React - подходы

классовый подход
функциональный подход

Классовые компоненты

Классовые компоненты в React являются более старым способом создания компонентов и базируются на концепции классов из ES6 (ES2015). Они позволяют управлять состоянием компонента и использовать методы жизненного цикла, что делает их мощным инструментом для создания сложных пользовательских интерфейсов.

Структура классового компонента

Классовый компонент в React — это класс, который наследует React.Component. Он должен обязательно содержать метод render, который возвращает JSX, описывающий пользовательский интерфейс компонента.

import React, { Component } from 'react';: Импортируем React и базовый класс Component.

class MyComponent extends Component { ... }: Создаем класс MyComponent, который наследует возможности React-компонента.

render(): Обязательный метод, который возвращает JSX для рендеринга UI.

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

class MyComponent extends Component {
  render() {
    return (
      <div>
        <h1>Hello, World!</h1>
        <p>This is a class component.</p>
      </div>
    );
  }
}

export default MyComponent;
```

Состояние (state) в классовых компонентах

Классовые компоненты могут иметь собственное состояние (state), которое хранит данные, определяющие поведение и отображение компонента. Состояние компонента может меняться со временем в ответ на действия пользователя или другие события.

```
class Counter extends Component {
  constructor(props) {
    super(props);
    // Инициализация состояния
    this.state = {
      count: 0
    };
  }
}
```

Обновление состояния

Для изменения состояния используется метод setState. При вызове setState React объединяет новый объект состояния с текущим и затем перерендеривает компонент.

```
increment = () => {
  this.setState({ count: this.state.count + 1 });
};
```

Слияние состояния: React автоматически объединяет новый объект состояния с текущим. Если в объекте переданном в setState указаны не все ключи, остальные сохраняются без изменений.

```
this.setState((prevState) => ({
  count: prevState.count + 1
}));
```

setState не изменяет состояние немедленно. Это асинхронный процесс, поэтому для вычисления нового состояния на основе предыдущего нужно использовать функцию

Методы жизненного цикла

Методы жизненного цикла позволяют разработчикам управлять различными этапами существования компонента: от его создания и рендеринга до обновления и удаления из DOM. Эти методы особенно важны для управления состоянием, подписками на события и асинхронными операциями.

Основные методы жизненного цикла:

- constructor(props)
- componentDidMount()
- componentDidUpdate(prevProps, prevState)
- componentWillUnmount()
- shouldComponentUpdate(nextProps, nextState)

constructor(props)

Вызывается при создании компонента. Используется для инициализации состояния и привязки методов.

```
constructor(props) {
  super(props);
  this.state = { count: 0 };
}
```

componentDidMount()

Вызывается сразу после монтирования компонента в DOM. Здесь можно выполнять действия, которые требуют наличия DOM, такие как запросы к API или установка подписок.

```
componentDidMount() {
  console.log('Component mounted');
}
```

componentDidUpdate(prevProps, prevState)

Вызывается после обновления компонента. Здесь можно реагировать на изменения состояния или пропсов.

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    console.log('Count updated');
  }
}
```

componentWillUnmount()

Вызывается перед удалением компонента из DOM. Используется для очистки ресурсов, таких как таймеры или подписки на события.

```
componentWillUnmount() {
  console.log('Component will unmount');
}
```

shouldComponentUpdate(nextProps, nextState)

Позволяет управлять процессом рендеринга. Этот метод должен возвращать true или false, определяя, нужно ли повторно рендерить компонент.

```
shouldComponentUpdate(nextProps, nextState) {
  return nextState.count !== this.state.count;
}
```

Передача пропсов (props)

Пропсы (props) — это параметры, передаваемые в компонент извне. В классовых компонентах пропсы доступны через this.props. Пропсы передаются родительским компонентом и не могут быть изменены в дочернем компоненте.

```
class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

```
<Greeting name="Alice" />
```

Контекст (context)

Контекст позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на каждом уровне. В классовых компонентах доступ к контексту можно получить с помощью this.context.

Этапы:

- создать контекст
- Обернуть компоненты в провайдер контекста
- Добавить в дочерний компонент

```
const MyContext = React.createContext();
```

```
<MyContext.Provider value={/* какое-то значение */}>
  <MyComponent />
</MyContext.Provider>
```

```
class MyComponent extends Component {
  static contextType = MyContext;

  render() {
    return <div>{this.context}</div>;
  }
}
```

Работа с событиями

Обработчики событий в классовых компонентах должны быть привязаны к экземпляру компонента, чтобы сохранить контекст this. Это делается либо в конструкторе, либо через стрелочные функции.

```
class ClickButton extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log('Button clicked');
  }

  render() {
    return <button onClick={this.handleClick}>Click me</button>;
  }
}
```

Использование стрелочной функции:

```
class ClickButton extends Component {
  handleClick = () => {
    console.log('Button clicked');
  }

  render() {
    return <button onClick={this.handleClick}>Click me</button>;
  }
}
```

Расширение классовых компонентов

Классовые компоненты могут быть расширены (наследованы) другими классами. Это позволяет создавать более сложные компоненты на основе существующих.

