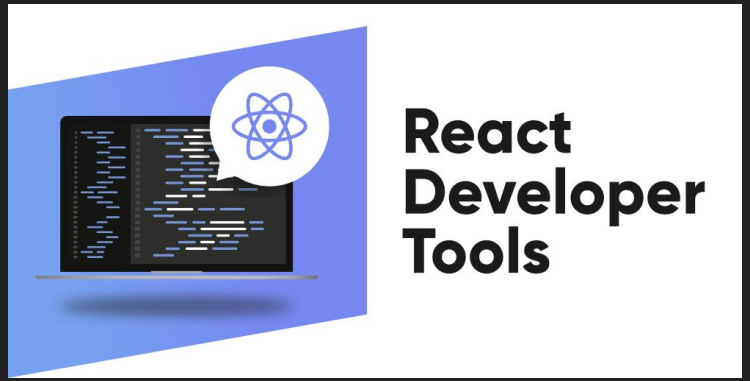
React - базовые хуки

Хуки состояния
Хуки эффектов
Контекстные хуки
React Developer Tools

React Developer Tools (ReactDevTools)

React Developer Tools — это расширение для браузеров, которое предоставляет инструменты для анализа и отладки React-приложений. Оно позволяет вам исследовать структуру компонентов, состояние, пропсы, хуки и многие другие аспекты приложения, что делает процесс разработки и отладки более эффективным.



Как пользоваться React Developer Tools:

Установка:

Установите расширение React Developer Tools для вашего браузера (Google Chrome, Firefox).

После установки и открытия React-приложения в браузере, в инструментах разработчика появятся новые вкладки: Components и Profiler.

Использование:

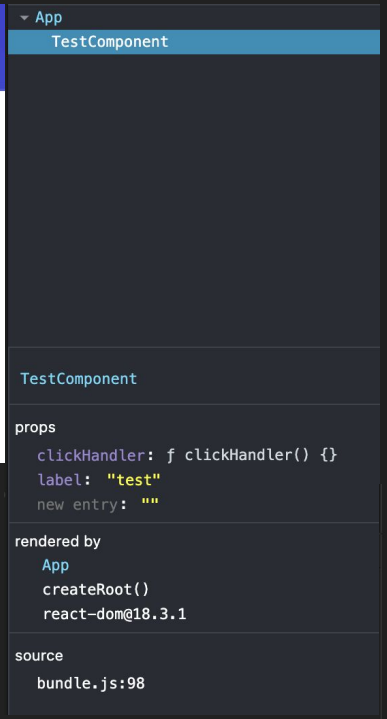
Components: Откройте вкладку Components, чтобы увидеть дерево React-компонентов. Выберите любой компонент, чтобы изучить его пропсы, состояние и хуки. Можно изменять пропсы и состояние для проверки поведения приложения в режиме реального времени.

Profiler: Переключитесь на вкладку Profiler, начните запись, выполните действия в приложении, а затем остановите запись, чтобы проанализировать, какие компоненты обновлялись и сколько времени это заняло.

Практическое применение:

Используйте React DevTools для отладки проблем с состоянием и пропсами, для оптимизации производительности и для лучшего понимания структуры вашего приложения.

Пример react-dev-tools



App.jsx M

src > App.jsx > App

You, 1 second ago | 1 author (You)

```jsx
import { TestComponent } from "./components";

export default function App() {
  return (
    <>
      <h1>hello world</h1>
      <TestComponent
        label={"test"}
        clickHandler={() => {
          console.log("test");
        }}
      />
    </>
  );
}

```

TestComponent.jsx U

src > components > TestComponent.jsx > ...

```jsx
export default function TestComponent({ label, clickHandler }) {
  return (
    <>
      <button onClick={clickHandler}>{label}</button>
    </>
  );
}

```

Что такое хуки в react

Хуки позволяют вам использовать различные функции React из ваших компонентов. Вы можете использовать встроенные хуки или комбинировать их для создания своих собственных.

Хуки сделали функциональные компоненты более мощными и гибкими, упростив код и способствует лучшей реюзабельности логики.

Основные хуки:

- Хуки состояния: useState, useReducer;
- Хуки эффектов: useEffect, useLayoutEffect, useInsertionEffect;
- Контекстные хуки: useContext;
- Реф хуки: useRef, useImperativeHandle;

Хуки состояния

Хуки состояния в React — это хуки, которые позволяют функциональным компонентам использовать внутреннее состояние. До появления хуков состояние могло использоваться только в классовых компонентах с помощью this.state. В функциональных компонентах состояние было невозможно управлять напрямую, но с появлением хуков стало возможным.

