Хуки

**Жизненный цикл**

Каждый компонент проходит какие-то стадии, и контролировать эти стадии или узнавать о них позволяют жизненные циклы.

Жизненных циклов бывает: базовым, изменения и удаления.

**Базовый жизненный цикл**

Базовый жизненный цикл – это цикл создания самого компонента.

* componentWillMount сообщает, что компонент инициализирован и необходимые параметры собраны
* render формирует JSX для внедрения его в DOM
* componentDidMount сообщает, что компонент уже полностью создан и его можно уже внедрить

Первый практически не используется, а в componentDidMount используется, когда нужно изменить state и/или получить данные с сервера.

import React from 'react'

class Car extends React.Component {

constructor(props) {

super(props)

this.state = {}

}

componentWillMount() {

alert('will mount')

}

componentDidMount() {

alert('did mount')

}

inputHandle = event => {

this.props.setName(event.target.value)

}

render() {

alert('render')

return <div>

<h1>{this.props.name}</h1>

<input type='text' onChange={this.inputHandle} />

<h1>{this.props.year}</h1>

</div>

}

}

export default Car

**Жизненный цикл изменения**

* componentWillReceiveProps означает поступление новых параметров и создан, чтоб синхронизировать данные. Используется редко.
* shouldComponentUpdate должен вернуть true, если компонент нужно обновить, иначе false
* componentWillUpdate означает, что компонент точно изменится
* render составляет новый JSX
* componentDidUpdate означает, что компонент обновлен и, что его можно внедрить

import React from 'react'

class Car extends React.Component {

constructor(props) {

super(props)

this.state = {}

}

componentWillReceiveProps(nextProps) {

console.log(nextProps)

alert('will receive props')

}

shouldComponentUpdate(nextProps, nextState) {

console.log(nextProps)

console.log(nextState)

alert('should update')

return true

}

componentWillUpdate(nextProps, nextState) {

console.log(nextProps)

console.log(nextState)

alert('will update')

}

componentDidUpdate() {

alert('did update')

}

inputHandle = event => {

this.props.setName(event.target.value)

}

render() {

alert('render')

return <div>

<h1>{this.props.name}</h1>

<input type='text' onChange={this.inputHandle} />

<h1>{this.props.year}</h1>

</div>

}

}

export default Car

**Жизненный цикл удаления**

Жизненный цикл удаления состоит из одного, но полезного метода componentWillUnmount, который означает, что компонент готов удалению, так что в нем нужно очищать таймеры, сохранять настройки и т.д.

import React from 'react'

class Car extends React.Component {

constructor(props) {

super(props)

this.state = {}

}

componentWillUnmount() {

alert('will unmount')

}

inputHandle = event => {

this.props.setName(event.target.value)

}

render() {

alert('render')

return <div>

<h1>{this.props.name}</h1>

<input type='text' onChange={this.inputHandle} />

<h1>{this.props.year}</h1>

</div>

}

}

export default Car

**Замена для componentWillReceiveProps и componentWillUpdate**

Эти два метода, по сути, позволяют синхронизировать состояние, но они делают это напрямую, что плохо. В новых версиях это исправили, тем что ввели новый статический метод getDerivedStateFromProps.

Он получает новые параметры и прошлое состояние, и он должен вернуть объект, который после просто добавится в состояние. Использование его с устаревшими методами влечет к ошибке.

import React from 'react'

class Car extends React.Component {

constructor(props) {

super(props)

this.state = {}

}

shouldComponentUpdate(nextProps, nextState) {

console.log(nextProps)

console.log(nextState)

alert('should update')

return true

}

static getDerivedStateFromProps(nextProps, prevState) {

alert('derived state from props')

return prevState

}

componentDidUpdate() {

alert('did update')

}

inputHandle = event => {

this.props.setName(event.target.value)

}

render() {

alert('render')

return <div>

<h1>{this.props.name}</h1>

<input type='text' onChange={this.inputHandle} />

<h1>{this.props.year}</h1>

</div>

}

}

export default Car

**Метод getSnapshotBeforeUpdate**

Метод getSnapshotBeforeUpdate вызывается после render, но перед componentDidUpdate, поэтому он может работать с еще не обновленным состоянием. Это полезно, ведь он позволяет сохранить текущий скролл пользователя, момент остановки видео и т.д.

Он должен вернуть что-то или null, а после что он вернул можно получить в componentDidMount третьим параметром.

import React from 'react'

class Car extends React.Component {

constructor(props) {

super(props)

this.state = {}

}

shouldComponentUpdate(nextProps, nextState) {

console.log(nextProps)

console.log(nextState)

alert('should update')

return true

}

static getDerivedStateFromProps(nextProps, prevState) {

alert('derived state from props')

return prevState

}

getSnapshotBeforeUpdate() {

alert('snapshot before update')

return null

}

componentDidUpdate(prevProps, prevState, snapshot) {

console.log(snapshot)

alert('did update')

}

inputHandle = event => {

this.props.setName(event.target.value)

}

render() {

alert('render')

return <div>

<h1>{this.props.name}</h1>

<input type='text' onChange={this.inputHandle} />

<h1>{this.props.year}</h1>

</div>

}

}

export default Car

**Жизненный цикл ошибки**

Жизненный цикл ошибок состоит из метода componentDidCatch, который позволяет обработать ошибку. В параметры он получает объект ошибку и информацию.

