

Хуки

Хуки 1

useState	1
useEffect	2
useRef	3
useContext	6
useMemo	9
useCallback	10

useState

Пример счетчика с начальным рандомным значением:

```
function getRandomValue() {  
  console.log('getRandomValue');  
  return Math.floor(Math.random() * 50)  
}
```

```
function App() {  
  const [counter, setCounter] = useState(() => {return getRandomValue()})
```

! Если начальное значение рассчитывается из функции, то, чтобы она не запускалась каждый раз при изменении состояния, нужно сделать для нее обертку.

```
  const resetCounter = () => {  
    setCounter(0) // устанавливаем значение counter  
  }  
  
  const increaseCounter = () => {  
    setCounter((prev) => {  
      return prev + 1 // изменяем значение counter, основываясь на прежнем его значении  
    })  
  }  
  
  const decreaseCounter = () => {  
    setCounter((prev) => {  
      return prev - 1 // изменяем значение counter, основываясь на прежнем его значении  
    })  
  }  
  return (  
    <div className='App'>  
      <h2>Counter: {counter}</h2>  
      <button onClick={increaseCounter}>Increase</button>  
      <button onClick={decreaseCounter}>Decrease</button>  
      <button onClick={resetCounter}>Reset</button>  
    </div>  
  );  
}
```

```
export default App;
```

Пример поста, у кот орого независимо друг от друга можем менять название и описание:

```
function App() {  
  // Пост в блоге  
  const [post, setPost] = useState({  
    title: 'Название поста',  
    desc: 'Описание поста.'  
  })  
  
  const changeTitle = (e) => {
```

```

        setPost((prev)=>{
            return {
                ...prev,
                title: e.target.value
            }
        })
    }

    const changeDesc = (e) => {
        setPost((prev)=>{
            return {
                ...prev,
                desc: e.target.value
            }
        })
    }

    return (
        <div className='App'>
            <h2>{post.title}</h2>
            <p>{post.desc}</p>
            <input onChange={changeTitle} value={post.title} type="text" placeholder="Введите название" /><br />
            <textarea value={post.desc} onChange={changeDesc} placeholder="Введите описание"></textarea>
        </div>
    );
}

```

useEffect

Основная информация

Имеет два аргумента. Функция внутри хука (первый аргумент) вызывается каждый раз, когда происходит отрисовка данного компонента на странице. При вызове хука `useEffect` мы можем передавать зависимости (второй аргумент). Если передан пустой массив, хук будет вызываться только при первичном рендере. При зависимостях от определенных состояний, нужно эти состояния перечислить в массиве, во втором аргументе.

```

import { useEffect, useState } from "react";

function App() {
    const [counter, setcounter] = useState(0);
    const increase = () => setcounter((prev) => prev + 1);

    useEffect(() => {
        console.log("useEffect run!");
    }, [counter]);

    return (
        <div className="App">
            <h1>useEffect</h1>
            <h3>Counter: {counter}</h3>
            <button onClick={increase}>Increase</button>
        </div>
    );
}

```

Очистка side эффектов запущенных в useEffect

Хотим повесить на хук прослушку события изменения размера окна.

```

function App() {
    const [windowWidth, setwindowWidth] = useState(4);

    const handleResize = () => {
        console.log('handleResize run');
        setwindowWidth(window.innerWidth)
    }

    useEffect(()=>{
        window.addEventListener('resize', handleResize)
        return () => {

```

```

        console.log('Remove resize listener');
        window.removeEventListener('resize', handleResize)
    }
}, [])

return (
    <div className='App'>
        <h1>useEffect</h1>
        <h3>Window width: {windowWidth}</h3>
    </div>
);
}

```

Здесь listener создается только один раз при первичном рендеринге. Есть один нюанс. Когда этот компонент **будет размонтирован**, н-р, при роутере, то **прослушка останется**. Поэтому ее нужно убрать в момент размонтирования. Хук может **возвращать функцию**, которая будет запущена в **момент размонтирования**.