Хуки состояния: **useState** и **useReducer**

- **useState**: Подходит для простого состояния или когда действия на состояние просты и предсказуемы (например, увеличение счетчика, изменение строки и т.д.).
- **useReducer**: Подходит для более сложных состояний и логики, особенно когда состояние зависит от предыдущего или существует множество различных действий.

useState

useState — это один из самых простых и часто используемых хуков в React. Он позволяет функциональному компоненту хранить и управлять состоянием.

const [state, setState] = useState(initialState);

- **state:** Переменная, которая хранит текущее значение состояния.
- **setState:** Функция для обновления состояния. При вызове с новым значением компонент будет перерисован с этим новым состоянием.
- **initialState:** Начальное значение состояния. Оно используется только при первом рендере компонента.

Обновление состояния в useState

При использовании хука useState в React есть два способа обновить состояние: передать новое значение напрямую или передать функцию, которая будет вычислять новое значение на основе текущего состояния. Эти два подхода имеют разные применения и поведение.

Передача функции обновления через useState более надежна, когда новое состояние зависит от текущего состояния или когда нужно обновить состояние несколько раз подряд. Передача состояния напрямую проще и подходит для случаев, когда обновление не зависит от предыдущего значения.

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

function Counter() {
  // Инициализация состояния с начальным значением 0
  const [count, setCount] = useState(0);

  // Функции для изменения состояния с использованием функции обратного вызова
  const increment = () => setCount(prevCount => prevCount + 1);
  const decrement = () => setCount(prevCount => prevCount - 1);

  return (
    <div>
      <p>Счетчик: {count}</p>
      <button onClick={increment}>Увеличить</button>
      <button onClick={decrement}>Уменьшить</button>
    </div>
  );
}

export default Counter;
```

Передача следующего состояния напрямую

Когда вы передаете новое значение напрямую в функцию обновления состояния (setState), оно просто заменяет текущее состояние новым значением.

В этом примере:

- При нажатии на кнопку "Увеличить" состояние count обновляется путем передачи нового значения count + 1 напрямую в setCount.
- При нажатии на кнопку "Сбросить" состояние count устанавливается в 0.

Подходит, когда новое состояние не зависит от предыдущего.

Удобно для простых обновлений состояния, например, сброс состояния к определенному значению.

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

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Счетчик: {count}</p>
      <button onClick={() => setCount(count + 1)}>Увеличить</button>
      <button onClick={() => setCount(0)}>Сбросить</button>
    </div>
  );
}
```

Передача функции обновления

Когда вы передаете функцию в setState, эта функция принимает текущее состояние в качестве аргумента и возвращает новое значение. Это особенно полезно, если новое состояние зависит от предыдущего.

В этом примере:

- При нажатии на кнопку "Увеличить" состояние обновляется функцией, которая принимает текущее значение count (prevCount) и возвращает prevCount + 1.
- При нажатии на кнопку "Удвоить" состояние обновляется функцией, которая возвращает удвоенное значение prevCount * 2.

Необходима, когда новое состояние зависит от текущего. Например, когда вы увеличиваете счетчик или выполняете другие операции, связанные с текущим значением.

Особенно полезно при асинхронных обновлениях, когда обновление состояния может происходить одновременно из нескольких источников. В этом случае передача функции гарантирует, что новое состояние всегда будет рассчитано корректно на основе последнего актуального значения.

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

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Счетчик: {count}</p>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>
        Увеличить
      </button>
      <button onClick={() => setCount(prevCount => prevCount * 2)}>
        Удвоить
      </button>
    </div>
  );
}
```

Пример проблемы с асинхронностью

incrementTwice увеличит count только на 1 вместо 2, потому что оба вызова setCount(count + 1) используют одно и то же значение count (0). Это происходит из-за асинхронного характера обновления состояния.

```
function Counter() {
  const [count, setCount] = useState(0);

  function incrementTwice() {
    setCount(count + 1); // count = 0 + 1 = 1
    setCount(count + 1); // count = 0 + 1 = 1 (должно быть 2)
  }

  return (
    <div>
      <p>Счетчик: {count}</p>
      <button onClick={incrementTwice}>Увеличить дважды</button>
    </div>
  );
}
```

Обновление объектов и массивов в состоянии

При работе с объектами и массивами в состоянии useState нужно учитывать, что обновление состояния должно быть иммутабельным (неизменяемым). Это означает, что вместо прямого изменения (мутации) объекта или массива, нужно создать их новую версию с обновлениями. Это важно для правильного обновления компонентов и предотвращения непредсказуемого поведения.

Ключевые моменты

- Иммутабельность: Не изменяйте объекты и массивы напрямую. Вместо этого создавайте новые версии с помощью операторов ... или методов, таких как map, filter и т.д.
- Обновление объектов: Используйте оператор расширения ..., чтобы скопировать старый объект и добавить или изменить нужные свойства.
- Обновление массивов: Используйте методы массива, такие как map, filter, concat, slice и оператор расширения для создания новых массивов.
- Производительность: В React важно избегать мутаций состояния, так как это может привести к неправильному рендерингу компонентов и проблемам с производительностью.

Обновление объектов в состоянии

Когда у вас есть объект в состоянии, и вам нужно обновить одно из его свойств, следует создать новый объект, который будет содержать все свойства старого объекта, плюс обновленное значение.

Здесь:

- Используется оператор расширения (...), чтобы скопировать все свойства из старого объекта prevUser.
- Затем указывается новое значение для свойства name.

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

function Profile() {
  const [user, setUser] = useState({ name: 'Alice', age: 25 });

  const updateName = () => {
    setUser(prevUser => ({
      ...prevUser, // Копируем все свойства старого объекта
      name: 'Bob'  // Обновляем только свойство name
    }));
  };

  return (
    <div>
      <p>Имя: {user.name}</p>
      <p>Возраст: {user.age}</p>
      <button onClick={updateName}>Изменить имя</button>
    </div>
  );
}
```

