React 購物車實作筆記

目錄

1. 基本概念
2. 核心功能實作
3. 衍生狀態計算
4. 最佳實踐
5. 常見問題與解決方案
6. 擴展建議
7. 補充說明
8. 實作範例
9. 進階功能
10. 測試範例

基本概念

1. 商品資料結構

```
const initialProducts = [
  {
    id: 0,
    name: '小熊餅乾',
    price: 50,
  },
  {
    id: 1,
    name: '巧克力豆餅乾',
    price: 100,
  },
  {
    id: 2,
    name: '小老板海苔',
    price: 150,
  },
]
```

2. 購物車狀態管理

```
// 購物車中的項目與商品的物件屬性會相差一個count屬性
// count代表購買數量
const [items, setItems] = useState([])
```

核心功能實作

1. 加入購物車

```
const onAdd = (product) => {
  // 判斷要加入的商品是否已在購物車中
  const foundIndex = items.findIndex((v) => v.id === product.id)

  if (foundIndex !== -1) {
    // 如果已存在，增加數量
    onIncrease(product.id)
  } else {
    // 如果不存在，新增到購物車
    const newItem = { ...product, count: 1 }
    const nextItems = [newItem, ...items]
    setItems(nextItems)
  }
}
```

2. 增加商品數量

```
const onIncrease = (itemId) => {
  const nextItems = items.map((v) => {
    if (v.id === itemId) {
      return { ...v, count: v.count + 1 }
    } else {
      return v
    }
  })
  setItems(nextItems)
}
```

3. 減少商品數量

```
const onDecrease = (itemId) => {
  const nextItems = items.map((v) => {
    if (v.id === itemId) {
      return { ...v, count: v.count - 1 }
    } else {
      return v
    }
  })
  setItems(nextItems)
}
```

4. 移除商品

```
const onRemove = (itemId) => {
  const nextItems = items.filter((v) => v.id !== itemId)
  setItems(nextItems)
}
```

5. 數量為0時自動刪除

```
function handleDecreaseClick(itemId) {
  // 先減少數量
  let nextItems = items.map((item) => {
    if (item.id === itemId) {
      return {
        ...item,
        count: item.count - 1,
      }
    } else {
      return item
    }
  })

  // 過濾出數量大於0的商品
  nextItems = nextItems.filter((p) => p.count > 0)

  // 檢查是否需要刪除商品
  if (nextItems.length < items.length) {
    if (confirm('你確定要刪除此商品?')) {
      setItems(nextItems)
    }
  } else {
    setItems(nextItems)
  }
}
```

衍生狀態計算

1. 計算總數量

```
// 使用 reduce 方法計算總數量
const totalQty = items.reduce((acc, v) => acc + v.count, 0)
```

2. 計算總金額

```
// 使用 reduce 方法計算總金額
const totalAmount = items.reduce((acc, v) => acc + v.count * v.price, 0)
```

最佳實踐

1. 狀態更新模式

- 使用不可變更新
- 使用 map 和 filter 方法
- 保持狀態的純淨性

2. 使用者體驗

- 提供確認對話框
- 即時更新數量
- 清晰的商品列表

3. 程式碼組織

- 將相關功能分組
- 使用有意義的函數名稱
- 保持程式碼簡潔

常見問題與解決方案

1. 狀態更新

- 問題：直接修改狀態
- 解決：使用不可變更新模式

2. 使用者確認

- 問題：意外刪除商品
- 解決：使用 confirm 對話框

3. 數量控制

- 問題：數量可能為負數
- 解決：在減少數量時檢查最小值

擴展建議

1. 加入本地儲存

```
useEffect(() => {
  localStorage.setItem('cart', JSON.stringify(items))
}, [items])

useEffect(() => {
  const savedCart = localStorage.getItem('cart')
  if (savedCart) {
```

```
      setItems(JSON.parse(savedCart))
    }
  }, [])
```

2. 加入動畫效果

```
.cart-item {
  transition: all 0.3s ease;
}

.cart-item-enter {
  opacity: 0;
  transform: translateX(-100%);
}

.cart-item-enter-active {
  opacity: 1;
  transform: translateX(0);
}
```

3. 效能優化

```
// 使用 useMemo 緩存計算結果
const totalAmount = useMemo(
  () => items.reduce((acc, v) => acc + v.count * v.price, 0),
  [items]
)
```

補充說明

1. 狀態更新注意事項

- 使用 `map` 方法時，確保返回新的物件而不是修改原物件
- 使用 `filter` 方法時，確保過濾條件正確
- 使用 `setItems` 時，確保傳入的是新的陣列

2. 效能優化建議

- 使用 `useMemo` 緩存計算結果
- 使用 `useCallback` 緩存函數
- 使用 `React.memo` 優化渲染

3. 使用者體驗優化

- 加入載入狀態
- 加入錯誤處理

- 加入成功提示

4. 程式碼組織建議

- 將相關功能分組
- 使用自定義 Hook
- 使用 Context API

5. 測試建議

- 單元測試
- 整合測試
- 端到端測試

實作範例

1. 商品列表渲染

```
<ul>
  {initialProducts.map((product) => {
    return (
      <li key={product.id}>
        {product.name} (NT${product.price})
        <button onClick={() => onAdd(product)}>加入購物車</button>
      </li>
    )
  })}
</ul>
```

2. 購物車列表渲染

```
<ul>
  {items.map((item) => (
    <li key={item.id}>
      {item.name} (<b>{item.count}</b>)(NT${item.price})
      <button onClick={() => onIncrease(item.id)}>+</button>
      <button onClick={() => handleDecreaseClick(item.id)}>-</button>
      <button onClick={() => onRemove(item.id)}>刪除</button>
    </li>
  ))}
</ul>
```

