

Цели занятия

- изучить принцип работы useMemo
- изучить принцип работы useCallback
- изучить принцип работы useReducer

Полезные материалы

- [Правила хуков в React](#)
- [Справочник API хуков](#)
- [Оптимизация компонентов React: как не навредить](#)

Как и зачем использовать useMemo?

Хук useMemo используется для мемоизации значений. Как это работает? В функциональном компоненте у нас могут быть функции, которые производят промежуточные вычисления для получения новых знаний в компонентах. Явной иллюстрацией использования useMemo может быть следующий случай.

Имеется компонент с двумя состояниями. Первое состояние — по клику на текст будет менять его цвет, а второе будет увеличивать счетчик на +1 и вызывать функцию sum().

В этой функции производится сложение входящего параметра с самим собой и возвращается сумма этого вычисления. Но представим, что это сложная функция, выполнение которой может требовать больше ресурсов и времени.

Пример:

```
const sum = (n) => {
  return n + n;
};

const Component = () => {
  const [num, setNum] = useState(1);
  const [isGreen, setIsGreen] = useState(true);
  const result = sum(num);

  return (
    <div>
      <h1
        onClick={() => setIsGreen(!isGreen)}
        style={{ color: isGreen ? "green" : "red" }}
      >
        Example
      </h1>
      <h2>Sum {result}</h2>
      <button onClick={() => setNum(num + 1)}>+</button>
    </div>
  );
};
```

В приведенном выше примере есть небольшая проблема. Каждый раз, когда мы кликаем по тексту Example, происходит вызов `setIsGreen`, что приводит к перерендериванию компонента и, как следствие, выполняется повторный вызов функции `sum()`, хотя данный функционал не задействован и должен вызываться только по клику на кнопку +.

Данное вычисление сейчас происходит при каждом рендере компонента, что с точки зрения оптимизации не эффективно.

Для того чтобы оптимизировать (мемоизировать) такие промежуточные вычисления требуется хук `useMemo`. Оборачиваем

функцию `sum()` в `useMemo` и передаем вторым параметром массив зависимостей — теперь только при изменении значения любой зависимости происходит повторный вызов функции `sum()`.

```
const sum = (n) => {
  return n + n;
};

const Component = () => {
  const [num, setNum] = useState(1);
  const [isGreen, setIsGreen] = useState(true);

  const result = useMemo(() => {
    return sum(num);
  }, [num]);

  return (
    <div>
      <h1
        onClick={() => setIsGreen(!isGreen)}
        style={{ color: isGreen ? "green" : "red" }}
      >
        Example
      </h1>
      <h2>Sum {result}</h2>
      <button onClick={() => setNum(num + 1)}>+</button>
    </div>
  );
};
```

Результат выполнения функции `sum()` был мемоизирован, теперь при изменении цвета текста не происходит повторный вызов функции `sum()`, а значит наша оптимизация прошла успешно.

Для чего нужен хук useCallback?

useMemo используется для мемоизации значений, а вот useCallback используется для мемоизации колбеков в компонентах. По своей сути, эти два хука ничем не отличаются и предназначение у них одно и то же — избавление от лишних расчетов.

Хук useCallback похож на хук useMemo с тем отличием, что он возвращает мемоизированный коллбек, который будет обновлен, только если одна из зависимостей будет изменена. Строение идентичное, как и в useEffect/useMemo, первым аргументом идёт коллбек, а вторым — массив зависимостей.

```
const handleClick = useCallback(() => {  
  doSomething(a, b);  
}, [a, b]);
```

В примере выше можно увидеть, как происходит процесс мемоизации функции. Достаточно в первый параметр передать функцию, которую требуется мемоизировать, поместив ее в колбек и передав ей зависимые параметры.

Вторым параметром useCallback является массив зависимых параметров, благодаря которому будет происходить мемоизация данного колбека. Если один из переданных параметров изменится, то useCallback вернет новую мемоизированную функцию.

Но если зависимые параметры не будут изменяться в функциональном компоненте React, то будет использоваться копия данной функции, что позволит избавиться от повторных лишних рендеров компонента.

Хук useCallback на практике

Для демонстрации различий в оптимизации кода с помощью хука useCallback и без него, создадим компонент, в который будем передавать функции и поместим вызов этих функций в useEffect для

контроля изменений при ее вызове. Это будет означать, что если функция не была изменена (мемоизирована), то ее вызов не будет происходить в этом компоненте.

```
import React, { useEffect } from "react";

function Child({ updateOne, updateTwo }) {
  useEffect(() => {
    updateOne();
  }, [updateOne]);

  useEffect(() => {
    updateTwo();
  }, [updateTwo]);

  return <div className="App" />;
}

export default Child;
```

Поместим данный компонент в App.js, в котором будет храниться состояние двух счетчиков, используя хук useState. Также там будут находиться две кнопки, при клике на которые вызываются функции для изменения значения счетчиков.