Обновление массивов в состоянии

Обновление массивов в состоянии также требует создания новой версии массива. Это можно сделать различными способами в зависимости от того, хотите ли вы добавить элемент, удалить его или изменить существующий элемент.

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

function TodoList() {
  const [tasks, setTasks] = useState(['Задача 1', 'Задача 2']);

  const addTask = () => {
    setTasks(prevTasks => [...prevTasks, 'Новая задача']);
  };

  return (
    <div>
      <ul>
        {tasks.map((task, index) => (
          <li key={index}>{task}</li>
        ))}
      </ul>
      <button onClick={addTask}>Добавить задачу</button>
    </div>
  );
}
```

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

function TodoList() {
  const [tasks, setTasks] = useState(['Задача 1', 'Задача 2', 'Задача 3']);

  const removeTask = (indexToRemove) => {
    setTasks(prevTasks => prevTasks.filter((_, index) => index !== indexToRemove));
  };

  return (
    <div>
      <ul>
        {tasks.map((task, index) => (
          <li key={index}>
            {task}
            <button onClick={() => removeTask(index)}>Удалить</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

```jsx
function TodoList() {
  const [tasks, setTasks] = useState([
    { id: 1, text: 'Задача 1', completed: false },
    { id: 2, text: 'Задача 2', completed: false }
  ]);

  const toggleTaskCompletion = (id) => {
    setTasks(prevTasks =>
      prevTasks.map(task =>
        task.id === id ? { ...task, completed: !task.completed } : task
      )
    );
  };

  return (
    <div>
      <ul>
        {tasks.map(task => (
          <li key={task.id}>
            <label>
              <input
                type="checkbox"
                checked={task.completed}
                onChange={() => toggleTaskCompletion(task.id)}
              />
              {task.text}
            </label>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Воссоздания начального состояния useState

Прямое использование начального значения

Если начальное значение состояния известно и не требует сложных вычислений, вы можете просто передать его напрямую в функцию обновления состояния (setState).

Использование функции для начального состояния

Если начальное состояние вычисляется, вы можете передать функцию в useState при инициализации, а затем использовать её для воссоздания начального состояния.

```
const initialState = 0;
const [count, setCount] = useState(initialState);
```

```
const initialState = () => Math.floor(Math.random() * 100);
const [count, setCount] = useState(initialState);
```

Избегание воссоздания начального состояния

React сохраняет начальное состояние один раз и игнорирует его при последующих рендерах.

Хотя результат createInitialTodos() используется только для начального рендеринга, вы все равно вызываете эту функцию при каждом рендеринге. Это может быть расточительно, если она создает большие массивы или выполняет дорогие вычисления.

Чтобы решить эту проблему, вы можете передать ее в качестве функции инициализатора в useState вместо этого

Обратите внимание, что вы передаете createInitialTodos, которая является самой функцией, а не createInitialTodos(), которая является результатом ее вызова. Если вы передадите функцию в useState, React будет вызывать ее только во время инициализации.

```
function TodoList() {
    const [todos, setTodos] = useState(
        createInitialTodos()
    );
    // ...
}
```

```
function TodoList() {
    const [todos, setTodos] = useState(createInitialTodos);
    // ...
}
```

useReducer

useReducer — это хук в React, который позволяет управлять состоянием более сложных компонентов, где необходимо контролировать логику изменения состояния, подобно тому, как это делается с Redux. Это хороший выбор, когда у вас есть несколько состояний, которые зависят друг от друга, или когда изменение состояния требует выполнения сложной логики.

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

- **state:** Текущее состояние, управляемое редьюсером. Его значение будет обновляться всякий раз, когда вы вызовете dispatch с новым действием.
- **dispatch:** Функция, которая используется для отправки действий в редьюсер. Она принимает один аргумент — action, который содержит информацию о том, как должно измениться состояние.
- **reducer** (обязательный аргумент): Это функция, которая определяет, как состояние должно изменяться в ответ на действие.
- **initialState** (обязательный аргумент): Начальное значение состояния, которое будет использовано при первой инициализации хука.
- **init** (необязательный аргумент): Функция инициализации, которая позволяет задать начальное состояние на основе initialState. Это полезно для ленивой инициализации состояния, если вычисление начального состояния является дорогим по ресурсам.

```
// Определяем редьюсер
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { ...state, count: state.count + 1 };
    case 'decrement':
      return { ...state, count: state.count - 1 };
    default:
      throw new Error('Неизвестное действие');
  }
}

// Начальное состояние
const initialState = { count: 0 };

function Counter() {
  // Используем хук useReducer
  const [state, dispatch] = useReducer(reducer, initialState);

  // Функции для отправки действий
  const increment = () => dispatch({ type: 'increment' });
  const decrement = () => dispatch({ type: 'decrement' });

  return (
    <div>
      <p>Счетчик: {state.count}</p>
      <button onClick={increment}>Увеличить на 1</button>
      <button onClick={decrement}>Уменьшить на 1</button>
    </div>
  );
}
```

