React - оптимизация (вспомогательные хуки)

memo
Реф хуки
Хуки производительности
Другие хуки

memo в кратце

memo — это высший компонент (higher-order component), который позволяет оптимизировать производительность функциональных компонентов в React, предотвращая их повторный ререндер, если их пропсы не изменились.

const MemoizedComponent = memo(SomeComponent, arePropsEqual?)

- **SomeComponent**: Компонент, который вы хотите оптимизировать.
- **arePropsEqual** (опционально): Функция, которая принимает предыдущие и следующие пропсы и возвращает true, если пропсы равны (и компонент не должен рендериться заново), или false, если пропсы изменились.

Когда использовать

- Часто обновляемые пропсы: Если компонент получает часто обновляемые пропсы, мемоизация может помочь избежать ненужных рендеров.
- Тяжелый рендеринг: Если компонент выполняет сложные вычисления или рендеринг, мемоизация может улучшить производительность.
- Оптимизация производительности: Используйте React.memo, когда вы видите, что рендеринг компонента вызывает проблемы с производительностью или когда вы уверены, что пропсы изменяются редко.

Реф хуки

Рефы позволяют компоненту хранить некоторую информацию, которая не используется для рендеринга, например, узел DOM или идентификатор таймаута. В отличие от состояния, обновление рефа не приводит к повторному рендерингу компонента. Рефы - это "аварийный люк" из парадигмы React. Они полезны, когда вам нужно работать с системами, не относящимися к React, например, со встроенными API браузера.

- **useRef** объявляет реф. Вы можете хранить в нем любое значение, но чаще всего он используется для хранения узла DOM.
- **useImperativeHandle** позволяет вам настроить реф, открываемый вашим компонентом. Это редко используется.

useRef

useRef - это хук React, позволяющий ссылаться на значение, которое не нужно для рендеринга. Функция useRef возвращает объект с одним свойством current. При следующем рендере useRef вернет тот же объект.

const ref = useRef(initialValue);

- **initialValue**: Значение, которое вы хотите, чтобы свойство current объекта ref было первоначальным. Это может быть значение любого типа. Этот аргумент игнорируется после первоначального рендеринга.
- **current**: Изначально оно установлено на initialValue, которое вы передали. Позже вы можете установить его в другое значение. Если вы передадите объект ref в React как атрибут ref узла JSX, React установит его свойство current.

Вы можете изменить свойство ref.current. В отличие от состояния, оно является изменяемым. Однако, если оно содержит объект, который используется для рендеринга (например, часть вашего состояния), то не стоит мутировать этот объект.

Когда вы изменяете свойство ref.current, React не перерисовывает ваш компонент. React не знает, когда вы изменяете его, потому что ref - это обычный объект JavaScript.

Не записывайте или читайте ref.current во время рендеринга, кроме инициализации. Это делает поведение вашего компонента непредсказуемым.

Использование useRef

Изменение ссылки не вызывает повторного рендеринга.

Используя реф-ссылку, вы гарантируете, что:

- Вы можете сохранять информацию между повторными рендерами (в отличие от обычных переменных, которые сбрасываются при каждом рендере).
- Ее изменение не вызывает повторного рендеринга (в отличие от переменных состояния, которые вызывают повторный рендеринг).
- Информация является локальной для каждой копии вашего компонента (в отличие от внешних переменных, которые являются общими).

Изменение ссылки не вызывает повторного рендеринга, поэтому ссылки не подходят для хранения информации, которую вы хотите вывести на экран. Для этого лучше использовать состояние.

Манипулирование DOM с помощью ссылки

Особенно часто ссылка используется для манипуляций с DOM. React имеет встроенную поддержку для этого.

Сначала объявите реф-ссылку с начальным значением равным null

Затем передайте ваш объект ref в качестве атрибута ref в JSX узла DOM, которым вы хотите манипулировать

После того как React создаст DOM-узел и поместит его на экран, React установит свойство current вашего объекта ref на этот DOM-узел. Теперь вы можете получить доступ к DOM-узлу input и вызвать такие методы, как focus()

React установит свойство current обратно в null, когда узел будет удален с экрана.

