Render Props

Что такое Render Props?

"Render Props" — это техника в React, которая позволяет передавать функцию через свойство компонента с целью изменения его поведения или отображения. Эта функция принимает определённые аргументы и возвращает React-элемент. Этот подход обеспечивает гибкость в организации переиспользуемого кода.

Простой пример

Допустим, у нас есть компонент `MouseTracker`, который отслеживает положение курсора мыши и передаёт эти данные через render props.

```
// MouseTracker.js
import React, { useState, useEffect } from "react";

const MouseTracker = ({ render }) => {
    const [position, setPosition] = useState({ x: 0, y: 0 });

    const handleMouseMove = (e) => {
        setPosition({
            x: e.clientX,
            y: e.clientY,
        });
    };

    useEffect(() => {
        window.addEventListener("mousemove", handleMouseMove);
```

```
        return () => {
            window.removeEventListener("mousemove", handleMouseMove);
        };
    }, []);

    return <div>{render(position)}</div>;
};

export default MouseTracker;
```

Теперь мы можем использовать этот компонент в другом компоненте и передать функцию через свойство `render`.

```
// App.js
import React from "react";
import MouseTracker from "./MouseTracker";

const App = () => {
    return (
        <MouseTracker
            render={({ x, y }) => (
                <h1>
                    Позиция мыши: ({x}, {y})
                </h1>
            )}
        />
    );
};

export default App;
```

Пример с условным рендерингом

В этом примере у нас есть компонент `List`, который может отображать список как в виде таблицы, так и в виде карточек.

```
// List.js
import React from "react";

const List = ({ items, render }) => {
    return <div>{items.map(render)}</div>;
};

export default List;
```

Теперь мы можем использовать этот компонент и передать функцию для рендеринга элементов списка.

```
// App.js
import React from "react";
import List from "./List";

const App = () => {
    const items = ["Apple", "Banana", "Cherry"];

    return (
        <List
            items={items}
            render={(item, index) => <div key={index}>{item}</div>}
        />
    );
};

export default App;
```

Или в виде таблицы:

```
<List
    items={items}
    render={(item, index) => (
        <tr key={index}>
            <td>{item}</td>
        </tr>
    )}
/>
```

Пример с счётчиком

Допустим, у нас есть компонент `Counter`, который управляет состоянием счётчика. Мы хотим сделать его универсальным, чтобы другие компоненты могли использовать его логику, но с разной визуализацией.

Counter.js

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

const Counter = ({ render }) => {
    const [count, setCount] = useState(0);
```

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

    return <div>{render(count, increment, decrement)}</div>;
};

export default Counter;
```

App.js

```
import React from "react";
import Counter from "./Counter";

const App = () => {
    return (
        <div>
            <Counter
                render={(count, increment, decrement) => (
                    <div>
                        <button onClick={decrement}>-</button>
                        <span>{count}</span>
                        <button onClick={increment}>+</button>
                    </div>
                )}
            />

            <Counter
                render={(count, increment, decrement) => (
                    <div>
                        <h1>{count}</h1>
                        <button onClick={increment}>Увеличить</button>
                        <button onClick={decrement}>Уменьшить</button>
                    </div>
                )}
            />
        </div>
    );
};

export default App;
```

В этом примере у нас есть компонент `Counter`, который принимает функцию `render` в качестве свойства. Эта функция вызывается с текущим состоянием счётчика и двумя функциями для его изменения (`increment` и `decrement`). Это позволяет нам использовать одну и ту же "бизнес-логику" счётчика в разных местах приложения, но с разным "внешним видом".

Таким образом, с помощью паттерна Render Props, мы можем сделать наши компоненты более гибкими и переиспользуемыми.

Преимущества Render Props

1. **Переиспользование кода:** Один компонент может быть настроен различными способами с помощью Render Props.
2. **Инкапсуляция:** Компоненты могут скрывать сложную логику и передавать только необходимый интерфейс через функцию render prop.
3. **Контроль:** Компонент, который получает render prop, имеет больше контроля над тем, как и когда отображать содержимое.
4. **Читаемость:** Код может быть более читаемым, поскольку он явно показывает, какие компоненты взаимодействуют друг с другом.

Недостатки Render Props

1. **Сложность:** Использование render props может сделать компоненты и их взаимодействия сложными и трудными для понимания.
2. **Производительность:** Если render prop — это inline-функция, это может привести к ненужным ререндерам.
3. **"Пропс-дрель":** Необходимость прокидывать render prop через несколько уровней компонентов может сделать код менее читаемым и поддерживаемым.