React.memo — это метод оптимизации производительности в React, который используется для мемоизации функциональных компонент. Основная его задача — предотвращать ненужные перерисовки компонента, если его пропсы остаются неизменными.

Идея проста: если пропсы компонента те же самые, что и в предыдущем рендере, то React может вернуть предвычисленный результат этого компонента (мемоизированную версию), вместо повторной функции-рендера.

Как это работает?

React.memo — это обертка для функционального компонента. Когда вы оборачиваете компонент в React.memo, React сравнивает текущие пропсы с предыдущими. Если они одинаковые, то компонент не перерендеривается.

Синтаксис:

import React from 'react';

const MyComponent = React.memo((props) => {

  console.log('Компонент рендерится!');

  return {props.text};

});

export default MyComponent;

Используя React.memo, вы минимизируете количество вызываемых рендеров. Чтобы понять, как это работает, рассмотрим пример.

Пример: Оптимизация компонента

Допустим, у нас есть компонент, который принимает текстовые данные в качестве пропсов:

import React, { useState } from 'react';

import MyComponent from './MyComponent';

function App() {

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

  const [text, setText] = useState('Привет, мир!');

  return (

       setCount(count + 1)}>Увеличить: {count}

       setText('Текст изменён')}>Изменить текст

  );

}

Если MyComponent не обернут в React.memo, то даже при изменении count (не связанного с text) получится, что `MyComponent рендерится заново, хотя его пропсы не изменились.

Но если обернуть его в React.memo, поведение будет таким:

При изменении count рендер MyComponent не произойдет, потому что проп text остался неизменным.

При изменении text рендер произойдет, так как пропсы изменились.

Когда использовать React.memo?

React.memo особенно полезен при выполнении следующих условий:

Компонент часто рендерится (например, в списке большого количества элементов).

Перерендеринг компонента дорогой по производительности (например, сложная логика или DOM-операции).

Пропсы редко меняются.

Однако, важно помнить: использование React.memo не всегда оправдано. Если компонент и так быстро рендерится, дополнительное сравнение пропсов (shallow comparison) может в итоге негативно повлиять на производительность.

Кастомное сравнение пропсов

По умолчанию, React.memo использует поверхностное сравнение (shallow comparison) пропсов. Если вам нужно контролировать эту логику, вы можете передать свою функцию сравнения вторым аргументом.

Синтаксис:

React.memo(Component, areEqual);

Функция areEqual должна возвращать true, если пропсы одинаковы, и false, если разные:

const MyComponent = React.memo((props) => {

  console.log('Компонент рендерится!');

  return {props.text};

}, (prevProps, nextProps) => {

  return prevProps.text === nextProps.text;

});

В этом случае React будет использовать функцию areEqual вместо стандартного сравнения.

Ограничения и предостережения

React.memo работает только с функциональными компонентами.

   Классовые компоненты можно оптимизировать с помощью PureComponent или метода shouldComponentUpdate.

Контекст и состояние.

   Если компонент использует useState или useContext, React.memo не предотвращает рендеринг в случае изменений состояния или контекста.

Глубокая иммутабельность.

   По умолчанию React проверяет только верхний уровень объекта (shallow comparison). Если вы передаете объекты или массивы через пропсы, измененные даже внутри, это будет считаться изменением, и рендер все равно произойдет.

Оверинжиниринг: оптимизация лишнего.

   Если ваш компонент небольшой и рендерится быстро, React.memo может добавить ненужную сложность. Проверяйте, действительно ли есть прирост производительности.

Итог

React.memo — мощный инструмент для оптимизации, который предотвращает лишние рендеры компонентов. Однако его использование требует понимания вашего приложения, шаблонов работы с данными и профилирования производительности. Не стоит использовать его везде подряд