Примеры

```jsx
import { useState, useRef } from 'react';

export default function VideoPlayer() {
    const [isPlaying, setIsPlaying] = useState(false);
    const ref = useRef(null);

    function handleClick() {
        const nextIsPlaying = !isPlaying;
        setIsPlaying(nextIsPlaying);

        if (nextIsPlaying) {
            ref.current.play();
        } else {
            ref.current.pause();
        }
    }

    return (
        <>
            <button onClick={handleClick}>
                {isPlaying ? 'Pause' : 'Play'}
            </button>
            <video
                width="250"
                ref={ref}
                onPlay={() => setIsPlaying(true)}
                onPause={() => setIsPlaying(false)}
            >
                <source
                    src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
                    type="video/mp4"
                />
            </video>
        </>
    );
}
```

```jsx
import { useRef } from 'react';

export default function Form() {
    const inputRef = useRef(null);

    function handleClick() {
        inputRef.current.focus();
    }

    return (
        <>
            <input ref={inputRef} />
            <button onClick={handleClick}>
                Focus the input
            </button>
        </>
    );
}
```

forwardRef и собственные компоненты

forwardRef позволяет вашему компоненту передать узел DOM родительскому компоненту с помощью ref.

forwardRef — это мощный инструмент для создания компонентов, которые могут работать с рефами, передаваемыми из родительских компонентов. Это особенно полезно для создания библиотек компонентов, взаимодействующих с элементами DOM или для управления фокусом и другими аспектами UI.

Выставление ссылки на собственный компонент

Иногда вы можете захотеть позволить родительскому компоненту манипулировать DOM внутри вашего компонента. Например, вы пишете компонент MyInput, но хотите, чтобы родительский компонент мог фокусироваться на вводе (к которому родительский компонент не имеет доступа). Вы можете использовать комбинацию useRef для хранения ввода и forwardRef для передачи его родительскому компоненту.

```
import { forwardRef, useRef } from 'react';

const MyInput = forwardRef((props, ref) => {
    return <input {...props} ref={ref} />;
});

export default function Form() {
    const inputRef = useRef(null);

    function handleClick() {
        inputRef.current.focus();
    }

    return (
        <>
            <MyInput ref={inputRef} />
            <button onClick={handleClick}>
                Focus the input
            </button>
        </>
    );
}
```

useImperativeHandle

Хук useImperativeHandle в React позволяет настраивать значения и методы, которые будут доступны родительскому компоненту через реф, когда используется forwardRef. Это дает возможность скрыть внутренние детали компонента и предоставить только необходимые методы или свойства, доступные через реф.

Основная идея: Хук useImperativeHandle используется внутри компонента, который оборачивается forwardRef. Он позволяет настроить объект рефа, который будет доступен родительскому компоненту, чтобы скрыть внутренние детали и предоставить только нужные методы или свойства.

Синтаксис useImperativeHandle

useImperativeHandle(ref, createHandle, dependencies?)

- **ref:** ref, полученный в качестве второго аргумента от функции рендеринга forwardRef.
- **createHandle:** Функция, которая не принимает аргументов и возвращает хэндл ссылки, которую вы хотите раскрыть. Этот ref handle может иметь любой тип. Обычно вы возвращаете объект с методами, которые вы хотите раскрыть.
- **dependencies:** Список всех реактивных значений, на которые ссылается код createHandle. Реактивные значения включают пропсы, состояние, а также все переменные и функции, объявленные непосредственно в теле вашего компонента.

Пример

Вы можете настроить любые методы и свойства, которые хотите сделать доступными родительскому компоненту. Например, если вам нужно предоставить метод для установки значения, вы можете сделать это следующим образом

В этом примере MyInput предоставляет методы setValue и clear, которые можно использовать из родительского компонента.

