React база

реактивность
useState
условный рендеринг
рендеринг списка
компоненты (props/callbacks)
жизненный цикл компонента
useEffect

Реактивность

Реактивность в React — это концепция, которая позволяет компонентам автоматически обновляться в ответ на изменения данных. В React она реализована через систему управления состоянием и свойствами компонентов.

Когда состояние (state) или свойства (props) компонента изменяются, React автоматически определяет, какие части пользовательского интерфейса (UI) нужно перерендерить, чтобы отобразить новые данные. Это достигается за счет использования виртуального DOM, который сравнивает предыдущую и текущую версии DOM и обновляет только те части, которые действительно изменились.

хук useState

useState — это один из самых часто используемых хуков в React, который позволяет функциональным компонентам управлять состоянием. До появления хуков, управление состоянием было возможно только в классовых компонентах, но с введением хуков, функциональные компоненты тоже получили эту возможность.

Основы работы с useState

useState позволяет вам добавить состояние в функциональный компонент. Хук возвращает массив из двух элементов:

- Текущее значение состояния (или начальное значение).
- Функция для обновления состояния.

Синтаксис useState

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

state: Переменная, содержащая текущее значение состояния.

setState: Функция, которую вы вызываете для обновления значения состояния.

initialState: Начальное значение состояния, которое устанавливается при первом рендере компонента.

Асинхронность обновлений:

Обновления состояния в React асинхронны. Это значит, что React может сгруппировать несколько обновлений в одно, чтобы оптимизировать производительность. Поэтому нельзя полагаться на текущее значение состояния сразу после вызова setState.

Например, если вам нужно обновить состояние на основе предыдущего значения, лучше использовать функциональный синтаксис:

setCount(prevCount => prevCount + 1);

В одном компоненте можно использовать несколько вызовов useState для управления разными кусочками состояния:

const [count, setCount] = useState(0);
const [name, setName] = useState('React');

Пример использования useState (задача счетчик)

```jsx
export default function () {
  let counterWithoutRender = 0;

  function incrementCountWithoutRender() {
    counterWithoutRender = counterWithoutRender + 1;
    console.log(counterWithoutRender);
  }

  function decrementCountWithoutRender() {
    counterWithoutRender = counterWithoutRender - 1;
    console.log(counterWithoutRender);
  }

  return (
    <>
      <h3>Пример без useState</h3>
      <div style={{ textAlign: 'center' }}>{counterWithoutRender}</div>
      <div style={{ display: 'flex', justifyContent: 'space-between' }}>
        <button onClick={incrementCountWithoutRender}>increment</button>
        <button onClick={decrementCountWithoutRender}>decrement</button>
      </div>
    </>
  );
}
```

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

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

  function incrementCountWithRender() {
    setCount((prevValue) => prevValue + 1);
  }

  function decrementCountWithRender() {
    setCount((prevValue) => prevValue - 1);
  }

  return (
    <>
      <h3>Пример с useState</h3>
      <div style={{ textAlign: 'center' }}>{count}</div>
      <div style={{ display: 'flex', justifyContent: 'space-between' }}>
        <button onClick={incrementCountWithRender}>increment</button>
        <button onClick={decrementCountWithRender}>decrement</button>
      </div>
    </>
  );
}
```

Условный рендеринг

Условный рендеринг в React позволяет отображать различные компоненты или элементы на основе определенных условий. Это аналогично тому, как работают условные операторы (if, else, switch) в JavaScript.

Основные способы условного рендеринга

- Оператор if-else
- Использование тернарного оператора
- Логическое И (&&)
- Оператор switch

Оператор if-else

```
export default function ({ myCondition }) {
  if (myCondition) {
    return <div>if else condition is true</div>;
  } else {
    return <div>if else condition is false</div>;
  }
}
```

Использование тернарного оператора

```
export default function ({ myCondition }) {
  return <div>{myCondition ? 'Условия приняты' : 'Условия нарушены'}</div>;
}
```

Логическое И (&&)

```
export default function ({ myCondition }) {
  return (
    <>
      {myCondition && <div>все окей</div>}
      {!myCondition && <div>все не окей</div>}
    </>
  );
}
```

Оператор switch

```
export default function ({ myCondition }) {
  switch (myCondition) {
    case true:
      return <div>Условия со значением true</div>;
    case false:
      return <div>Условия со значением false</div>;
  }
}
```

Рендеринг списка

Рендеринг списков в React является одной из основных задач при разработке приложений, так как часто необходимо отображать наборы данных, такие как массивы объектов, массивы строк и т.д. React предлагает простой и эффективный способ рендеринга списков с использованием метода map для итерации по массиву и генерации JSX-элементов для каждого элемента.

