...AppState.projects]}).

Второй недостаток — это невозможность хранить в состоянии объекты, которые ссылаются друг на друга. Если нам где-то в обработчике компонента нужно получить проект в котором он находится задача — то мы не можем при создании объекта просто присвоить ссылку на родительский проект — task.project = project, потому, что необходимость при иммутабельном подходе возвращать новый объект не только задачи, но и проекта приводит к тому что нам нужно обновить все остальные задачи в проекте — ведь ссылка на объект проекта поменялась, а значит нужно выполнить обновление всех задач, присвоив новую ссылку, а обновление как мы знаем нужно выполнить через пересоздание объекта, а если задачи хранят комментарии, нам нужно выполнить пересоздание всех комментариев, потому что они хранят ссылку на объект задачи, и так рекурсивно мы придем к пересозданию всего состояния и это будет ужасно медленно.

В итоге нам приходится либо каждый раз изменять props вышестоящих компонентов чтобы передать нужный объект, либо вместо ссылок на объект сохранить id task.project = '12345'; а потом где-то хранить и поддерживать hash проектов по их id ProjectHash['12345'] = project;

Поскольку решение с иммутабельностью имеет кучу недостатков давайте подумаем можно ли решить задачу точечного обновления компонентов другим способом? Нам нужно при изменении данных в приложении выполнить pre-render только тех компонентов, которые зависят от этих данных. Что значит зависят? Например, есть простой компонент комментария, который render-ит текст комментария:

class Comment extends React.Component {

render(){

const {comment} = this.props;

return <div>{comment.text}</div>

}

}

Этот компонент зависит от comment.text и его нужно обновить каждый раз, когда меняется comment.text. Но также если компонент выводит <div>{comment.parent.text}</div>, но теперь нужно обновлять компонент каждый раз когда изменится не только .text, но и .parent. Решить эту задачу мы можем, не применяя никакого иммутабельного подхода, а задействовав возможности геттеров и сеттеров javascript и это второй из известных подходов решить задачу точечного обновления ui.

Геттеры и сеттеры — это довольно старая возможность javascript поставить свой обработчик на обновление свойства или получение значение свойства:

Object.defineProperty(comment, 'text', {

get(){

console.log('>text getter');

return this.\_text;

},

set(val){

console.log('>text setter');

this.\_text = val;

}

})

comment.text; // выведет в консоль >text getter

comment.text = 'new text' // выведет в консоль >text setter

Итак, можем поставить на сеттер функцию, которая будет выполнятся каждый раз, когда выполняется присвоение нового значение и будем вызывать pre-render списка компонентов, которые зависят от этого свойства. Для того чтобы узнать какие компоненты от каких свойств зависят нужно перед в начале функции render() компонента присвоить в некую глобальную переменную текущий компонент, а при вызове геттера любого свойства объекта нужно добавить в список зависимостей этого свойства текущий компонент, который находится в глобальной переменной. И поскольку компоненты могут redner-ся древовидно надо еще не забывать возвращать назад в эту глобальную переменную в предыдущий компонент.

let CurrentComponent;

class Comment extends React.Component {

render(){

const prevComponent = CurrentComponent;

CurrentComponent = this;

const {comment} = this.props;

var result = <div>{comment.text}</div>

CurrentComponent = prevComponent;

return result

}

}

comment.\_components = [];

Object.defineProperty(comment, 'text', {

get(){

this.\_components.push(CurrentComponent);

return this.\_text

},

set(val){

this.\_text = val;

this.\_components.forEach(component => component.setState({}))

}

})

При таком подходе каждое свойство будет хранить массив своих зависимых компонентов и при изменении свойства будет вызывать их обновление.

