렌더링 최적화

1. 불변성

❖불변성(immutability)이란?

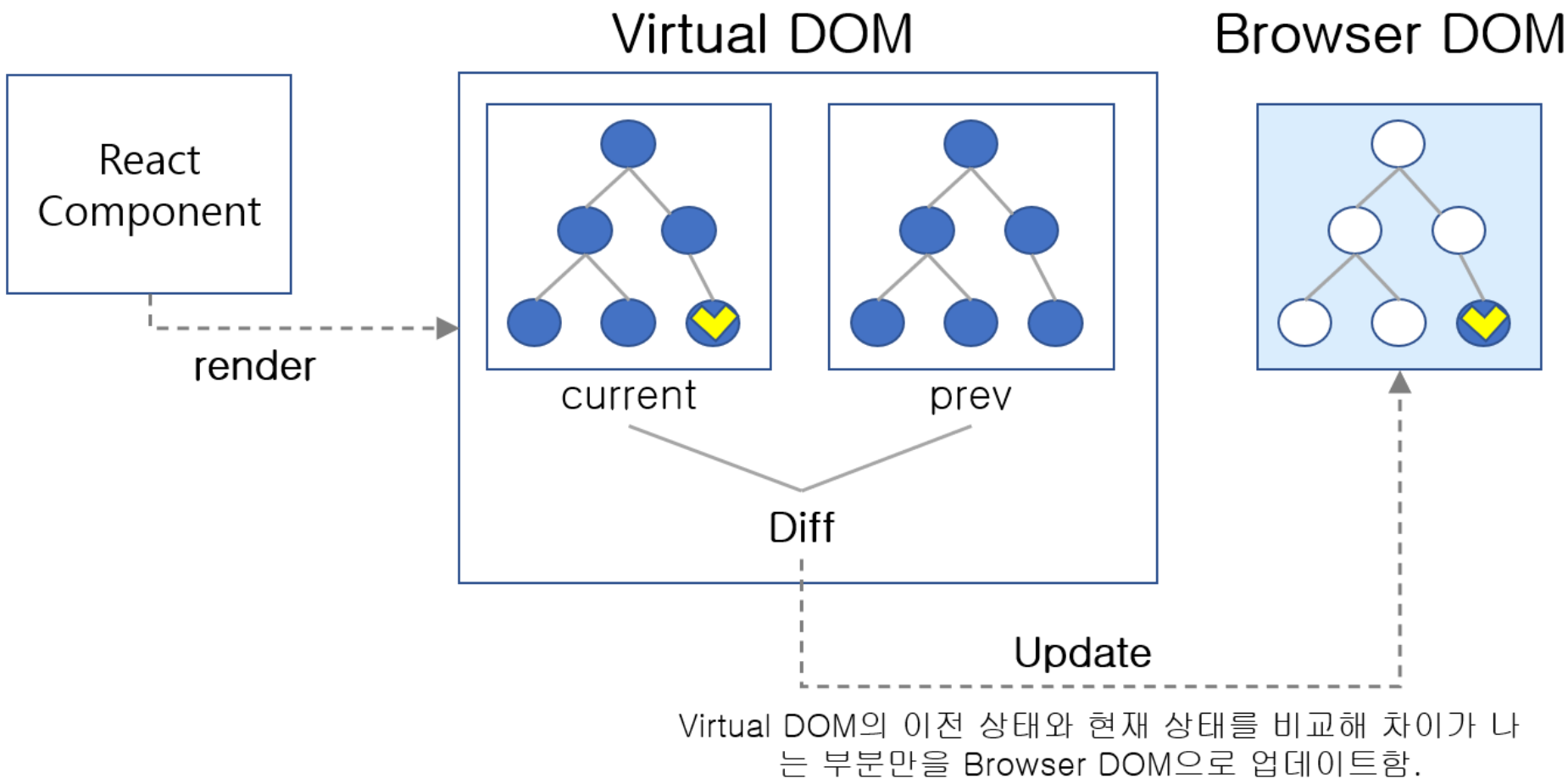
- 객체 내부의 속성을 직접 변경하지 않고, 이전 객체와는 다른 새로운 객체를 만들어 변경된 속성을 할당하는 방법
- 이전 객체가 불변(immutable)함
- 불변성을 확보하는 방법
  - Spread 연산자 : 간단한 객체,배열일 때 사용
  - immer 라이브러리 : 복잡한 객체 구조일 때 사용