```
function ItemList({ items }) {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

Ключи (keys) в списках

Ключи (keys) — это важный аспект рендеринга списков в React. Они помогают React отслеживать элементы списка и управлять обновлениями, удалением или добавлением элементов. Ключи должны быть уникальными среди братьев и сестер (siblings) в одном списке, но не обязаны быть уникальными глобально.

Здесь в качестве ключа используется уникальный идентификатор item.id, что предпочтительнее, чем индекс массива. Использование индексов массива в качестве ключей может приводить к проблемам при изменении порядка элементов, добавлении или удалении элементов из массива.

```
function ItemList({ items }) {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

Рендеринг списков компонентов

В списках можно рендерить не только строки или простые элементы, но и более сложные компоненты:

Здесь каждый элемент списка todos передается в компонент TodoItem, который отвечает за рендеринг отдельных задач.

```
function TodoItem({ todo }) {
  return (
    <div>
      <h3>{todo.title}</h3>
      <p>{todo.description}</p>
    </div>
  );
}

function TodoList({ todos }) {
  return (
    <div>
      {todos.map((todo) => (
        <TodoItem key={todo.id} todo={todo} />
      ))}
    </div>
  );
}
```

Условный рендеринг списков

Можно также использовать условный рендеринг для отображения списков, например, если список пуст:

Если массив items пуст, вместо списка отображается сообщение "Список пуст".

```
function ItemList({ items }) {
  if (items.length === 0) {
    return <p>Список пуст.</p>;
  }

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

Компонент

Компоненты являются основой React, и они позволяют разбивать интерфейс на независимые, повторно используемые части. Каждый компонент управляет своим собственным состоянием и может быть комбинирован с другими компонентами для создания сложных пользовательских интерфейсов.

В React есть два основных типа компонентов:

- Функциональные компоненты (Function Components)
- Классовые компоненты (Class Components)

props & callbacks

Компоненты в React можно воспринимать как функции в JS

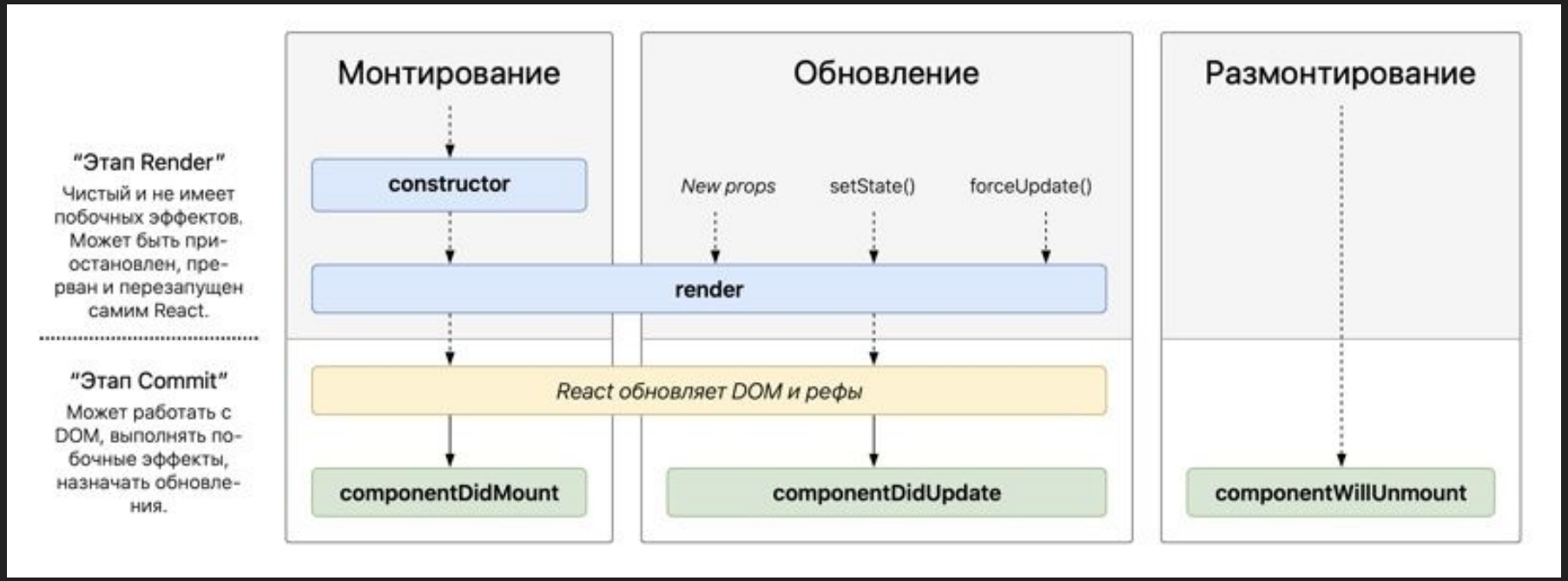
Функции в JS могут работать с callback-ами и аргументами/параметрами