Теперь для того чтобы не смешивать хранение массива зависимых компонентов с данными и для упрощения кода вынесем логику такого свойства в класс Cell, который, как можно понять из аналогии, очень похож на принцип работы ячеек в Excel — если другие ячейки содержат формулы от которых зависит текущая ячейка, то нужно при изменении значения вызвать обновления всех зависимых ячеек.

let CurrentObserver = null;

class Cell {

constructor(val){

this.value = val;

this.reactions = new Set();

//для простоты и скорости воспользуемся классом

//множества из es6 стандарта

}

get(){

if(CurrentObserver){

this.reactions.add(CurrentObserver);

}

return this.value;

}

set(val){

this.value = val;

for(const reaction of this.reactions){

reaction.run();

}

}

unsibscribe(reaction){

this.reactions.delete(reaction);

}

}

А вот роль ячейки c формулой будет играть класс ComputedCell который наследуется от класса Cell (потому что от этой ячейки может зависеть и другие ячейки). Класс ComputedCell принимает в конструкторе функцию (формулу) для пересчета и также опционально функцию для выполнения сайд-эффектов (как например вызов .forceUpdate() компонентов)

class ComputedCell extends Cell {

constructor(computedFn, reactionFn, ){

super(undefined);

this.computedFn = computedFn;

this.reactionFn = reactionFn;

}

run(){

const prevObserver = CurrentObserver;

CurrentObserver = this;

const newValue = this.computedFn();

if(newValue !== this.value){

this.value = newValue;

CurrentObserver = null;

this.reactionFn();

this.reactions.forEach(r=>r.run());

}

CurrentObserver = prevObserver;

}

}

А теперь для выполнения каждый раз установки геттеров и сеттеров воспользуемся декораторами из typescript или babel. Да, это накладывает ограничения на необходимость использование классов и создание объектов не через литерал const newComment = {text: 'comment1'}, а через const comment = new Comment('comment1'), но зато вместо ручной установки геттеров и сеттеров мы можем удобно пометить свойство как @observable и дальше работать с ним как с обычным свойством.

class Comment {

@observable text;

constructor(text){

this.text = text;

}

}

function observable(target, key, descriptor){

descriptor.get = function(){

if(!this.\_\_observables) this.\_\_observables = {};

const observable = this.\_\_observables[key];

if (!observable) this.\_\_observables[key] = new Observable()

return observable.get();

}

descriptor.set = function(val){

if (!this.\_\_observables) this.\_\_observables = {};

const observable = this.\_\_observables[key];

if (!observable) this.\_\_observables[key] = new Observable()

observable.set(val);

}

return descriptor

}

А для того чтобы не работать напрямую с классом ComputedCell внутри компонента, можем вынести этот код в декоратора @observer, который просто оборачивает метод render() и создает при первом вызове вычисляемую ячейку, передавая в качестве формулы метод render(), а в качестве функции-реакции вызов this.forceUpdate() (в реальности нужно еще добавить отписку в методе componentWillUnmount() и некоторые моменты правильного оборачивания компонентов react, но оставим пока для простоты понимания такой вариант):

function observer(Component) {

const oldRender = Component.prototype.render;

Component.prototype.render = function(){

if (!this.\_reaction) this.\_reaction = new ComputedCell(oldRender.bind(this), ()=>this.forceUpdate());

return this.\_reaction.get();

}

}

И будем использовать как:

@observer

class Comment extends React.Component {

render(){

const {comment} = this.props;

return <div>{comment.text}</div>

}

}

В примере есть один недостаток — что, если зависимости компонента могут меняться? Взглянем на следующий компонент

class User extends React.Component {

render(){

const {user} = this.props;

return <div>{user.showFirstName ? user.firstName : user.lastName}</div>

}

}

Компонент зависит от свойства user.showFirstName и дальше в зависимости от значение может зависеть либо от user.firstName, либо от user.lastName, то есть если user.showFirstName == true, то мы не должны реагировать на изменение user.lastName и наоборот если user.showFirstName поменялось на false, то мы не должны реагировать (и делать pre-render компонента) если меняется свойство user.firstName;

Этот момент легко решается путем добавления списка зависимостей this.dependencies = new Set() в класс ячейки и небольшой логики в функцию run() — чтобы после вызова render() react сравнили предыдущий список зависимостей с новым и отписались от неактуальных зависимостей.

class Cell {

constructor(){

...

this.dependencies = new Set();