reducer

Функция reducer является ключевым элементом, когда вы используете хук useReducer в React. Она определяет, как состояние вашего компонента должно изменяться в ответ на действия. Если проводить аналогию, reducer похож на "управляющего", который отвечает за изменение состояния в зависимости от типа действия.

Функция reducer принимает два аргумента:

- state: текущее состояние вашего компонента.
- action: объект, который описывает, какое действие нужно выполнить и какую информацию это действие содержит.

Функция reducer должна вернуть новое состояние на основе текущего состояния и действия.

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}
```

Иммутабельность состояния

Одно из ключевых правил при работе с редьюсерами — состояние должно быть неизменным. Это означает, что вы никогда не должны изменять текущее состояние напрямую. Вместо этого нужно всегда возвращать новый объект состояния.

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      state.count += 1; // Плохо: изменяем текущее состояние напрямую
      return state;
    default:
      return state;
  }
}
```

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { ...state, count: state.count + 1 }; // Хорошо: возвращаем новый объект
    default:
      return state;
  }
}
```

Использование action.payload

Часто в объекте действия (action) передаются дополнительные данные, которые используются для изменения состояния. Эти данные передаются через поле payload.

В этом примере:

- action.type — 'updateName'.
- action.payload содержит значение 'John', которое используется для обновления поля name в состоянии.

```
function reducer(state, action) {
  switch (action.type) {
    case 'updateName':
      return { ...state, name: action.payload };
    default:
      return state;
  }
}
```

```
dispatch({ type: 'updateName', payload: 'John' });
```

функция dispatch

Функция dispatch является важным элементом, когда вы используете хук useReducer в React. Она служит механизмом для передачи действий в редьюсер, который затем обновляет состояние на основе переданного действия.

Функция dispatch в React позволяет централизованно управлять изменением состояния компонента, делая код более управляемым и предсказуемым.

Назначение dispatch:

dispatch используется для отправки (или диспатча) действия в функцию reducer, которая затем на основе типа этого действия определяет, как должно измениться состояние компонента.

Аргумент dispatch:

dispatch принимает один аргумент — объект action. Этот объект должен как минимум содержать поле type, которое описывает тип действия, и может содержать дополнительные данные, необходимые для обновления состояния, обычно называемые payload.

Избегание воссоздания начального состояния

React сохраняет начальное состояние один раз и игнорирует его при последующих рендерах.

Хотя результат createInitialState(username) используется только для начального рендеринга, вы все равно вызываете эту функцию при каждом рендеринге. Это может быть расточительно, если она создает большие массивы или выполняет дорогие вычисления.

Чтобы решить эту проблему, вы можете передать ее в качестве инициализатора функции в useReducer в качестве третьего аргумента вместо этого.

Обратите внимание, что вы передаете createInitialState, которая является самой функцией, а не createInitialState(), которая является результатом ее вызова. Таким образом, начальное состояние не создается заново после инициализации.

```
function createInitialState(username) {
    // ...
}

function TodoList({ username }) {
    const [state, dispatch] = useReducer(
        reducer,
        createInitialState(username)
    );
    // ...
}
```

```
function createInitialState(username) {
    // ...
}

function TodoList({ username }) {
    const [state, dispatch] = useReducer(
        reducer,
        username,
        createInitialState
    );
    // ...
}
```

Хуки эффектов

Эффекты позволяют компоненту подключаться к внешним системам и синхронизироваться с ними Это включает работу с сетью, DOM браузера, анимацией, виджетами, написанными с использованием другой библиотеки UI, и другим не-React кодом.

Эффекты - это "аварийный люк" из парадигмы React. Не используйте эффекты для оркестровки потока данных в вашем приложении. Если вы не взаимодействуете с внешней системой, возможно, вам не нужен эффект.

Есть две редко используемые вариации **useEffect** с различиями во времени:

- **useLayoutEffect** срабатывает до того, как браузер перерисовывает экран. Вы можете измерить компоновку здесь.
- **useInsertionEffect** срабатывает до того, как React внесет изменения в DOM. Здесь библиотеки могут вставлять динамические CSS.

useEffect

Хук useEffect в React позволяет выполнять побочные эффекты в функциональных компонентах. Это мощный инструмент, который используется для различных задач, таких как работа с API, подписка на события, управление таймерами и обновление DOM.

useEffect(callback, dependencies)

- callback: Функция, выполняющая побочный эффект.
- dependencies: Массив зависимостей (опционально).

Функция очистки:

callback может возвращать функцию, которая будет вызвана при размонтировании компонента или перед выполнением следующего эффекта. Это позволяет очистить ресурсы, такие как таймеры или подписки.

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

function Example() {
  useEffect(() => {
    // Побочный эффект: здесь можно выполнять любые операции
    console.log('Компонент был отрендерен или обновлен');

    // Функция очистки (опционально)
    return () => {
      console.log('Компонент будет удален или обновлен');
    };
  }, []); // Зависимости

  return <div>Пример</div>;
}
```