В этом примере родительский компонент App вызывает методы setValue и clear, предоставленные MyInput, для управления значением поля ввода.

```jsx
const MyInput = forwardRef((props, ref) => {
  const [value, setValue] = useState('');
  const inputRef = useRef();

  useImperativeHandle(ref, () => ({
    setValue: (newValue) => {
      setValue(newValue);
      if (inputRef.current) {
        inputRef.current.value = newValue;
      }
    },
    clear: () => {
      setValue('');
      if (inputRef.current) {
        inputRef.current.value = '';
      }
    },
  }));

  const handleChange = (e) => {
    setValue(e.target.value);
  };

  return (
    <input
      ref={inputRef}
      value={value}
      onChange={handleChange}
      {...props}
    />
  );
});

export default MyInput;
```

```jsx
// Импортируем компонент с useImperativeHandle
import MyInput from './MyInput';

function App() {
  const myInputRef = useRef();

  const handleSetValue = () => {
    if (myInputRef.current) {
      // Установка нового значения
      myInputRef.current.setValue('New Value');
    }
  };

  const handleClear = () => {
    if (myInputRef.current) {
      // Очистка значения
      myInputRef.current.clear();
    }
  };

  return (
    <div>
      <MyInput ref={myInputRef} />
      <button onClick={handleSetValue}>Set Value</button>
      <button onClick={handleClear}>Clear</button>
    </div>
  );
}

export default App;
```

Хуки производительности

Общий способ оптимизации производительности повторного рендеринга - пропустить ненужную работу. Например, вы можете указать React повторно использовать кэшированные вычисления или пропустить повторный рендеринг, если данные не изменились с момента предыдущего рендеринга.

Чтобы пропустить вычисления и ненужный повторный рендеринг, используйте один из этих хуков:

- **useMemo** позволяет кэшировать результат дорогостоящего вычисления.
- **useCallback** позволяет кэшировать определение функции перед передачей ее оптимизированному компоненту.

Иногда нельзя пропустить повторный рендеринг, потому что экран действительно должен обновляться. В этом случае можно повысить производительность, отделив блокирующие обновления, которые должны быть синхронными (например, ввод данных в поле ввода), от неблокирующих обновлений, которые не должны блокировать пользовательский интерфейс (например, обновление графика).

Чтобы установить приоритет рендеринга, используйте один из этих хуков:

- **useTransition** позволяет пометить переход состояния как неблокирующий и разрешить другим обновлениям прерывать его.
- **useDeferredValue** позволяет отложить обновление некритичной части пользовательского интерфейса и позволить другим частям обновляться первыми.

useMemo

Хук **useMemo** в React используется для оптимизации производительности, предотвращая ненужные повторные вычисления значений или вычисляемых данных между рендерами. Он кэширует результат вычислений и возвращает его, если зависимости не изменились. Это может быть полезно для оптимизации работы компонентов, которые зависят от дорогих вычислений или операций, и не хотят их повторять без необходимости.

Когда компонент рендерится, любые вычисления внутри него выполняются заново. Если эти вычисления затратны по времени или ресурсам, это может привести к снижению производительности. Хук useMemo позволяет избежать ненужных повторных вычислений, кэшируя результат и обновляя его только тогда, когда зависимости изменяются.

Синтаксис useMemo

const cachedValue = useMemo(calculateValue, dependencies);

- **calculateValue:** Функция, которая выполняет вычисления и возвращает значение, которое нужно кэшировать.
- **dependencies:** Массив зависимостей. Хук useMemo пересчитывает значение только в том случае, если хотя бы одно из значений в массиве зависимостей изменилось.

useMemo - это хук, поэтому вы можете вызывать его только на верхнем уровне вашего компонента или ваших собственных хуков. Вы не можете вызывать его внутри циклов или условий. Если вам это нужно, создайте новый компонент и переместите состояние в него.