useEffect u fetch

Этот хук используется часто, чтобы вставлять в него код для получения данных с сервера, н-р, с помощью **fetch**

```

function App() {
    const [todoTitle, setTodoTitle] = useState('Watch react tuts');

    function getRandomValue (max){
        return Math.floor(Math.random() * max) + 1
    }

    useEffect(() => {
        fetch(`https://jsonplaceholder.typicode.com/todos/${getRandomValue(200)}`)
            .then(response => response.json())
            .then(data => setTodoTitle(data.title))
    }, [])

    return (
        <div className='App'>
            <h1>useEffect</h1>
            <hr />
            <p>Random ToDo item: <strong>{todoTitle}</strong></p>
        </div>
    );
}

```

! Частая ошибка, когда useEffect **выполняется бесконечно** - это, когда **внутри него обновляется состояние**. Здесь, это будет так, если убрать второй аргумент. В этом случае нужно грамотно задать зависимости

useRef

Используется для работы с **dom-элементами**. Возвращает **объект со свойством current**, в который записано текущее значение ref (**{current: текущий_элемент}**). Чтобы связать его с dom-элементом, в элементе нужно добавить атрибут **ref** со значением - этим объектом.

Рассмотрим пример, когда при нажатии на кнопку selectName фокус нужно переместить в поле input:

```

import { useEffect, useRef } from "react";

function App() {
    const nameRef = useRef(null); // установили стартовое значение и записали в переменную

    const selectName = () => {
        nameRef.current.focus();
    };

    return (
        <div className="App">
            <h1>useRef</h1>
            <input ref={nameRef} type="text" placeholder="Введите имя" />
            <button onClick={selectName}>selectName</button>
        </div>
    );
}

```

```
}
```

useRef ререндер

useRef используется для того, чтобы задать какое-то значение внутри компонента, которое используется в течение всего жизненного цикла компонента и изменение этого значения не будет приводить к ререндеру компонента.

Рассмотрим пример, когда мы хотим вывести количество ререндеров компонента. useState вместе с useEffect не подходит для этого, так как в useEffect используется расчет состояния в зависимости от предыдущего состояния, и мы уходим в бесконечный цикл. используем useRef.

```
function App() {
  //const [renderCount, setRenderCount] = useState(1);
  const renderCount = useRef(1);
  const [name, setName] = useState(""); // для того, чтобы сделать ререндер
  const nameRef = useRef(null);

  const selectName = () => {
    nameRef.current.focus();
  };

  const changeName = (e) => {
    setName(e.target.value);
  };

  useEffect(() => {
    //setRenderCount((prev) => prev + 1);
    renderCount.current = renderCount.current + 1;
  });

  return (
    <div className="App">
      <h1>useRef</h1>
      <input onChange={changeName} ref={nameRef} type="text" placeholder="Введите имя" />
      <button onClick={selectName}>selectName</button>
      <hr />
      <p>render Count: {renderCount.current}</p>
      <hr />
      <h3>name: {name}</h3>
    </div>
  );
}
```

prevValue

useRef можно использовать для сохранения предыдущего состояния.

```
function App() {
  const renderCount = useRef(1);
  const [name, setName] = useState("");
  const nameRef = useRef(null);
  const prevName = useRef("");

  const selectName = () => {
    nameRef.current.focus();
  };

  const changeName = (e) => {
    prevName.current = name;
    setName(e.target.value);
  };

  useEffect(() => {
    renderCount.current = renderCount.current + 1;
  });

  return (
    <div className="App">
      <h1>useRef</h1>
      <input onChange={changeName} ref={nameRef} type="text" placeholder="Введите имя" />
    </div>
  );
}
```

```

    <button onClick={selectName}>selectName</button>
  <hr />
  <p>render Count: {renderCount.current}</p>
  <hr />
  <h3>name: {name}</h3>
  <h3>prevName: {prevName.current}</h3>
</div>
);
}

```

useRef модальное окно

```

function App() {
  const modalRef = useRef(null);

  const openModal = () => {
    modalRef.current.classList.add("active");
  };

  const closeModal = () => {
    modalRef.current.classList.remove("active");
  };

  return (
    <div className="App">
      <h1>Modal window</h1>
      <button onClick={openModal}>Open modal</button>

      <div ref={modalRef} className="modal-overlay">
        <div className="modal">
          <h2>Modal window</h2>
          <p>Some text...</p>
          <button onClick={closeModal}>Close</button>
        </div>
      </div>
    </div>
  );
}