3. 總計顯示

```
<div>
  總數量: {totalQty}/ 總金額:NT${totalAmount.toLocaleString()}
```

```
</div>
```

進階功能

1. 購物車持久化

```
// 使用 localStorage 保存購物車狀態
const useCartPersist = () => {
  const [items, setItems] = useState(() => {
    const savedCart = localStorage.getItem('cart')
    return savedCart ? JSON.parse(savedCart) : []
  })

  useEffect(() => {
    localStorage.setItem('cart', JSON.stringify(items))
  }, [items])

  return [items, setItems]
}
```

2. 購物車優惠券

```
const useCoupon = (items) => {
  const [coupon, setCoupon] = useState('')
  const [discount, setDiscount] = useState(0)

  const applyCoupon = (code) => {
    // 根據優惠券代碼計算折扣
    if (code === 'DISCOUNT10') {
      setDiscount(0.1)
    } else if (code === 'DISCOUNT20') {
      setDiscount(0.2)
    }
    setCoupon(code)
  }

  const totalAmount = items.reduce((acc, v) => acc + v.count * v.price, 0)
  const finalAmount = totalAmount * (1 - discount)

  return {
    coupon,
    discount,
    applyCoupon,
    totalAmount,
    finalAmount,
  }
}
```

3. 購物車動畫

```
import { motion, AnimatePresence } from 'framer-motion'

const CartItem = ({ item, onIncrease, onDecrease, onRemove }) => {
  return (
    <motion.li
      initial={{ opacity: 0, x: -100 }}
      animate={{ opacity: 1, x: 0 }}
      exit={{ opacity: 0, x: 100 }}
    >
      {item.name} (<b>{item.count}</b>)(NT${item.price})
      <button onClick={() => onIncrease(item.id)}>+</button>
      <button onClick={() => onDecrease(item.id)}>-</button>
      <button onClick={() => onRemove(item.id)}>刪除</button>
    </motion.li>
  )
}
```

4. 購物車效能優化

```
// 使用 useMemo 優化計算
const CartSummary = ({ items }) => {
  const totalQty = useMemo(
    () => items.reduce((acc, v) => acc + v.count, 0),
    [items]
  )

  const totalAmount = useMemo(
    () => items.reduce((acc, v) => acc + v.count * v.price, 0),
    [items]
  )

  return (
    <div>
      總數量: {totalQty}/ 總金額:NT${totalAmount.toLocaleString()}
    </div>
  )
}

// 使用 React.memo 優化渲染
const CartItem = React.memo(({ item, onIncrease, onDecrease, onRemove }) => {
  return (
    <li>
      {item.name} (<b>{item.count}</b>)(NT${item.price})
      <button onClick={() => onIncrease(item.id)}>+</button>
      <button onClick={() => onDecrease(item.id)}>-</button>
      <button onClick={() => onRemove(item.id)}>刪除</button>
```

```
      </li>
    )
  })
```

5. 購物車錯誤處理

```
const useCartError = () => {
  const [error, setError] = useState(null)

  const handleError = (error) => {
    setError(error.message)
    setTimeout(() => setError(null), 3000)
  }

  return { error, handleError }
}

const Cart = () => {
  const { error, handleError } = useCartError()

  const onAdd = (product) => {
    try {
      // 加入購物車邏輯
    } catch (err) {
      handleError(err)
    }
  }

  return (
    <div>
      {error && <div className="error">{error}</div>}
      {/* 其他購物車內容 */}
    </div>
  )
}
```

測試範例

1. 單元測試

```
import { render, fireEvent } from '@testing-library/react'

test('加入商品到購物車', () => {
  const { getByText } = render(<Cart />)
  const addButton = getByText('加入購物車')

  fireEvent.click(addButton)
  expect(getByText('小熊餅乾')).toBeInTheDocument()
```

```
})

test('增加商品數量', () => {
  const { getByText } = render(<Cart />)
  const increaseButton = getByText('+')

  fireEvent.click(increaseButton)
  expect(getByText('2')).toBeInTheDocument()
})
```

2. 整合測試

```
test('完整購物流程', () => {
  const { getByText } = render(<Cart />)

  // 加入商品
  fireEvent.click(getByText('加入購物車'))

  // 增加數量
  fireEvent.click(getByText('+'))

  // 減少數量
  fireEvent.click(getByText('-'))

  // 刪除商品
  fireEvent.click(getByText('刪除'))

  expect(getByText('購物車是空的')).toBeInTheDocument()
})
```

3. 效能測試

```
test('大量商品渲染效能', () => {
  const items = Array(1000)
    .fill()
    .map((_, i) => ({
      id: i,
      name: `商品${i}`,
      price: 100,
      count: 1,
    }))

  const { getByText } = render(<Cart items={items} />)

  const startTime = performance.now()
  fireEvent.click(getByText('+'))
  const endTime = performance.now()
```

```
  expect(endTime - startTime).toBeLessThan(100)
})
```

注意事項

1. **狀態管理**
    - 使用不可變更新模式
    - 避免直接修改狀態
    - 使用適當的狀態更新方法
2. **效能優化**
    - 使用 useMemo 緩存計算結果
    - 使用 useCallback 緩存函數
    - 使用 React.memo 優化渲染
3. **使用者體驗**
    - 提供適當的錯誤處理
    - 加入載入狀態
    - 提供清晰的提示訊息
4. **程式碼組織**
    - 將相關功能分組
    - 使用自定義 Hook
    - 保持程式碼簡潔
5. **測試**
    - 撰寫單元測試
    - 撰寫整合測試
    - 進行效能測試