Хук useMemo помогает оптимизировать производительность компонентов, кэшируя результаты дорогих вычислений и возвращая их только при изменении зависимостей. Это особенно полезно, когда у вас есть сложные вычисления или операции, которые не должны выполняться на каждом рендере. Однако, важно использовать его осторожно, так как избыточное использование useMemo может привести к дополнительным затратам на управление кэшем и усложнить код.

useMemo пример

В этом примере, функция computeExpensiveValue выполняет ресурсоемкие вычисления. Благодаря useMemo, вычисления выполняются только тогда, когда number изменяется. При каждом рендере, если number не изменился, memoizedValue будет использоваться из кэша, и ресурсоемкие вычисления не будут повторяться.

```jsx
import React, { useMemo, useState } from 'react';

function ExpensiveComponent({ number }) {
  // Пример затратного вычисления
  const computeExpensiveValue = (num) => {
    console.log('Computing...');
    let result = 0;
    for (let i = 0; i < 1000000000; i++) {
      result += num;
    }
    return result;
  };

  const memoizedValue = useMemo(() => computeExpensiveValue(number), [number]);

  return (
    <div>
      <p>Computed Value: {memoizedValue}</p>
    </div>
  );
}

function App() {
  const [number, setNumber] = useState(1);

  return (
    <div>
      <ExpensiveComponent number={number} />
      <button onClick={() => setNumber(number + 1)}>Increment</button>
    </div>
  );
}

export default App;
```

Пример с фильтром массива

Мы используем useMemo для кэширования отфильтрованного массива filteredItems.

Внутри useMemo происходит фильтрация массива items на основе введенного пользователем значения в поле поиска (searchTerm).

Фильтрация выполняется только тогда, когда изменяется значение searchTerm или исходный массив items.

```jsx
import React, { useState, useMemo } from 'react';

function App() {
  const [searchTerm, setSearchTerm] = useState('');

  // Массив данных для фильтрации
  const items = [
    { id: 1, name: 'Apple' },
    { id: 2, name: 'Banana' },
    { id: 3, name: 'Cherry' },
    { id: 4, name: 'Date' },
    { id: 5, name: 'Grape' },
  ];

  // Используем useMemo для кэширования отфильтрованного списка
  const filteredItems = useMemo(() => {
    console.log('Filtering items...');
    return items.filter((item) =>
      item.name.toLowerCase().includes(searchTerm.toLowerCase())
    );
  }, [searchTerm, items]);

  return (
    <div>
      <input
        type="text"
        placeholder="Search..."
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
      />
      <ul>
        {filteredItems.map((item) => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

useCallback

Хук **useCallback** в React предназначен для оптимизации производительности, предотвращая создание новых экземпляров функций при каждом рендере компонента, если зависимости функции не изменились. Это может быть полезно, если функция передается в дочерние компоненты или используется в зависимостях других хуков, таких как **useEffect** или **useMemo**.

Когда вы создаете функцию внутри компонента, новая функция создается при каждом рендере. Если эта функция передается как пропс в дочерний компонент или используется в зависимостях других хуков, это может привести к ненужным повторным рендерам или пересчетам. Хук useCallback позволяет кэшировать функцию и возвращать ее только тогда, когда зависимости изменяются.

Синтаксис useCallback

const cachedFn = useCallback(fn, dependencies);

- **fn**: Функция, которую нужно кэшировать.
- **dependencies**: Массив зависимостей. Функция будет пересоздаваться только тогда, когда хотя бы одно из значений в массиве зависимостей изменится.

useCallback - это хук, поэтому вы можете вызывать его только на верхнем уровне вашего компонента или ваших собственных хуков. Вы не можете вызывать его внутри циклов или условий. Если вам это нужно, создайте новый компонент и перенесите состояние в него.

Хук useCallback помогает оптимизировать производительность, предотвращая ненужное создание новых экземпляров функций и обеспечивая, что функции передаются только в случае изменений зависимостей. Это полезно для оптимизации компонентов, которые зависят от функций, передаваемых через пропсы или используемых в зависимостях других хуков. Однако, как и с useMemo, важно использовать useCallback разумно, чтобы не усложнять код без необходимости.

Пример

useCallback мемоизирует функцию helloWorldHandler, которая выводит "hello world" в консоль. Мемоизация означает, что функция будет пересоздаваться только в том случае, если изменятся зависимости, указанные в массиве зависимостей (в данном случае массив пустой).

Зачем это нужно? Если бы helloWorldHandler не была обернута в useCallback, новая функция создавалась бы при каждом рендере UseCallBackExample, что приводило бы к тому, что HelloWorldComponent всегда получал бы новый пропс и перерендерился бы, даже если его пропсы логически не изменились.

Почему это важно? Поскольку HelloWorldComponent обернут в memo, он перерендерится только тогда, когда его пропсы изменяются. Использование useCallback гарантирует, что helloWorldHandler не будет пересоздаваться на каждом рендере, что предотвращает ненужные перерендеры HelloWorldComponent.