Mounting effect - Пустой массив [] зависимостей

Эффект запускается только один раз, при первом рендере компонента.
Эффект ведет себя как componentDidMount в классовых компонентах.

```
useEffect(() => {
  console.log('Этот эффект запускается один раз при первом рендере');
}, []);
```

Render effect - Без массива зависимостей

Эффект запускается при каждом рендере компонента. Это может вызвать проблемы с производительностью и непредсказуемое поведение.

```
useEffect(() => {
  console.log('Этот эффект запускается при каждом рендере');
});
```

Dependency effect - Массив с зависимостями

Эффект запускается при изменении значений в массиве зависимостей.

```
useEffect(() => {
  console.log('Этот эффект запускается при изменении значения count');
}, [count]);
```

Unmount effect - функция возвращающая функцию

Функция очистки, возвращаемая из useEffect, выполняется в двух случаях:

Перед выполнением следующего эффекта:

Если ваш компонент обновляется (т.е. происходит рендеринг с новыми пропсами или состоянием), React сначала вызывает функцию очистки предыдущего эффекта, а затем выполняет новый эффект. Это обеспечивает корректное управление ресурсами и предотвращает конфликты между предыдущими и текущими эффектами.

При размонтировании компонента:

Когда компонент удаляется из DOM, функция очистки вызывается для выполнения необходимых действий, таких как остановка таймеров, отписка от событий или отмена запросов.

Функция очистки, возвращаемая из useEffect, позволяет React правильно управлять ресурсами, освобождая их перед установкой новых эффектов и при удалении компонента. Это обеспечивает корректную работу приложения и предотвращает утечки памяти и другие проблемы.

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

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    // Устанавливаем таймер
    const timer = setInterval(() => {
      console.log('Таймер сработал');
      setCount(prevCount => prevCount + 1);
    }, 1000);

    // Функция очистки: вызывается при размонтировании компонента
    // или перед выполнением следующего эффекта
    return () => {
      clearInterval(timer);
      console.log('Таймер очищен');
    };
  }, []); // Пустой массив зависимостей, таймер запускается один раз при монтировании

  return <div>Счетчик: {count}</div>;
}

export default Timer;
```

Получение асинхронных данных

Состояния:

- posts: Хранит данные постов, полученные с сервера.
- loading: Информирует о процессе загрузки данных.
- error: Хранит сообщения об ошибках, если запрос не удался.

useEffect:

- Запускает асинхронную функцию fetchPosts при монтировании компонента (пустой массив зависимостей [] означает, что эффект выполнится только один раз).
- Функция fetchPosts делает запрос к API jsonplaceholder, проверяет успешность ответа, преобразует данные в формат JSON и обновляет состояние.
- В случае ошибки, ошибка сохраняется в состоянии error.

Отображение данных:

- Пока данные загружаются, отображается сообщение "Загрузка...".
- Если произошла ошибка, отображается сообщение об ошибке.
- После успешного получения данных отображается список постов.

Функция очистки:

- В этом примере функция очистки не требуется, но она включена в useEffect для потенциальной очистки ресурсов, если это необходимо (например, отмена запросов или очистка подписок).

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

function Posts() {
  const [posts, setPosts] = useState([]); // Состояние для хранения постов
  const [loading, setLoading] = useState(true); // Состояние для индикатора загрузки
  const [error, setError] = useState(null); // Состояние для хранения ошибки

  useEffect(() => {
    // Определяем асинхронную функцию для получения данных
    async function fetchPosts() {
      try {
        // Выполняем запрос
        const response = await fetch('https://jsonplaceholder.typicode.com/posts');

        // Проверяем, был ли запрос успешным
        if (!response.ok) {
          throw new Error('Сеть ответила с ошибкой');
        }

        // Получаем данные в формате JSON
        const data = await response.json();

        // Обновляем состояние с полученными данными
        setPosts(data);
      } catch (err) {
        // Обрабатываем ошибку
        setError(err.message);
      } finally {
        // Устанавливаем состояние загрузки в false после завершения запроса
        setLoading(false);
      }
    }

    // Вызываем асинхронную функцию
    fetchPosts();

    // Функция очистки (если необходима)
    return () => {
      // Здесь можно выполнять очистку (если потребуется)
    };
  }, []); // Пустой массив зависимостей - запрос выполнится только один раз при монтировании

  // Отображаем индикатор загрузки
  if (loading) return <p>Загрузка...</p>;

  // Отображаем ошибку, если она произошла
  if (error) return <p>Ошибка: {error}</p>;

  // Отображаем посты
  return (
    <div>
      <h1>Список постов</h1>
      <ul>
        {posts.map(post => (
          <li key={post.id}>
            <h2>{post.title}</h2>
            <p>{post.body}</p>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default Posts;
```