```

Стили:

```

.modal-overlay {
  display: none;

```

```

  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(95, 158, 160, 0.67);

```

```

  justify-content: center;
  align-items: center;
}

```

```

.modal-overlay.active {
  display: flex;
}

```

```

.modal {
  background-color: #fff;
  padding: 20px;
  width: 400px;
}

```

forwardRef

Возвращение ссылки на ref элемент из компонентов.

Рассмотрим пример формы. Хотим, чтобы после нажатия enter в инпуте, фокус перемещался на следующее поле.

App.js:

```

import { useEffect, useRef } from "react";
import Input from "./Input";

function App() {
  const firstNameRef = useRef(null);
  const secondNameRef = useRef(null);
  const buttonRef = useRef(null);

  useEffect(() => {
    firstNameRef.current.focus();
  }, []);

  const firstKeyDown = (e) => {
    if (e.key === "Enter") secondNameRef.current.focus();
  };

  const secondKeyDown = (e) => {
    if (e.key === "Enter") buttonRef.current.focus();
  };

  return (
    <div className="App">
      <h1>forwardRef</h1>
      <div className="forwardRefForm">
        <Input ref={firstNameRef} onKeyDown={firstKeyDown} placeholder="first name" />
        <Input ref={secondNameRef} onKeyDown={secondKeyDown} placeholder="second name" />
        <button ref={buttonRef}>Submit</button>
      </div>
    </div>
  );
}
export default App;

```

Сокращенный вариант использования:

input.js:

```
import {forwardRef} from 'react'
```

```

const Input = ({onKeyDown, placeholder}, ref)=>{
  return (
    <input
      onKeyDown={onKeyDown}
      placeholder={placeholder}
      type="text"
      ref={ref}
    />);
}

```

```
const forwardedInput = forwardRef(Input)
```

```
export default forwardedInput;
```

Полный вариант forwardRef:

input.js:

```
import React from 'react';
```

```

React.forwardRef(([{onKeyDown, placeholder}, ref]) =>{
  return <input onKeyDown={onKeyDown} placeholder={placeholder} type="text" ref={ref} />;
});

```

В **React.forwardRef** передаем функцию, внутри которой 1-ый аргумент - **пропсы**, а 2-ой - **ref**. Возвращает **разметку**, где мы можем повесить **атрибут ref**

useContext

Рассмотрим пример, когда есть несколько компонентов. Состояние и логика описывается в компоненте App.js - родительском компоненте. Мы можем передать состояние и функции из родительского компонента в компоненты через

пропсы. Если у нас несколько компонентов в компонентах, то нам нужно несколько раз передавать пропсы. Это называется **props drilling**. Это неудобно. Чтобы этого избежать, используется **createContext** и хук **useContext**.

App.js:

```
import { useState, createContext } from 'react'; // импортируем из react

import Header from './components/Header';
import Main from './components/Main';
import Footer from './components/Footer';

export const AppContext = createContext(null); // создаем контекст перед описанием компонента и экспортируем его

function App() {
  const [counter, setCounter] = useState(10);

  const reset = () => setCounter(0);
  const increase = () => setCounter((prev) => prev + 1);
  const decrease = () => setCounter((prev) => prev - 1);

  return (
    <div className='App'>
      <AppContext.Provider value={{ counter, reset, increase, decrease }}> // создали провайдер
        <h1>useContext</h1>
        <Header />
        <Main />
        <Footer />
      </AppContext.Provider>
    </div>
  );
}
export default App;
```

В примере выше был создан контекст `AppContext` и передан как **провайдер**. В `AppContext` **вносим значения**, которые в последствии можно будет использовать в рамках данного контекста. И теперь не нужно передавать их как свойства от компонента к компоненту. Достаточно лишь вызвать хук `useContext` данного контекста и получить необходимые данные:

Main.js - ничего не передаем

```
import Section from './Section';
const Main = () => {
  return (
    <header className='main'>
      <h2>Main</h2>
      <Section />
    </header>
  );
}
export default Main;
```

Section.js:

```
import { useContext } from 'react';
import { AppContext } from './App'; // импортируем контекст из App.js

const Section = () => {
  const { counter, reset, increase, decrease } = useContext(AppContext); // достаем переменные из контекста

  return (
    <section className='header'>
      <h2>Section</h2>
      <h2>Counter: {counter}</h2>
      <button onClick={increase}>Increase</button>
      <button onClick={decrease}>Decrease</button>
      <button onClick={reset}>Reset</button>
    </section>
  );
}
export default Section;
```

useReducer

Рассмотрим пример с двумя счетчиками, которые могут работать одновременно или по отдельности и кнопки для этих действий.

```
function App() {
  const [countLeft, setCountLeft] = useState(0)
  const [countRight, setCountRight] = useState(20)

  const increaseLeft = () => {
    setCountLeft((value) => value + 1)
  }
  const increaseRight = () => {
    setCountRight((value) => value + 1)
  }
  const increaseBoth = () => {
    setCountLeft((value) => value + 1)
    setCountRight((value) => value + 1)
  }

  return (
    <div className='App'>
      <div className="counters">
        <div className="counter">
          <h2>Counter Left</h2>
          <h1>{countLeft}</h1>
        </div>
        <div className="counter">
          <h2>Counter Right</h2>
          <h1>{countRight}</h1>
        </div>
      </div>
      <div className="buttons-wrapper">
        <button onClick={increaseLeft}>Left</button>
        <button onClick={increaseRight}>Right</button>
        <button onClick={increaseBoth}>Both</button>
      </div>
    </div>
  );
}
```

useReducer используется, когда **при одном действии нужно изменять сразу несколько элементов состояния**.

```
const [state, dispatch] = useReducer(reducer, {});
```

1-ый аргумент - функция, которая будет изменять состояние, 2-ой аргумент - объект с начальным состоянием. Возвращает массив из двух значений, первое из которых - состояние, 2-е - функция, которую будем запускать и при этом будет на самом деле запускаться функция **reducer**. Внутри нее часто используют **switch**.

Перепишем код нашего примера:

```
import { useReducer } from "react"; // делаем импорт из react
```

```
function App() {
  const reducer = (state, action) => {
    switch (action.type) {
      case "LEFT":
        return { ...state, counterLeft: state.counterLeft + 1 };
      case "RIGHT":
        return { ...state, counterRight: state.counterRight + 1 };
      case "BOTH":
        return {
          counterLeft: state.counterLeft + 1,
          counterRight: state.counterRight + 1,
        };
      default:
        return state;
    }
  };
}
```



```

const [state, dispatch] = useReducer(reducer, {
  counterLeft: 0,
  counterRight: 20,
});

return (
  <div className="App">
    <div className="counters">
      <div className="counter">
        <h2>Counter Left</h2>
        <h1>{state.counterLeft}</h1>
      </div>
      <div className="counter">
        <h2>Counter Right</h2>
        <h1>{state.counterRight}</h1>
      </div>
    </div>
    <div className="buttons-wrapper">
      <button onClick={() => { dispatch({ type: "LEFT" }); }}> Left </button> //внутри dispatch передаем второй аргумент из reducer
      <button onClick={() => { dispatch({ type: "RIGHT" }); }}> Right </button>
      <button onClick={() => { dispatch({ type: "BOTH" }); }}> Both </button>
    </div>
  </div>
);
}

```

useMemo

Рассмотрим пример с двумя счетчиками. Хотим рассчитать какие-либо значения, основываясь на каком-либо состоянии. Н-р, хотим увеличить значение второго счетчика в два раза. Пусть у нас есть **ресурсоемкая функция** (multiply). useMemo будет запоминать результат вычисления этой функции и отдаст его, не запуская эту функцию. В противном случае без этого хука при изменении первого состояния, функция multiply все равно запускается

useMemo похож на **useEffect**. 1-ый аргумент у него - **функция**, возвращаемое значение которой мы запишем в переменную, 2-ой аргумент - **зависимости**, которые работают точно также, как и в useEffect.

```
import { useState, useMemo } from "react";
```

```

function multiply(num) {
  console.log("multiply");
  for (let i = 0; i <= 1000000000; i++) {} // задержка
  return num * 2;
}