```jsx
const HelloWorldComponent = memo(({ helloWorldHandler }) => {
  useEffect(() => {
    console.log('rerender');
  });
  return (
    <>
      <button onClick={helloWorldHandler}>hello wolrd</button>
    </>
  );
});

const Counter = ({ counter, counterHandler }) => (
  <>
    <div>{counter}</div>
    <div>
      <button onClick={() => counterHandler((v) => v + 1)}>increase</button>
      <button onClick={() => counterHandler((v) => v - 1)}>decrease</button>
    </div>
  </>
);

function UseCallBackExample() {
  const [counter, setCounter] = useState(0);

  const helloWorldHandler = useCallback(() => {
    console.log('hello world');
  }, []);

  return (
    <>
      <Counter counter={counter} counterHandler={setCounter} />
      <HelloWorldComponent helloWorldHandler={helloWorldHandler} />
    </>
  );
}
```

useTransition

useTransition — это хук, представленный в React 18, который помогает управлять асинхронными обновлениями состояния в пользовательском интерфейсе. Он позволяет разделять приоритетные и менее приоритетные обновления, делая интерфейс более отзывчивым.

Основная цель useTransition — разделить обновления состояния на приоритетные и менее приоритетные, чтобы избежать блокировки пользовательского интерфейса. Например, если у вас есть крупная операция фильтрации или рендеринга, которая может занять некоторое время, вы можете использовать useTransition, чтобы выполнить эту операцию в фоновом режиме, сохраняя отзывчивость интерфейса.

Синтаксис useTransition

const [isPending, startTransition] = useTransition();

- **isPending**: Булевое значение, указывающее, происходит ли в данный момент переходное обновление. Если true, то переходное обновление еще выполняется.
- **startTransition**: Функция, с помощью которой вы оборачиваете менее приоритетные обновления. React будет выполнять эти обновления в фоновом режиме, не блокируя пользовательский интерфейс.

useTransition позволяет вам пометить обновления как менее приоритетные, чтобы React мог выполнять их в фоновом режиме, сохраняя при этом отзывчивость интерфейса.

Этот хук особенно полезен в ситуациях, когда вы имеете дело с крупными операциями или сложными вычислениями, которые могут замедлить рендеринг, если выполнять их сразу.

Используя useTransition, вы можете улучшить пользовательский опыт, позволяя интерфейсу оставаться отзывчивым, даже когда выполняются ресурсоемкие операции.

useTransition пример

Создание состояния:

- Мы создаем три состояния: query для хранения поискового запроса, filteredItems для хранения отфильтрованных элементов и isPending для отслеживания того, происходит ли в данный момент переходное обновление.

Фильтрация элементов:

- Мы создаем массив items из 20,000 элементов.
- В функции handleInputChange обновляется состояние query, которое хранит значение поля ввода.
- Затем с помощью startTransition мы выполняем фильтрацию элементов, оборачивая эту операцию в переходное обновление. Это гарантирует, что интерфейс остается отзывчивым, даже если операция фильтрации занимает значительное время.

Отображение статуса загрузки:

- Пока выполняется переходное обновление (фильтрация), значение isPending будет true, и мы можем отображать индикатор загрузки или текст «Loading...», показывая пользователю, что данные обновляются.