Почему не использовать асинхронную функцию

useEffect принимает функцию обратного вызова, которая должна быть синхронной. Нельзя передать асинхронную функцию напрямую в useEffect, так как это приведет к проблемам с выполнением и обработкой промисов.

useEffect может возвращать функцию очистки, которая вызывается перед выполнением следующего эффекта или при размонтировании компонента. Функция очистки должна быть синхронной. Асинхронные функции возвращают промисы, которые не совместимы с ожидаемым возвращаемым значением useEffect.

```
// Неправильно
useEffect(async () => {
  // Код
}, []);
```

useLayoutEffect

useLayoutEffect — это хук в React, который позволяет выполнять побочные эффекты, синхронно изменяющие DOM после изменения состояния и перед следующими отрисовками экрана. Он похож на useEffect, но с некоторыми ключевыми отличиями в том, когда и как он выполняется.

Основные отличия от useEffect

- **Синхронное выполнение:** useLayoutEffect вызывается синхронно после всех изменений DOM, но до того, как браузер выполнит отрисовку изменений на экране. Это позволяет выполнять операции, которые должны быть выполнены до отрисовки, такие как измерение размеров элементов или их позиционирование.
- **Блокировка отрисовки:** Поскольку useLayoutEffect выполняется синхронно, это может блокировать отрисовку, пока не завершится выполнение эффекта. Это может привести к задержкам в рендеринге, если эффект выполняется долго.

useLayoutEffect(callback, dependencies);

- **callback:** Функция, которая выполняется после изменения DOM.
- **dependencies:** Массив зависимостей, определяющий, когда эффект должен выполняться. Если массив пуст, эффект выполняется только при монтировании и размонтировании компонента.

Когда использовать useLayoutEffect

- Измерение размеров элементов: Если нужно измерить размеры элементов и отреагировать на это до следующей отрисовки.
- Манипуляции с DOM: Если необходимо выполнить манипуляции с DOM, которые могут влиять на следующую отрисовку.
- Программное позиционирование: Если требуется программно изменить позицию или размеры элементов, основываясь на их текущих значениях.

Пример useLayoutEffect

Состояние и реф:

- size: Хранит размеры элемента.
- divRef: Используется для доступа к элементу в DOM.

useLayoutEffect:

- Внутри useLayoutEffect мы измеряем размеры элемента, используя ссылку divRef, и обновляем состояние size.
- Это происходит до следующей отрисовки, что обеспечивает актуальные данные о размерах элемента.

Функция очистки:

- В этом примере функция очистки не требуется, но она может быть полезна, если требуется выполнить очистку (например, отмену событий или таймеров).

```
import React, { useRef, useState, useLayoutEffect } from 'react';

function MeasureComponent() {
  const [size, setSize] = useState({ width: 0, height: 0 });
  const divRef = useRef(null);

  useLayoutEffect(() => {
    // Функция для измерения размера элемента
    function measure() {
      if (divRef.current) {
        const { offsetWidth, offsetHeight } = divRef.current;
        setSize({ width: offsetWidth, height: offsetHeight });
      }
    }

    // Выполняем измерения
    measure();

    // Можно возвращать функцию очистки, если это необходимо
    return () => {
      // Очистка (если требуется)
    };
  }, []); // Пустой массив зависимостей – эффект выполнится один раз при монтировании

  return (
    <div>
      <div ref={divRef} style={{ width: '100%', height: '100px', backgroundColor: 'lightblue' }}>
        Измеряем этот элемент
      </div>
      <p>Ширина: {size.width}px</p>
      <p>Высота: {size.height}px</p>
    </div>
  );
}

export default MeasureComponent;
```

useInsertionEffect

useInsertionEffect — это хук, введённый в React 18, предназначенный для выполнения побочных эффектов, связанных с вставкой стилей в DOM. Этот хук используется для выполнения эффектов, которые должны быть выполнены до того, как React выполнит финальные обновления DOM, что особенно важно для управления стилями и предотвращения визуальных сдвигов или мерцаний.

Основные особенности useInsertionEffect

- **Синхронное выполнение до отрисовки:** useInsertionEffect выполняется синхронно после изменения DOM и перед тем, как React завершит обновление экрана. Это позволяет выполнять операции, которые должны произойти до финальной отрисовки, таких как динамическое добавление стилей.
- **Используется для управления стилями:** Этот хук особенно полезен для управления динамическими стилями, например, при использовании CSS-in-JS библиотек или при работе с библиотеками, которые манипулируют стилями на лету.

useInsertionEffect(callback, dependencies);

- **callback**: Функция, которая выполняется после изменения DOM и перед отрисовкой.
- **dependencies**: Массив зависимостей, определяющий, когда эффект должен выполняться. Если массив пуст, эффект выполняется только при монтировании и размонтировании компонента.