❖불변성이 필요한 이유

- 상태 변경 추적을 위해서
- 렌더링 최적화를 위해서
  - 불변성을 가진 상태 변경이 없다면 가상 DOM 환경에서 렌더링 최적화는 어려움

1. 불변성

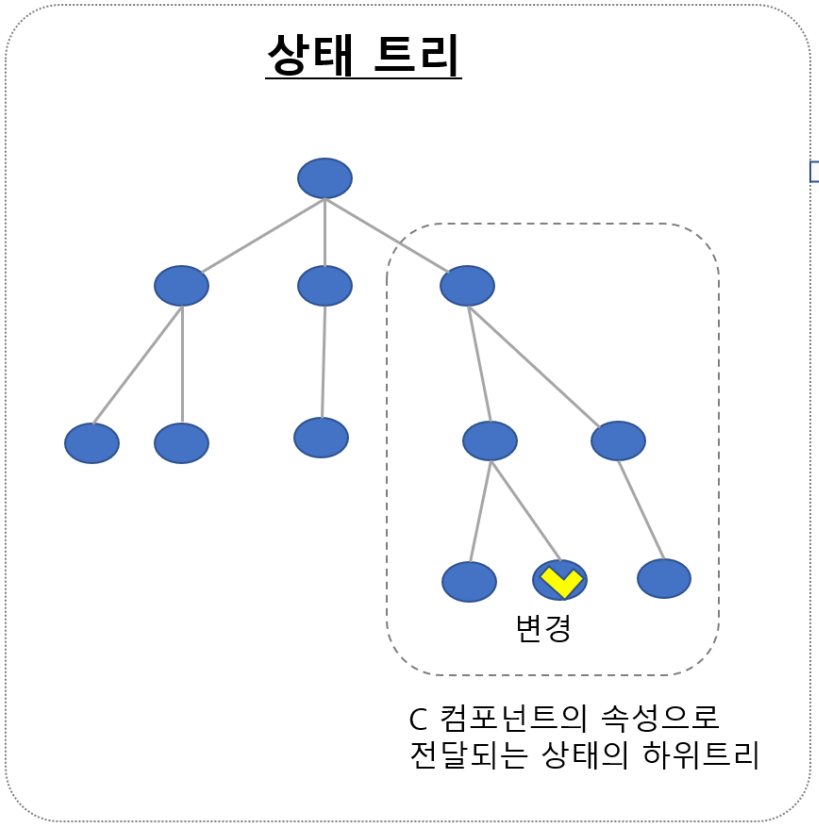
❖Virtual DOM 리뷰

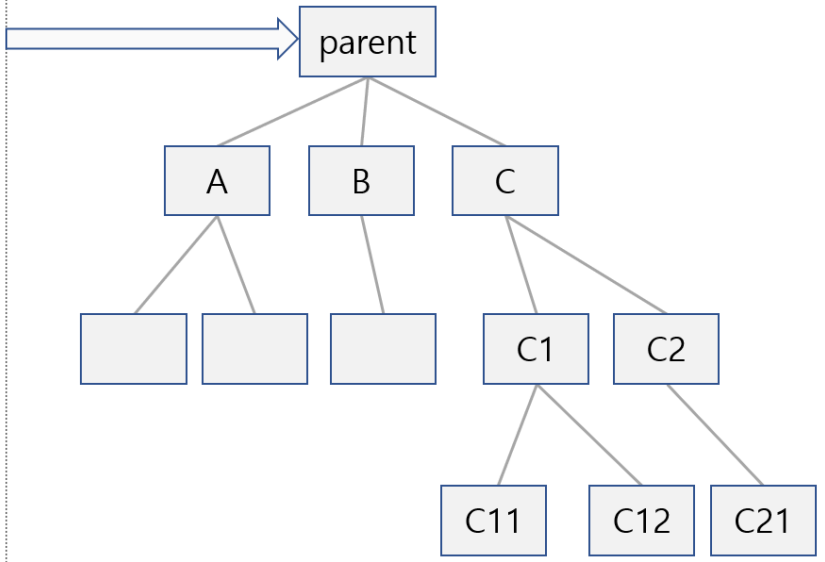