```jsx
import React, { useState, useTransition } from 'react';

function App() {
  const [query, setQuery] = useState('');
  const [filteredItems, setFilteredItems] = useState([]);
  const [isPending, startTransition] = useTransition();

  const items = Array.from({ length: 20000 }, (_, i) => `Item ${i + 1}`);

  const handleInputChange = e => {
    const newQuery = e.target.value;
    setQuery(newQuery);

    // Используем startTransition для выполнения фильтрации в фоновом режиме
    startTransition(() => {
      const filtered = items.filter(item =>
        item.toLowerCase().includes(newQuery.toLowerCase())
      );
      setFilteredItems(filtered);
    });
  };

  return (
    <div>
      <input
        type='text'
        value={query}
        onChange={handleInputChange}
        placeholder='Search items...'
      />
      {isPending && <p>Loading...</p>}
      <ul>
        {filteredItems.map(item => (
          <li key={item}>{item}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

useDeferredValue

useDeferredValue — это хук, также представленный в React 18, который позволяет откладывать обновление состояния до тех пор, пока пользовательский интерфейс не завершит обработку более приоритетных задач.

Основная идея использования useDeferredValue — управлять временем обновления состояния, чтобы избежать блокировки пользовательского интерфейса при работе с ресурсозатратными операциями. Вместо того, чтобы сразу обновлять состояние и рендерить новые данные, React может отложить обновление, пока не будут выполнены все более важные задачи.

Синтаксис useDeferredValue

const deferredValue = useDeferredValue(value);

- **deferredValue**: Отложенное значение, которое обновляется не сразу, а только после завершения рендеринга всех более приоритетных задач.
- **value**: Исходное значение, которое будет отложено.

useDeferredValue помогает откладывать обновления состояния, позволяя интерфейсу оставаться отзывчивым даже при работе с ресурсоемкими операциями.

Этот хук полезен в сценариях, где необходимо поддерживать высокую производительность при взаимодействии с пользователем, особенно когда быстрые изменения состояния могут привести к значительным задержкам в рендеринге.

Использование useDeferredValue улучшает пользовательский опыт, минимизируя задержки при обновлении сложных данных.

useDeferredValue пример

Создание состояния:

- Мы создаем состояние query, чтобы хранить поисковый запрос пользователя.

Отложенное значение:

- Мы используем useDeferredValue, чтобы создать отложенное значение deferredQuery, которое будет обновляться с задержкой после обновления основного состояния query.
- Это означает, что даже если пользователь быстро вводит текст, обновление фильтрованных данных будет происходить с задержкой, что предотвращает замедление интерфейса.

Фильтрация данных:

- Массив items содержит 20,000 элементов.
- Мы фильтруем этот массив на основе отложенного значения deferredQuery, чтобы избежать задержек при быстром вводе пользователем текста.

```jsx
import React, { useState, useDeferredValue } from 'react';

function App() {
  const [query, setQuery] = useState('');

  // Отложенное значение, обновляющееся позже
  const deferredQuery = useDeferredValue(query);

  const items = Array.from({ length: 20000 }, (_, i) => `Item ${i + 1}`);

  const filteredItems = items.filter(item =>
    item.toLowerCase().includes(deferredQuery.toLowerCase())
  );

  return (
    <div>
      <input
        type='text'
        value={query}
        onChange={e => setQuery(e.target.value)}
        placeholder='Search items...'
      />
      <ul>
        {filteredItems.map(item => (
          <li key={item}>{item}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Дополнительные хуки

Эти хуки в основном полезны авторам библиотек и не часто используются в коде приложения.

- **useDebugValue** позволяет вам настроить метку, которую React DevTools отображает для вашего пользовательского хука.
- **useId** позволяет компоненту связать с собой уникальный ID. Обычно используется в API доступности.
- **useSyncExternalStore** позволяет компоненту подписаться на внешний магазин.

useDebugValue

useDebugValue — это хук, который позволяет улучшить отладку пользовательских хуков, добавляя полезную информацию в React DevTools. Этот хук предназначен исключительно для использования внутри пользовательских хуков и не влияет на функциональность компонента или хуков, в которых он используется.

Основная идея использования useDebugValue заключается в улучшении процесса отладки пользовательских хуков. Когда вы создаете собственные хуки, useDebugValue позволяет вам отображать полезную информацию о состоянии или поведении хука непосредственно в React DevTools, что делает отладку проще и удобнее.

Синтаксис useDebugValue

useDebugValue(value, format?)

- **value**: Значение, которое вы хотите отобразить в React DevTools.
- **format**(опционально): Функция форматирования, которая позволяет настраивать отображение значения. Она должна возвращать строку, которая будет отображена в DevTools.

Отладка сложных хуков: Когда вы создаете сложные пользовательские хуки, и хотите добавить дополнительную информацию для облегчения отладки.

Пользовательские хуки, используемые в разных проектах: Если ваш пользовательский хук будет использоваться в нескольких проектах, useDebugValue может помочь другим разработчикам быстрее понять, как работает хук и что он делает.

useDebugValue — это полезный инструмент для улучшения отладки пользовательских хуков, особенно в ситуациях, когда нужно отображать дополнительные данные в React DevTools.

Этот хук делает ваш код более прозрачным и понятным для других разработчиков, а также упрощает процесс отладки в сложных приложениях.

Пример useDebugValue

Пользовательский хук useIsOnline:

- Мы создаем хук useIsOnline, который отслеживает статус подключения к сети (isOnline).
- Внутри useEffect добавляем и удаляем обработчики событий для изменения состояния, когда пользователь подключается или отключается от сети.

Использование useDebugValue:

- Мы вызываем useDebugValue(isOnline ? 'Online' : 'Offline'), чтобы отобразить строку «Online» или «Offline» в React DevTools в зависимости от текущего статуса подключения.
- Это не влияет на функциональность хука, но позволяет разработчикам, использующим React DevTools, сразу видеть, в каком состоянии находится этот хук.

Компонент App:

- Мы используем хук useIsOnline в компоненте App, чтобы отображать сообщение пользователю в зависимости от его статуса подключения.

```jsx
import React, { useState, useEffect, useDebugValue } from 'react';

function useIsOnline() {
  const [isOnline, setIsOnline] = useState(navigator.onLine);

  useEffect(() => {
    const handleOnline = () => setIsOnline(true);
    const handleOffline = () => setIsOnline(false);

    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);

    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);

  // Используем useDebugValue для отображения статуса в DevTools
  useDebugValue(isOnline ? 'Online' : 'Offline');

  return isOnline;
}

function App() {
  const isOnline = useIsOnline();

  return (
    <div>
      <h1>{isOnline ? 'You are online' : 'You are offline'}</h1>
    </div>
  );
}

export default App;
```

useId

useId — это хук, представленный в React 18, который позволяет генерировать уникальные идентификаторы для элементов в вашем компоненте. Эти идентификаторы можно использовать, например, для связывания <label> с соответствующим <input> элементом или для создания уникальных ключей для динамически создаваемых элементов.

Основная цель использования useId — избегать проблем с уникальностью идентификаторов при создании динамически генерируемых элементов, таких как формы или списки, и при этом сохранять консистентность между рендерами. Это особенно важно в приложениях, где возможна клиентская и серверная отрисовка (SSR), так как идентификаторы должны оставаться одинаковыми на обеих сторонах.

Синтаксис useId

const id = useId();

id: Строка, содержащая уникальный идентификатор. Этот идентификатор будет стабильным при каждом рендеринге компонента.

useId — это удобный хук для генерации уникальных идентификаторов, который упрощает создание форм и других элементов, требующих уникальных ID.

Этот хук помогает избежать проблем с дублированием идентификаторов и делает компонентный код более чистым и безопасным, особенно в сценариях, где важна уникальность идентификаторов.

Пример useId

Генерация идентификатора:

- Мы вызываем useId внутри компонента MyForm, чтобы получить уникальный идентификатор, который будет использоваться в пределах этого компонента.
- Значение, возвращаемое useId, является уникальным для каждого экземпляра компонента, но оно будет оставаться постоянным при каждом последующем рендеринге.

Использование идентификатора:

- Сгенерированный идентификатор используется для связывания label с соответствующими полями ввода (input) с помощью атрибута htmlFor и атрибута id.
- Мы комбинируем базовый идентификатор с конкретным именем поля (-name или -email), чтобы создать уникальные значения для каждого элемента формы.

