Compound Components

Compound Components (компоненты составного типа) — это паттерн в React, который позволяет создавать компоненты с явно определенным, гибким и расширяемым API. Этот паттерн часто используется для создания более универсальных и многоразовых компонентов. Основная идея заключается в том, чтобы создать группу компонентов, которые работают вместе и обмениваются данными через контекст или пропсы (props).

Например, допустим, у нас есть компонент `Accordion`, который состоит из нескольких `AccordionItem`:

```
<Accordion>
    <AccordionItem title="Item 1">Content 1</AccordionItem>
    <AccordionItem title="Item 2">Content 2</AccordionItem>
    <AccordionItem title="Item 3">Content 3</AccordionItem>
</Accordion>
```

Здесь `Accordion` и `AccordionItem` являются компонентами составного типа.

Пример на функциональных компонентах

В следующем примере мы создадим простой аккордеон, используя React и паттерн Compound Components.

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

const AccordionContext = createContext();
```

```
function Accordion({ children }) {
    const [activeIndex, setActiveIndex] = useState(null);
    return (
        <AccordionContext.Provider value={{ activeIndex, setActiveIndex }}>
            <div>{children}</div>
        </AccordionContext.Provider>
    );
}

function AccordionItem({ children, index }) {
    const { activeIndex, setActiveIndex } = useContext(AccordionContext);
    const isActive = index === activeIndex;
    return (
        <div>
            <button onClick={() => setActiveIndex(isActive ? null : index)}>
                Toggle
            </button>
            {isActive && children}
        </div>
    );
}

export function App() {
    return (
        <Accordion>
            <AccordionItem index={0}>Content 1</AccordionItem>
            <AccordionItem index={1}>Content 2</AccordionItem>
            <AccordionItem index={2}>Content 3</AccordionItem>
        </Accordion>
    );
}
```

В этом примере:

- `Accordion` — родительский компонент, который использует контекст для управления активным элементом.
- `AccordionItem` — дочерний компонент, который использует этот же контекст для определения, является ли он активным.

Таким образом, Compound Components предоставляют гибкий и удобный способ для создания сложных UI-компонентов.

Вот простой пример на функциональных компонентах с использованием React.

Для начала, убедитесь, что у вас установлен Node.js и npm. Затем выполните следующие команды для создания нового React-проекта:

```
npx create-react-app compound-components-example
cd compound-components-example
```

Пример кода

Допустим, у нас есть компонент `Toggle`, который может быть включен или выключен. У этого компонента будут две "составные" части: `Toggle.On` и `Toggle.Off`, которые будут отображаться в зависимости от состояния `Toggle`.

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

// Создание контекста
const ToggleContext = createContext();

// Родительский компонент
const Toggle = ({ children }) => {
    const [on, setOn] = useState(false);
    const toggle = () => setOn(!on);

    return (
        <ToggleContext.Provider value={{ on, toggle }}>
            <div>{children}</div>
        </ToggleContext.Provider>
    );
};

// Дочерние компоненты
const ToggleOn = ({ children }) => {
    const { on } = useContext(ToggleContext);
    return on ? children : null;
};

const ToggleOff = ({ children }) => {
    const { on } = useContext(ToggleContext);
    return on ? null : children;
};

const ToggleButton = () => {
    const { toggle } = useContext(ToggleContext);
    return <button onClick={toggle}>Toggle</button>;
};

// Использование компонента Toggle
const App = () => {
    return (
        <Toggle>
            <ToggleOn>On</ToggleOn>
            <ToggleOff>Off</ToggleOff>
            <ToggleButton />
```

```
        </Toggle>
    );
};

export default App;
```

В этом примере:

- `Toggle` является родительским компонентом, который хранит состояние (`on` или `off`).
- `ToggleOn` и `ToggleOff` являются дочерними компонентами, которые используют состояние из `Toggle` через Context API.
- `ToggleButton` — еще один дочерний компонент, который переключает состояние.

Этот пример демонстрирует базовое использование составных компонентов для создания гибкого и легко расширяемого интерфейса.

Как это работает?

1. **Создание Контекста**: В начале кода создается React контекст с помощью `createContext()`.

   ```
   const ToggleContext = createContext();
   ```

   Этот контекст будет использоваться для передачи данных между родительским и дочерними компонентами.

2. **Родительский Компонент (`Toggle`)**: Родительский компонент `Toggle` использует состояние `on`, чтобы определить, активен ли переключатель или нет.

   ```
   const [on, setOn] = useState(false);
   ```

   Это состояние и функция для его изменения (`toggle`) передаются через `ToggleContext.Provider`.

```
<ToggleContext.Provider value={{ on, toggle }}>
```

3. **Дочерние Компоненты (`ToggleOn`, `ToggleOff`, `ToggleButton`)**: Эти компоненты используют `useContext` для доступа к `ToggleContext` и получения текущего состояния (`on`) и функции для его изменения (`toggle`).

```
const { on } = useContext(ToggleContext);
const { toggle } = useContext(ToggleContext);
```

На основе этого состояния, `ToggleOn` и `ToggleOff` решают, следует ли им отображать своих детей.

```
return on ? children : null;
```

```
return on ? null : children;
```

Кнопка `ToggleButton` использует функцию `toggle` для изменения состояния `on` в родительском компоненте.

```
<button onClick={toggle}>Toggle</button>
```

4. **Использование в `App`**: В компоненте `App`, все эти компоненты используются вместе. Компоненты `ToggleOn` и `ToggleOff` будут отображаться в зависимости от состояния в родительском компоненте `Toggle`.

```
<Toggle>
    <ToggleOn>On</ToggleOn>
    <ToggleOff>Off</ToggleOff>
    <ToggleButton />
</Toggle>
```

Эта структура делает компонент `Toggle` очень гибким и расширяемым, так как вы можете свободно добавлять, удалять или изменять дочерние компоненты, не затрагивая логику родительского компонента.

Преимущества Compound Components

1. **Явное API**: Составные компоненты предоставляют явный и понятный способ организации интерфейса.
2. **Гибкость**: Этот подход позволяет создавать гибкие и расширяемые компоненты. Разработчики могут легко добавлять или удалять дочерние компоненты без изменения внутренней логики родительского компонента.
3. **Повторное использование**: Составные компоненты легко переиспользовать и комбинировать в различных конфигурациях.
4. **Лучшая читаемость кода**: Легче понять, как компонент должен быть использован, и какие дочерние компоненты с ним взаимодействуют.
5. **Инкапсуляция логики**: Родительский компонент инкапсулирует всю логику, делая код более модульным и легким для тестирования.
6. **Интеграция с Context API**: Взаимодействие между родительским и дочерними компонентами часто упрощается с использованием React Context, что позволяет избежать "prop drilling" (передачи свойств через несколько уровней иерархии).

Недостатки Compound Components

1. **Сложность**: Для новичков этот подход может показаться сложным и запутанным.
2. **Перерендер**: Использование React Context может привести к ненужным перерендерам, если не оптимизировать компоненты.
3. **Тесная связанность**: Дочерние компоненты часто тесно связаны с родительским компонентом, что может сделать их менее переиспользуемыми в других контекстах.
4. **Типизация**: Если вы используете системы типов, такие как TypeScript, вам может потребоваться дополнительная работа для правильной типизации дочерних компонентов.
5. **Отладка и тестирование**: Из-за инкапсуляции логики в родительском компоненте, отладка и тестирование могут быть более сложными по сравнению с "плоскими" компонентами.