2. 렌더링 최적화와 불변성

❖ 시나리오

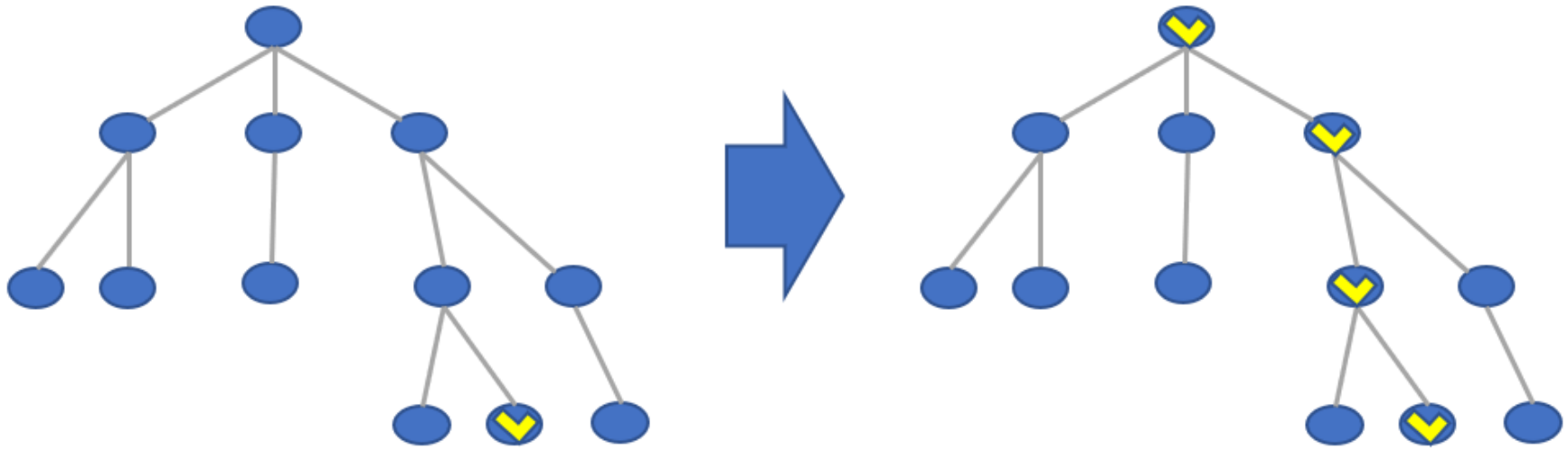
- 복잡한 객체 구조의 상태 데이터가 부모 컴포넌트에서 정의되었고,
- 상태 데이터의 하위 필드들을 자식 컴포넌트의 속성(props)으로 전달하며,
- 상태 데이터의 객체 트리 구조가 컴포넌트의 트리 구조와 비슷하다고 가정함.