Когда использовать useInsertionEffect

- **Динамическое управление стилями:** При необходимости динамически добавлять или изменять стили до финальной отрисовки компонента.
- **Избегание визуальных сдвигов:** При использовании библиотек стилей, которые требуют вставки стилей до окончательной отрисовки для предотвращения мерцаний или сдвигов.

Пример useInsertionEffect

Состояние:

- color: Состояние, которое управляет цветом текста.

useInsertionEffect:

- Создаем новый элемент <style>, который содержит динамический стиль, основанный на текущем состоянии color.
- Добавляем этот элемент в <head> документа, чтобы применить динамические стили.
- В функции очистки удаляем элемент <style>, чтобы избежать утечек памяти и конфликтов стилей.

Эффект и очистка:

- Эффект выполняется при изменении состояния color, что позволяет динамически обновлять стили.
- Функция очистки удаляет элемент <style> из документа при размонтировании компонента или перед выполнением следующего эффекта.

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

function DynamicStyleComponent() {
  const [color, setColor] = useState('blue');

  useInsertionEffect(() => {
    // Создаем стиль и добавляем его в DOM
    const style = document.createElement('style');
    style.textContent = `
      .dynamic {
        color: ${color};
      }
    `;
    document.head.appendChild(style);

    // Функция очистки
    return () => {
      document.head.removeChild(style);
    };
  }, [color]); // Эффект будет выполняться при изменении состояния color

  return (
    <div>
      <p className="dynamic">Этот текст имеет динамический цвет!</p>
      <button onClick={() => setColor(color === 'blue' ? 'red' : 'blue')}>
        Изменить цвет
      </button>
    </div>
  );
}

export default DynamicStyleComponent;
```

Контекстный хук useContext

useContext — это хук в React, который позволяет компонентам получать данные из контекста, созданного с помощью **React.createContext**. Контексты в React позволяют передавать данные через дерево компонентов без необходимости явно передавать их через пропсы на каждом уровне.

Основные концепции

- **Контекст:** Контекст предоставляет способ делиться значениями между компонентами без необходимости передавать пропсы через все уровни дерева компонентов. Контексты полезны для управления глобальным состоянием или данными, такими как темы, аутентификация или текущий язык.
- **Создание контекста:** Используйте React.createContext для создания контекста. Это возвращает объект контекста, который содержит два компонента: Provider и Consumer.

const value = useContext(MyContext);

- **MyContext**: Это объект контекста, созданный с помощью React.createContext.
- **value**: Значение контекста, которое предоставляет ближайший Provider в дереве компонентов.

Когда использовать useContext

- **Глобальное состояние:** Если у вас есть данные, которые должны быть доступны в нескольких компонентах на разных уровнях дерева, контекст помогает избежать передачи пропсов через каждый уровень.
- **Темы и настройки:** Для предоставления глобальных настроек, таких как тема приложения, язык или пользовательские настройки.
- **Аутентификация**: Для хранения и доступа к информации о текущем пользователе и его статусе аутентификации.

Создание контекста с помощью createContext

React.createContext — это функция, используемая для создания нового объекта контекста в React. Контекст позволяет передавать данные через дерево компонентов без необходимости явно передавать их через пропсы на каждом уровне.

Компоненты контекста:

- **Provider:** Компонент, который предоставляет значение контекста для всех дочерних компонентов.
- **Consumer:** Компонент, который позволяет получать значение контекста в дочерних компонентах.

const MyContext = React.createContext(defaultValue);

- **defaultValue:** Значение, которое используется в случае, если компонент не обернут в Provider. Обычно это начальное значение, которое может быть полезно для тестирования или при использовании контекста без Provider.

```
import React from 'react';

// Создаем контекст с начальным значением
const ThemeContext = React.createContext('light');

export default ThemeContext;
```

```
import React from 'react';
import ThemeContext from './ThemeContext';

function ThemeProvider({ children }) {
  const [theme, setTheme] = React.useState('light');

  // Передаем значение контекста через Provider
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

export default ThemeProvider;
```

Использование компонента Consumer

Consumer — это компонент, предоставляемый объектом контекста, созданным с помощью React.createContext. Он используется для получения значения контекста в компонентах, которые находятся ниже в дереве компонентов от Provider.

Основные моменты о Consumer

- Получение значения контекста: Consumer позволяет компонентам, находящимся внутри дерева контекста, получать значение контекста, которое было установлено ближайшим Provider.
- Синтаксис использования: Компонент Consumer принимает функцию в качестве дочернего элемента. Эта функция получает текущее значение контекста в качестве аргумента и возвращает элемент, который будет отрисован.