Самая интересная часть, это две функции, находящиеся в App.js, которые передаются дочернему компоненту Child. Функции будут просто вызывать console.log и их единственное различие будет заключаться в том, что одна из них будет обернута хуком useCallback и иметь в зависимости состояние одного из счетчиков.

```
import React, { useEffect, useState, useCallback } from "react";
import Child from "./child";
import "./styles.css";
```

```

export default function App() {
  const [counter, setCounter] = useState(0);
  const [counterTwo, setCounterTwo] = useState(0);

  const updateOne = () => {
    console.log("Я не мемоизирован");
  };

  const updateTwo = useCallback(() => {
    console.log("Я мемоизирован!");
  }, [counter]);

  return (
    <div className="App">
      <button onClick={() => setCounter(counter + 1)}>One</button>
      <br />
      <button onClick={() => setCounterTwo(counterTwo +
1)}>Two</button>
      <Child updateOne={updateOne} updateTwo={updateTwo} />
    </div>
  );
}

```

Если кликнуть на кнопку “One”, то можно увидеть вызов одновременно двух консолей, при этом, если кликнуть на кнопку “Two”, то будет виден вызов только одной консоли с текстом «Я не мемоизирован». Почему так происходит? Давайте рассмотрим этот процесс подробнее.

По клику на кнопку “One” происходит вызов функции `setCounter`, в которой мы меняем значение `counter` с прибавлением к нему единицы. Функция `updateTwo` мемоизирована и ее новая копия появляется только в момент, когда изменяется значение состояния `counter`. Как следствие мы получаем нашу обновленную функцию и

далее по ссылке в компоненте Child происходит проверка на равенство в `useEffect`. Ссылка на старую функцию изменилась, что приводит к принудительному перерендериванию дочернего компонента с вызовом самой функции.

При нажатии на кнопку “Two” также происходит изменение счетчика `counterTwo` на одну единицу. Но данное состояние не указано в массиве зависимостей хука `useCallback`. Это означает, что функция `updateTwo` не обновится, а React будет использовать ее закешированную версию. При дальнейшем сравнении в компоненте Child ссылка на функцию останется прежней, что не приведет к ее вызову.

Для чего нам `useReducer`?

Данный хук является альтернативой хуку `useState` и применяется в случаях, когда первоначальный стейт является более сложным и много составным или в моменты, когда новое состояние зависит от предыдущего.

```
const [state, dispatch] = useReducer(reducer, initialState, init);
```

Хук `useReducer` принимает три параметра. Первым параметром является функция редьюсер, идея работы которой взята из Redux. Она получает текущее состояние и экшен с типом для его изменения.

```
function reducer(state, action) {
  switch (action.type) {
    case "plus":
      return { count: state.count + 1 };

    case "minus":
      return { count: state.count - 1 };

    default:
      return state;
  }
}
```

Второй параметр, передаваемый `useReducer` является начальное состояние, которое и попадает в первый аргумент нашего редьюсера. Третьим параметром является функция для «ленивой» инициализации первоначального состояния.

Пример:

```
function init(initialState) {  
  return { count: initialState };  
}
```

Данная функция позволяет убрать логику из редьюсера для расчета начального состояния и в основном используется в качестве динамических изменений в начальных состояниях или для быстрого доступа к «сбросу» до этого состояния.

После вызова `useReducer` возвращает массив с состоянием и функцией `dispatch`. Для изменения состояния в редьюсере и получения нового стейта, достаточно передать в функцию `dispatch` нужный экшн. Как видите, данный механизм повторяет логику работы с `Redux`.

useReducer на примерах

Давайте реализуем классический пример счетчика, используя `useReducer` с дополнительной возможностью — пусть каждые 1000 миллисекунд будет происходить увеличение счетчика на 1 при условии, что у нас остаются кнопки для ручного изменения значения счетчика.

```
function reducer(state, action) {  
  switch (action.type) {  
    case "plus":  
      return {
```

```

        ...state,
        count: state.count + 1,
    };

    case "minus":
        return {
            ...state,
            count: state.count - 1,
        };

    default: {
        return state;
    }
}
}

```

Тут ничего интересно. Создаем редьюсер, в который будет попадать экшн. Если тип экшена равен “plus”, то count увеличиваем на 1, если “minus” count уменьшается на 1.

Стоит обратить внимание на использование деструктуризации: запись типа { ...state} говорит о том, что прежде чем поменять объект мы клонируем все его содержимое и уже у этого нового объекта изменяем поле. Это важный момент и концепт иммутабельности (на основе данной концепции построен весь Redux). Получается, что каждый раз редьюсер не меняет первоначальное состояние в старом объекте, а возвращает полностью новое состояние в виде нового объекта.

```

function Counter() {
    const [state, dispatch] = useReducer(reducer, {
        count: 0,
    });
    const { count } = state;

    useEffect(() => {
        const id = setInterval(() => {

```

```

    dispatch({ type: "plus" });
  }, 1000);

  return () => clearInterval(id);
}, []);

return (
  <>
    <h1>count: {count}</h1>
    <button onClick={() => dispatch({ type: "plus"
  })}>INCREMENT</button>
    <button onClick={() => dispatch({ type: "minus"
  })}>DECREMENT</button>
  </>
);
}

```

Counter принимает в хуке `useReducer` функцию `reducer`, которую мы рассматривали выше, и вторым параметром получает начальное состояние счетчика. Компонент имеет две кнопки, по клику на которые происходит вызов функции `dispatch` с передачей ей экшенов и приводящих к изменению стейт компонента.

Для реализации асинхронного эффекта используется хук `useEffect` в котором создается интервал `setInterval` с передачей ему функции `dispatch` и экшена `{ type: «plus» }`. Вторым параметром `setInterval` указывается время, за которое данный интервал будет повторяться.

Для того чтобы очистить память после удаления компонента из DOM, нужно не забыть вызвать `clearInterval` и передать ему ссылку на уже созданный интервал, что позволит почистить его после `unmount` состояния компонента.