2. 렌더링 최적화와 불변성

- 이 때 부모 컴포넌트에서 상태 객체의 끝단의 속성 값만을 변경했다라고 가정하면...
  - 부모 컴포넌트부터 끝단 컴포넌트의 부모까지의 컴포넌트에서 re-render를 할지 말지 결정하기 힘듬
    - 속성으로 전달된 객체의 트리구조를 타고 이동하면서 하나라도 변경된 값이 있는지 여부를 확인해야 하므로.....
  - 오히려 객체 트리를 이동하면서 비교하는 작업(deep compare)이 오히려 성능저하를 일으킬 수 있음
- 해결책
  - 상태 데이터 끝단의 값을 변경하면 Root로 거슬러 올라가는 경로 상의 값(객체인 경우는 참조 주소)만을 바꿔줌
  - 이것이 불변성의 핵심 : **얕은 비교(shallow compare)**



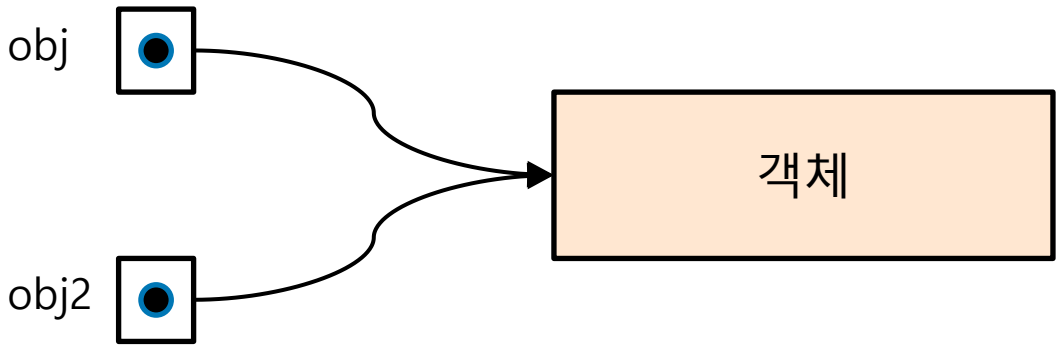
2. 렌더링 최적화와 불변성

❖불변성과 shallow copy

- 다음 코드는 불변성을 가지는가?

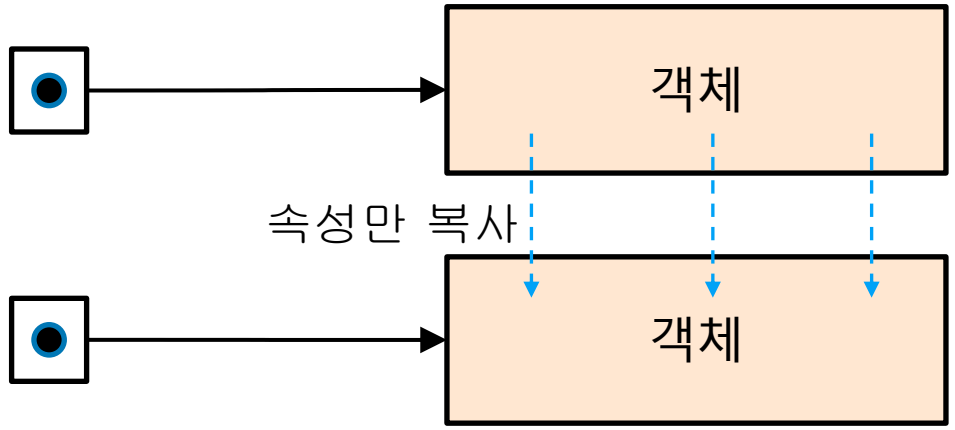
```
let obj2 = obj;            //shallow copy
obj2.name = "이순신";
```

- shallow copy 이므로 같은 객체를 참조함
  - 객체의 메모리 주소를 복사
  - 원본이 함께 변경되므로 불변성(X)



❖불변성을 위해 전개(spread) 연산자 사용

```
//기존 객체의 속성값을 복사한 후 name 속성을 이순신으로
변경한 새로운 객체를 생성함.
//따라서 obj3의 속성을 변경한다하더라도 obj의 값은 변경되지
않음
let obj3 = { ...obj, name: "이순신" };
```