Props - аргумент/параметр - данные передаваемые компоненту

callback - callback - внешние функции передаваемые компоненту

Пропсы (props) и колбеки (callbacks) — это важные концепции в React, которые позволяют компонентам взаимодействовать друг с другом и управлять поведением интерфейса.

Жизненный цикл компонента (классовый компонент)



хук useEffect

Функциональные компоненты в React не имеют встроенных методов жизненного цикла, но с введением хуков появилась возможность управлять жизненным циклом компонентов с помощью useEffect.

Хук useEffect — это один из самых мощных и часто используемых хуков в React. Он позволяет выполнять побочные эффекты в функциональных компонентах. Побочные эффекты могут включать в себя взаимодействие с внешними API, изменение DOM, подписку на события, таймеры и многое другое.

Основные концепции useEffect

useEffect заменяет методы жизненного цикла, такие как componentDidMount, componentDidUpdate, и componentWillUnmount, в классовых компонентах. В функциональных компонентах с useEffect вы можете управлять поведением компонента на различных этапах его жизни.

Синтаксис useEffect

Первый аргумент — это функция-эффект, которая содержит код, который вы хотите выполнить после рендеринга компонента.

Второй аргумент (необязательный) — это массив зависимостей, который определяет, когда эффект должен выполняться. Если массив зависимостей пустой ([]), эффект будет выполняться только один раз, аналогично componentDidMount. Если массив не указан, эффект будет выполняться после каждого рендера, как componentDidUpdate.

```
useEffect(() => {
  // Ваш код побочного эффекта

  return () => {
    // Очистка (cleanup) эффекта
  };
}, [dependencies]);
```

Эффект без зависимостей (аналог componentDidMount)

Эффект, который выполняется только один раз при монтировании компонента:

Этот useEffect выполнится только один раз, когда компонент будет добавлен в DOM, и очистка произойдет при его удалении.

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

function App() {
  useEffect(() => {
    console.log('Компонент смонтирован');

    // Код для очистки эффекта (аналог componentWillUnmount)
    return () => {
      console.log('Компонент размонтирован');
    };
  }, []); // Пустой массив зависимостей

  return <div>Привет, мир!</div>;
}
```

Эффект с зависимостями

Эффект, который зависит от определенных значений и будет выполняться только при их изменении:

Здесь эффект будет выполняться каждый раз, когда значение count изменится.

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

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

  useEffect(() => {
    console.log('Счетчик изменился:', count);

    // Очистка не обязательна, если не требуется

  }, [count]); // Эффект выполняется при изменении count

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

Эффект без указания зависимостей (аналог componentDidUpdate)

Если вы не укажете массив зависимостей, эффект будет выполняться после каждого рендера:

В этом случае useEffect будет срабатывать после каждого изменения состояния или пропсов компонента.

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

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

  useEffect(() => {
    console.log('Компонент обновился');
  }); // Эффект выполняется после каждого рендера

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

Очистка эффекта (аналог componentWillUnmount)

Если эффект требует очистки, например, для отмены подписок или таймеров, вы можете вернуть функцию очистки из useEffect:

Этот пример демонстрирует, как использовать useEffect для создания таймера и его очистки при размонтировании компонента.

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

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

  useEffect(() => {
    const interval = setInterval(() => {
      setCount(c => c + 1);
    }, 1000);

    // Очистка эффекта
    return () => {
      clearInterval(interval);
    };
  }, []); // Пустой массив зависимостей, эффект выполняется один раз

  return <div>Прошло секунд: {count}</div>;
}
```

Ресурсы

Условный рендеринг - ТЫК, другой ТЫК

Рендеринг списков - ТЫК, другой ТЫК

Рендеринг (обновление состояния с отрисовкой) - ТЫК

Хук useState - ТЫК

Компонент - ТЫК

Жизненный цикл - ТЫК

Хук useEffect - ТЫК

Официальная документация на английском:

условный рендеринг - ТЫК

рендеринг списков - ТЫК

работа с состоянием (реактивность/интерактивность) - ТЫК

Код с урока (stackBlitz) - ТЫК