```

```

function App() {
  const [counter1, setCounter1] = useState(0);
  const [counter2, setCounter2] = useState(50);

```

```

  const counter2X2 = useMemo(() => {
    return multiply(counter2);
  }, [counter2]);

```

```
//const counter2X2 = multiply(counter2); // это если без мемо
```

```

const increase1 = () => setCounter1((prev) => prev + 1);
const increase2 = () => setCounter2((prev) => prev + 1);

```

```

return (
  <div className="App">
    <h1>useMemo</h1>
    <hr />
    <h2>Counter 1: {counter1}</h2>
    <button onClick={increase1}>Increase</button>
    <hr />
    <h2>Counter 2: {counter2}</h2>
    <button onClick={increase2}>Increase</button>
    <hr />
  </div>
);

```

```

    <h2>Counter 2x2: {counter2X2}</h2>
  </div>
);
}

```

export default App;

useCallback

Рассмотрим на примере **двух счетчиков**. У **родительского** компонента есть **дочерний**, который завязан на **одном из состояний** родительского компонента (будет выводить **значение второго счетчика + 100**). Функция подсчета (**increaseOn100**) - в родительском компоненте. Передаем ее в дочерний через пропсы. Внутри дочернего компонента есть какие-то расчеты своего состояния с переданным пропсом с **useEffect** в зависимости от функции **increaseOn100**:

App.js:

```

function App() {
  const [counter1, setCounter1] = useState(0);
  const [counter2, setCounter2] = useState(50);

  const increase1 = () => setCounter1((prev) => prev + 1);
  const increase2 = () => setCounter2((prev) => prev + 1);

  const increaseOn100 = () => {
    return counter2 + 100;
  };

  return (
    <div className="App">
      <h1>useCallback</h1>
      <hr />
      <h2>Counter 1: {counter1}</h2>
      <button onClick={increase1}>Increase</button>
      <hr />
      <h2>Counter 2: {counter2}</h2>
      <button onClick={increase2}>Increase</button>
      <hr />
      <ChildComponent increaseOn100={increaseOn100} />
    </div>
  );
}

```

ChildComponent.js:

```

const ChildComponent = ({ increaseOn100 }) => {
  const [value, setvalue] = useState(null);

  useEffect(() => {
    console.log('Child component useEffect Run');
    setvalue(increaseOn100());
  }, [increaseOn100]);

  return <div>Child Component. Counter 2 + 100 = {value}</div>;
};

```

Заметим, что при изменении любого из состояний родительского компонента будет **заново воссоздаваться** и функция **increaseOn100**. Чтобы этого избежать, можно использовать хук **useCallback**. Он похож на **useMemo**, только возвращает не значение, а **функцию**. 1-ый аргумент у него - **функция**, возвращаемое значение которой мы запишем в переменную, 2-ой аргумент - **зависимости**, которые работают точно также, как и в **useEffect**.

```

import { useCallback, useState } from "react";
import ChildComponent from "./ChildComponent";

```

```

function App() {
  const [counter1, setCounter1] = useState(0);
  const [counter2, setCounter2] = useState(50);

  const increase1 = () => setCounter1((prev) => prev + 1);
  const increase2 = () => setCounter2((prev) => prev + 1);

  const increaseOn100 = useCallback(() => {

```

```
    return counter2 + 100;
  }, [counter2]);
```

```
return (
  <div className="App">
    <h1>useCallback</h1>
    <hr />
    <h2>Counter 1: {counter1}</h2>
    <button onClick={increase1}>Increase</button>
    <hr />
    <h2>Counter 2: {counter2}</h2>
    <button onClick={increase2}>Increase</button>
    <hr />
    <ChildComponent increaseOn100={increaseOn100} />
  </div>
);
```