}

get() {

if (CurrentObserver) {

this.reactions.add(CurrentObserver);

CurrentObserver.dependencies.add(this);

}

return this.value;

}

}

class ComputedCell {

track(){

const prevObserver = CurrentObserver;

CurrentObserver = this;

//сохраняем список текущих зависимостей

const oldDependencies = this.dependencies;

//заменяем на пустое множество,

// в которое будут добавляться новые зависимости

this.dependencies = new Set();

const newValue = this.computedFn();

// отписываемся от зависимостей которых нет в новом списке

for(const dependency of oldDependencies){

if(!this.dependencies.has(dependency)){

dependency.unsubscribe(this);

}

}

CurrentObserver = prevObserver;

return newValue;

}

}

Второй момент — что, если мы сразу меняем много свойств в объекте? Поскольку зависимые компоненты будут обновляться синхронно мы получим два лишних обновления компонента:

comment.text = 'edited text'; //произойдет первый перередер компонента

comment.editedCount+=1; //будет второй перерендер компонента

Чтобы избежать лишних обновлений, в начале этой функции мы можем поставить глобальных флаг, а наш @observer декоратор не будет сразу вызывать this.forceUpdate(), а вызовет только тогда когда мы уберем этот флаг. И для упрощения мы вынесем эту логику в декоратор action и вместо флага будем увеличивать или уменьшать счетчик, потому что декораторы могут вызываться внутри других декораторов.

updatedComment = action(()=>{

comment.text = 'edited text';

comment.editedCount+=1;

})

let TransactionCount = 0;

let PendingComponents = new Set();

function observer(Component) {

const oldRender = Component.prototype.render;

Component.prototype.render = function(){

if (!this.\_reaction) this.\_reaction = new ComputedCell(oldRender.bind(this), ()=>{ TransactionCount ?PendingComponents.add(this) : this.forceUpdate() });

return this.\_reaction.get();

}

}

function action(fn){

TransactionCount++

const result = fn();

TransactionCount--

if(TransactionCount == 0){

for(const component of PendingComponents){

component.forceUpdate();

}

}

return result;

}

В итоге такой подход c использованием очень старого паттерна observer (не путать с observable RxJS) намного лучше подходит для реализации задачи точечного обновления компонентов чем подход с использованием иммутабельности.

Из недостатков можно заметить только необходимость создавать объекты не через литералы, а через классы, а это значит, что мы не можем просто принять какие-то данные от сервера и передать компонентам — необходимо провести дополнительную обработку данных оборачивая в объекты классов с @observable декораторами.

Также к недостаткам можно записать невозможность добавлять новые свойства к объектам на лету (хотя это и так считается анти-паттерном с точки зрения производительности js), неудобства отладки кода в chrome-devtools, потому что данные скрыты за геттерами и вместо значений мы будем видеть три точки и чтобы увидеть значение на кликнуть на это свойство, и также попытка выполнить по шагам любое изменение или получение свойства будет переносить нас в глубь сеттера или геттера внутри библиотеки.

Но достоинства таковы. Во первых — в отличии от иммутабельного подхода скорость работы никак не зависит от количества компонентов потому, что мы сразу знаем список компонентов которые надо обновить — а значит имеем сложность o(1) вместо o(log(n)) или o(n) как заметил [Ден Абрамов](https://twitter.com/dan_abramov/status/719973322453348352) и что более важно — не происходит создание n-объектов в функции mapStateToProps. Во-вторых — когда нам нужно обновить какие-то данные мы можем просто написать comment.text = 'new text' и нам не придется выполнять еще кучу работы по обновлению родительских объектов состояния, и что важно — не будет нагрузки на сборщике мусора из-за постоянного пересоздания объектов. Ну и главное — мы можем моделировать состояния с помощью объектов которые ссылаются друг на друга и сможем удобно работать с состоянием без необходимости хранить вместо объекта id, а потом вытаскивать каждый раз из:

hash.AppState.folders[AppState.projects[AppState.tasks[comment.taskId].projectId].folderId].name

вместо простого обращения по ссылке comment.task.project.folder.name