```
class BaseComponent extends Component {
  render() {
    return <div>Base Component</div>;
  }
}

class ExtendedComponent extends BaseComponent {
  render() {
    return (
      <div>
        <h1>Extended Component</h1>
        {super.render()}
      </div>
    );
  }
}
```

Основные отличия:

Простота: Функциональные компоненты проще в написании и понимании, особенно для новичков.

Логика: В функциональных компонентах используется декларативный подход к написанию логики, что упрощает код.

Методы жизненного цикла: Классовые компоненты имеют встроенные методы жизненного цикла, а в функциональных компонентах за это отвечают хуки, такие как useEffect.

Производительность: Функциональные компоненты в большинстве случаев работают быстрее, так как не требуют создания экземпляров класса.

Функциональные компоненты

Функциональные компоненты в React являются более современным способом создания компонентов, и они всё больше вытесняют классовые компоненты благодаря своей простоте и поддержке хуков, которые добавляют функциональности, ранее доступной только в классовых компонентах.

Основы функциональных компонентов

Функциональный компонент — это просто функция, которая принимает props (если они есть) и возвращает JSX, который определяет, как должен выглядеть компонент в пользовательском интерфейсе.

- import React from 'react';: Импортируем React, который нужен для работы с JSX.
- function MyComponent() { ... }: Определяем функциональный компонент MyComponent.
- return (...): Функция возвращает JSX, описывающий UI компонента.

```jsx
import React from 'react';

function MyComponent() {
  return (
    <div>
      <h1>Hello, World!</h1>
      <p>This is a functional component.</p>
    </div>
  );
}

export default MyComponent;
```

Использование props

Как и классовые компоненты, функциональные компоненты могут получать входные данные через props. props передаются компоненту от его родителя и используются внутри компонента для настройки отображаемого UI.

Деструктуризация props:

Для удобства и читаемости кода можно использовать деструктуризацию props:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

export default Greeting;
```

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

Использование хуков

Хуки (hooks) были добавлены в React 16.8 и значительно расширили возможности функциональных компонентов, предоставив доступ к состоянию, методам жизненного цикла и другим возможностям, которые ранее были доступны только в классовых компонентах.

Основные хуки:

- useState: позволяет функциональному компоненту иметь собственное состояние.
- useEffect: позволяет выполнять побочные эффекты в функциональных компонентах, такие как запросы к API, подписки на события и обновление документа.
- useContext: позволяет использовать контекст в функциональном компоненте.

Методы жизненного цикла

Функциональные компоненты не имеют встроенных методов жизненного цикла, как классовые компоненты, но с помощью хуков можно имитировать их поведение.

- componentDidMount: Имитируется с помощью useEffect с пустым массивом зависимостей.
- componentDidUpdate: Имитируется с помощью useEffect, следя за изменением определенных зависимостей.
- componentWillUnmount: Имитируется с помощью useEffect, возвращающего функцию очистки.

Контекст API в функциональных компонентах

Контекст API позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на каждом уровне. В функциональных компонентах контекст используется с помощью useContext.

- useContext: Хук для использования значения контекста в функциональном компоненте.
- MyContext.Provider: Компонент, который предоставляет значение контекста.

```jsx
import React, { createContext, useContext } from 'react';

const MyContext = createContext();

function ChildComponent() {
  const value = useContext(MyContext);
  return <p>The context value is: {value}</p>;
}

function ParentComponent() {
  return (
    <MyContext.Provider value="Hello from Context!">
      <ChildComponent />
    </MyContext.Provider>
  );
}

export default ParentComponent;
```

HOC (Higher-Order Components)

Компоненты высшего порядка (Higher-Order Components, HOC) — это одна из мощных техник в React, позволяющая повторно использовать логику между компонентами. HOC — это функция, которая принимает компонент и возвращает новый компонент с добавленным поведением или данными.

```
import React from 'react';

function withExtraInfo(WrappedComponent) {
  return function EnhancedComponent(props) {
    return (
      <div>
        <WrappedComponent {...props} />
        <p>This is some extra information.</p>
      </div>
    );
  };
}

function MyComponent(props) {
  return <div>{props.message}</div>;
}

const EnhancedComponent = withExtraInfo(MyComponent);

export default EnhancedComponent;
```

Пользовательские хуки

Пользовательские хуки (custom hooks) — это функции, которые позволяют инкапсулировать и переиспользовать логику, связанную с состоянием или побочными эффектами, в функциональных компонентах React. Они могут использовать другие хуки внутри себя (например, useState, useEffect и другие), но сами по себе являются обычными JavaScript-функциями.

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

// Пользовательский хук useCounter
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  const reset = () => setCount(initialValue);

  return { count, increment, decrement, reset };
}

export default useCounter;
```

Ресурсы

классовый компонент - ТЫК

Функциональные и классовые компоненты - ТЫК