```jsx
import React, { useId } from 'react';

function MyForm() {
  const id = useId();

  return (
    <div>
      <label htmlFor={`${id}-name`}>Name:</label>
      <input id={`${id}-name`} type="text" name="name" />

      <label htmlFor={`${id}-email`}>Email:</label>
      <input id={`${id}-email`} type="email" name="email" />
    </div>
  );
}

export default MyForm;
```

useSyncExternalStore

useSyncExternalStore — это хук, который появился в React 18 и предназначен для интеграции компонентов с внешними источниками данных, такими как глобальное состояние, хранилища (store), подписки и т. д. Этот хук был создан для решения проблем с синхронизацией состояния и обновлениями компонентов, обеспечивая стабильность и консистентность состояния между серверным рендерингом (SSR) и клиентской стороной.

Основная цель useSyncExternalStore — гарантировать, что компоненты React будут синхронизированы с внешними источниками данных, такими как глобальное состояние или подписки на события. Этот хук был разработан для обеспечения консистентности между серверным и клиентским рендерингом и поддерживает работу с любой системой глобального состояния или подписки на данные.

Синтаксис useSyncExternalStore

const state = useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?);

- **subscribe**: Функция, которая используется для подписки на изменения внешнего источника данных. Эта функция должна возвращать функцию, которая отменяет подписку.
- **getSnapshot**: Функция, которая используется для получения текущего значения состояния из внешнего источника на клиенте.
- **getServerSnapshot** (опционально): Функция, которая используется для получения значения состояния при серверной отрисовке (SSR). Если не указана, getSnapshot будет использоваться как на сервере, так и на клиенте.

useSyncExternalStore — это мощный хук для работы с внешними источниками данных, который помогает синхронизировать компоненты с глобальным состоянием или другими внешними данными.

Этот хук особенно полезен при интеграции с глобальными хранилищами данных и подписками на внешние источники, обеспечивая стабильность и консистентность состояния в вашем приложении.

Пример useSyncExternalStore

```jsx
import { useSyncExternalStore } from 'react';

export function useOnlineStatus() {
  const isOnline = useSyncExternalStore(subscribe, getSnapshot);
  return isOnline;
}

function getSnapshot() {
  return navigator.onLine;
}

function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);
  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
}
```

```jsx
import { useOnlineStatus } from "./useOnlineStatus.js";

function StatusBar() {
  const isOnline = useOnlineStatus();
  return <h1>{isOnline ? "✅ Online" : "❌ Disconnected"}</h1>;
}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log("✅ Progress saved");
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? "Save progress" : "Reconnecting..."}
    </button>
  );
}

export default function UseSyncExternalStore() {
  return (
    <>
      <SaveButton />
      <StatusBar />
    </>
  );
}
```

Ресурсы

Код с урока - ТЫК | деплой - ТЫК

memo - документация (ru) - ТЫК | документация (en) - ТЫК

useRef - документация (ru) - ТЫК | документация (en) - ТЫК

useImperativeHandle - документация (ru) - ТЫК | документация (en) - ТЫК

useMemo - документация (ru) - ТЫК | документация (en) - ТЫК

useCallback - документация (ru) - ТЫК | документация (en) - ТЫК

useTransition - документация (ru) - ТЫК | документация (en) - ТЫК

useDeferredValue - документация (ru) - ТЫК | документация (en) - ТЫК

useDebugValue - документация (ru) - ТЫК | документация (en) - ТЫК

useId - документация (ru) - ТЫК | документация (en) - ТЫК

useSyncExternalStore - документация (ru) - ТЫК | документация (en) - ТЫК