```
import React from 'react';
import ThemeContext from './ThemeContext';

function ThemedButton() {
  return (
    <ThemeContext.Consumer>
      {({ theme, setTheme }) => (
        <button
          style={{
            background: theme === 'light' ? '#fff' : '#333',
            color: theme === 'light' ? '#000' : '#fff',
          }}
          onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}
        >
          Switch Theme
        </button>
      )}
    </ThemeContext.Consumer>
  );
}

export default ThemedButton;
```

Использование хука useContext

Хук useContext в React позволяет компонентам получать значение из контекста, созданного с помощью React.createContext. Это удобный способ для доступа к контексту внутри функциональных компонентов, без необходимости использования компонента Consumer.

```jsx
import React, { useContext } from 'react';
import ThemeContext from './ThemeContext';

function ThemedButton() {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <button
      style={{
        background: theme === 'light' ? '#fff' : '#333',
        color: theme === 'light' ? '#000' : '#fff',
      }}
      onClick={toggleTheme}
    >
      Switch Theme
    </button>
  );
}

export default ThemedButton;
```

Собираем все вместе

```
import React from 'react';
import ReactDOM from 'react-dom';
import ThemeProvider from './ThemeProvider';
import ThemedButton from './ThemedButton';

function App() {
  return (
    <ThemeProvider>
      <ThemedButton />
    </ThemeProvider>
  );
}

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

Компонент ThemeProvider реализован на основе createContext и компонента Provider

Компонент ThemeButton реализован либо через компонент Consumer либо через использование хука useContext с передачей контекста созданного createContext

Масштабирование с помощью контекста и редуктора

Масштабирование с помощью контекста и редуктора в React позволяет эффективно управлять глобальным состоянием приложения. Использование контекста (React.createContext) в сочетании с редуктором (useReducer) помогает управлять состоянием и действиями в больших приложениях, улучшая организацию и читаемость кода.

Преимущества комбинации контекста и редуктора

- **Глобальное состояние:** Контекст предоставляет глобальное состояние, доступное для всех компонентов, а редуктор позволяет централизованно управлять этим состоянием и обновлять его на основе действий.
- **Управление сложными состояниями:** Редуктор помогает справляться со сложным состоянием и логикой обновления, обеспечивая предсказуемость и улучшая управление состоянием.
- **Разделение обязанностей:** Контекст отвечает за предоставление состояния, а редуктор управляет его изменением, что упрощает отладку и расширение кода.

Шаги для реализации

- **Создание контекста и редуктора:** Определите контекст и редуктор для управления состоянием.
- **Создание провайдера:** Создайте провайдер, который будет использовать редуктор и передавать значение контекста.
- **Использование контекста в компонентах:** Используйте контекст и редуктор в компонентах для доступа к состоянию и диспатчинга действий.

Пример useContext + useReducer

```jsx
// src/CounterContext.js
import React, { createContext, useReducer } from 'react';

// Создаем контекст
const CounterContext = createContext();

// Определение начального состояния
const initialState = {
  count: 0,
};

// Определение редуктора
function counterReducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

// Провайдер контекста
function CounterProvider({ children }) {
  const [state, dispatch] = useReducer(counterReducer, initialState);

  return (
    <CounterContext.Provider value={{ state, dispatch }}>
      {children}
    </CounterContext.Provider>
  );
}

export { CounterProvider, CounterContext };
```

```jsx
// src/Counter.js
import React, { useContext } from 'react';
import { CounterContext } from './CounterContext';

function Counter() {
  const { state, dispatch } = useContext(CounterContext);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
    </div>
  );
}

export default Counter;
```

```jsx
// src/App.js
import React from 'react';
import ReactDOM from 'react-dom';
import CounterProvider from './CounterContext';
import Counter from './Counter';

function App() {
  return (
    <CounterProvider>
      <Counter />
    </CounterProvider>
  );
}

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

Создание контекста и редуктора:

Мы создаем контекст CounterContext и редуктор counterReducer, который управляет состоянием счетчика. Редуктор обрабатывает действия INCREMENT и DECREMENT.

Создание провайдера:

CounterProvider использует хук useReducer для управления состоянием счетчика и передает состояние и функцию dispatch через контекст.

Использование контекста в компоненте:

В компоненте Counter мы используем хук useContext для получения значения контекста. Мы отображаем текущее значение счетчика и предоставляем кнопки для увеличения и уменьшения значения.

Оборачивание компонентов:

В компоненте App мы оборачиваем Counter в CounterProvider, чтобы Counter мог получить доступ к контексту и управлять состоянием счетчика.

Ресурсы

Код с урока: исходник - ТЫК | деплой - ТЫК

React Developer Tools - ТЫК

useState: официальная документация (en) - ТЫК | переведенная документация (ru) - ТЫК

useReducer: официальная документация (en) - ТЫК | переведенная документация (ru) - ТЫК

useEffect: официальная документация (en) - ТЫК | переведенная документация (ru) - ТЫК

useContext: официальная документация (en) - ТЫК | переведенная документация (ru) - ТЫК