Благодаря нему можно красиво отображать ошибки. Это можно сделать, если установить этот метод в компоненте высшего порядка, который если будет ошибка будет отображать какое-то предупреждение.

import React, { Component } from 'react'

class ErrorBoundary extends Component {

state = {

hasError: false

}

componentDidCatch(error, info) {

this.setState({

hasError: true

})

}

render() {

return !this.state.hasError

? <h1>Error</h1>

: this.props.children

}

}

export default ErrorBoundary

**Хуки**

Жизненные циклы могут быть в классах, но в функциональных компонентах их нет, поэтому в React 16.8 добавили хуки.

Хуки позволяют создать состояние, воспользоваться контекстом, создать референцию и т.д. в функциональных компонентах. Есть всего 3 основных хуков: состояния, эффекта и контекста.

Классы сами по себе затратные и, если класс слишком большой, то его становится трудно понимать, поэтому создавать классы-компоненты можно использовать при создании корневого компонента, а все остальные следует сделать на компонентах функциях.

**Хук состояния**

Хук состояния позволяет создать состояние:

import React, { useState } from 'react'

function App() {

const [name, setName] = useState('Вася Пупкин')

function inputHandler(event) {

setName(event.target.value)

}

return <div className='App'>

<h1>Hello { name }</h1>

<input type='text' onChange={inputHandler} value={name}/>

</div>

}

export default App

Хук useState возвращает массив со свойством и функцией для изменения этого свойства. Свойству можно установить значение по-умолчанию, передав его в функцию.

В отличие от обычного состояния:

* свойство может быть и не объектом
* функция по изменению не соединяет значения
* можно использовать неограниченно раз

Хук useState в разы удобнее обычного state.

**Хук эффекта**

Эффект – это побочные изменения, которые нельзя выполнить во время рендеринга. Например:

* сетевые запросы
* логирование
* ручное изменение DOM
* подписки к данным

Хук эффекта вызывается сразу после рендеринга компонента, поэтому он является заменой componentDidMount, componentDidUpdate и componentWillUnmount.

import React, { useState, useEffect } from 'react'

function App() {

const [name, setName] = useState('Вася Пупкин')

useEffect(() => {

document.title = 'Hello ' + name

})

function inputHandler(event) {

setName(event.target.value)

}

return <div className='App'>

<h1>Hello { name }</h1>

<input type='text' onChange={inputHandler} value={name}/>

</div>

}

export default App

Здесь хук эффекта изменяет title при каждом обновлении компонента.

Это пример эффекта, который не использует сброс, т.е. его можно выполнить и забыть. К таким эффектам относят сетевые запросы, логирование и т.д. Хуки без сброса заменяют componentDidMount и componentDidUpdate.

Эффекты со сбросом – это эффекты, которые возвращают функцию, которая будет вызваны, как componentWillUnmount и перед рендером.

function FriendStatusWithCounter(props) {

const [count, setCount] = useState(0)

useEffect(() => {

document.title = `Вы нажали ${count} раз`

})

const [isOnline, setIsOnline] = useState(null)

useEffect(() => {

function handleStatusChange(status) {

setIsOnline(status.isOnline)

}

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange)

return () => {

ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange)

}

})

}

**Пользовательский хук**

Пользователь может создать собственный хук, который повторяет какую-то логику. Пользовательский хук должен начинается use.

Например, создадим хук для контроля над input:

import React, { useState } from 'react'

function useInputValue(defaultValue='') {

let [value, setValue] = useState(defaultValue)

return {

bind: {

value,

onChange: event => setValue(event.target.value)

},

clear: () => setValue(''),

value: () => value

}

}

function Form({ onCreate }) {

let input = useInputValue()

function submitHandler(event) {

event.preventDefault()

if (input.value().trim()) {

onCreate(input.value())

input.clear()

}

}

return (

<form onSubmit={submitHandler}>

<input {...input.bind}/>

<button type='submit'>Add </button>

</form>

)

}

export default Form

Здесь хук внутри создает свойство value и вокруг него создает объект, который позволяет очищать, привязывать и получать это свойство.

Преимущество пользовательских хуков, в том, что их можно неограниченно переиспользовать, ибо они повторяют логику состояния, а не само состояние.

**Хук контекста**

Получить свойства из контекста внутри функциональных компонентов можно через хук useContext:

function Button(props) {

const context = useContext(Context)

return <button>{props.text} {context}</button>

}