物件陣列狀態操作範例

這個 React 組件展示了如何對物件陣列進行各種狀態操作，包括新增、刪除、修改和過濾等常見操作。這些操作在實際開發中非常常見，例如：

- 待辦事項列表管理
- 購物車商品操作
- 使用者資料管理
- 動態表單處理

基本結構

狀態管理

組件使用 `useState` 來管理一個物件陣列的狀態：

```
const [data, setData] = useState(sample)
```

這裡的 `data` 是當前狀態，`setData` 是用來更新狀態的函數。使用 `useState` 的好處是：

- 當狀態更新時，React 會自動重新渲染組件
- 狀態變更會被追蹤，方便除錯
- 可以確保狀態的不可變性

初始資料結構

```
const sample = [
  { id: 1, text: 'a' },
  { id: 2, text: 'b' },
  { id: 3, text: 'c' },
  { id: 4, text: 'aa' },
]
```

這個資料結構的特點：

- 每個物件都有唯一的 `id` 屬性
- 包含一個 `text` 屬性用於顯示內容
- 使用陣列來組織多個物件

主要功能說明

1. 資料展示

表格渲染

```
<table border="1">
  <thead>
    <tr>
      <th>ID</th>
      <th>Text</th>
    </tr>
  </thead>
  <tbody>
    {data.map((v, i) => (
      <tr key={v.id}>
        <td>{v.id}</td>
        <td>{v.text}</td>
      </tr>
    ))}
  </tbody>
</table>
```

重要概念：

- 使用 `map` 方法遍歷陣列
- 每個項目需要唯一的 `key` 屬性
- 表格結構清晰展示資料

2. 資料操作功能

2.1 新增操作

1. **在列表最前面新增**

```
const newObj = { id: 99, text: 'xxx' }
const nextData = [newObj, ...data]
setData(nextData)
```

特點：

- 使用展開運算符 `...` 保留原有資料
- 新資料放在陣列最前面
- 適用於最新消息、最新訂單等場景

2. **在列表最後面新增**

```
const newObj = { id: 88, text: 'yyy' }
const nextData = [...data, newObj]
setData(nextData)
```

特點：

- 新資料放在陣列最後面
- 適用於追加資料的場景

3. **自動生成不重複 ID**

```
const ids = data.map((v) => v.id)
const newId = data.length > 0 ? Math.max(...ids) + 1 : 1
const newObj = { id: newId, text: 'xxx' }
```

特點：

- 自動計算最大 ID 值
- 確保 ID 不重複
- 適用於動態新增資料的場景

2.2 刪除操作

1. **使用 filter 方法**

```
const nextData = data.filter((v) => v.id !== 4)
setData(nextData)
```

特點：

- 不會修改原陣列
- 返回新陣列
- 適用於條件性刪除

2. **使用 splice 方法**

```
const foundIndex = data.findIndex((v) => v.id === 4)
const nextData = [...data]
nextData.splice(foundIndex, 1)
setData(nextData)
```

特點：

- 需要先找到索引
- 會修改原陣列，所以需要先拷貝
- 適用於精確位置刪除

2.3 修改操作

1. **使用 map + 展開運算符**

```
const nextData = data.map((v) => (v.id === 3 ? { ...v, text: 'cccc' } :
v))
setData(nextData)
```

特點：

- 使用展開運算符保留其他屬性
- 只修改特定屬性
- 適用於部分更新

2. **使用深拷貝 + 直接指定**

```
const nextData = JSON.parse(JSON.stringify(data))
nextData[foundIndex].text = 'cccc'
setData(nextData)
```

特點：

- 完全複製整個物件結構
- 可以直接修改深層屬性
- 適用於複雜物件結構

2.4 插入操作

1. **使用 splice 方法**

```
const foundIndex = data.findIndex((v) => v.id === 2)
const nextData = [...data]
nextData.splice(foundIndex + 1, 0, newObj)
setData(nextData)
```

特點：

- 在指定位置插入
- 需要先找到插入位置
- 適用於有序插入

2. **使用 slice 方法**

```
const aa = data.slice(0, foundIndex + 1)
const ab = data.slice(foundIndex + 1)
```

```
const nextData = [...aa, newObj, ...ab]
setData(nextData)
```

特點：

- 不修改原陣列
- 更靈活的插入方式
- 適用於複雜插入邏輯

重要注意事項

1. 狀態不可變性

- 永遠不要直接修改狀態

```
// 錯誤示範
data[0].text = 'new text' // 直接修改狀態
setData(data)

// 正確示範
const nextData = [...data]
nextData[0].text = 'new text'
setData(nextData)
```

2. ID 唯一性

- 使用時間戳記生成 ID

```
const newId = Date.now()
```

- 使用 UUID

```
import { v4 as uuidv4 } from 'uuid'
const newId = uuidv4()
```

3. 深拷貝與淺拷貝

- 淺拷貝 (只複製第一層)

```
const shallowCopy = [...data]
```

- 深拷貝 (複製所有層級)

```
const deepCopy = JSON.parse(JSON.stringify(data))
```

4. 效能優化

- 使用 `useMemo` 緩存計算結果

```
const filteredData = useMemo(
  () => data.filter((item) => item.text.includes('a')),
  [data]
)
```

- 使用 `useCallback` 緩存函數

```
const handleDelete = useCallback(
  (id) => {
    setData(data.filter((item) => item.id !== id))
  },
  [data]
)
```

最佳實踐

1. 程式碼組織

- 將複雜的狀態邏輯抽離到自定義 Hook
- 使用 TypeScript 定義介面
- 適當拆分組件

2. 錯誤處理

- 檢查邊界情況
- 提供預設值
- 處理異常情況

3. 效能優化

- 避免不必要的重新渲染
- 使用適當的資料結構
- 考慮使用虛擬列表處理大量資料

4. 測試策略

- 單元測試狀態更新邏輯
- 整合測試使用者互動

- 效能測試大規模資料操作