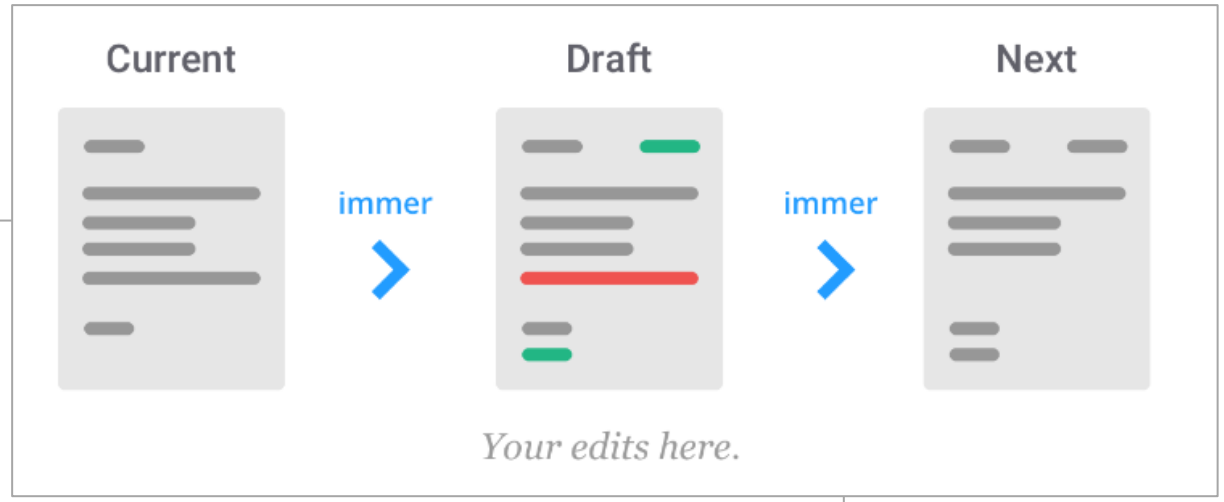
❖하지만 전개 연산자는 복잡한 트리구조의 객체는 불변성을 제공하지 않음

- 전개연산자를 사용한 수준의 속성까지만 불변성을 제공함

3. immer 라이브러리

❖immer

- immutability + ~er : 불변성을 제공하는 자
- 불변성을 가진 변경을 도와주는 라이브러리
- 사용 방법




```
import { produce } from "immer"

const currentState = [
    {  todo: "Learn es6",  done: true   },
    {  todo: "Try immer",   done: false  }
]

//** produce 함수의 첫번째 인자 : 변경 대상 객체
//** 두번째 인자 : 불변성 변경 함수
//  * 상태 변경 함수의 인자: 상태 변경을 위한 draft 버전의 객체
//** 불변성 변경 함수 안에서 currentState의 사본인 draft를 직접 변경하면 됨
//** 리턴값 : 새로운 상태 객체
const nextState = produce(currentState, (draft) => {
    draft[1].done = true
})
```

3. immer 라이브러리

❖immer 기능을 확인하기 위한 예제 : immer-test 예제 검토

- src/main.tsx 검토

```
import { produce } from "immer";

const quiz = {
  students: ["홍길동", "성춘향", "박문수"],
  quizlist: [
    {
      question: "한국 프로야구 팀이 아닌것은?",
      options: [
        { no: 1, option: "삼성라이온스" },
        { no: 2, option: "기아타이거스" },
        { no: 3, option: "두산베어스" },
        { no: 4, option: "LA다져스" },
      ],
      answer: 4,
    },
    {
      question: "2018년 x-mas는 무슨 요일인가?",
      options: [
        { no: 1, option: "월" },
        { no: 2, option: "화" },
        { no: 3, option: "수" },
        { no: 4, option: "목" },
      ],
      answer: 2,
    },
  ],
};

const quiz2 = produce(quiz, (draft) => {
  draft.quizlist[0].options[0].option = "LG트윈스";
});

//false,false,false,false,false,true
//true인 것은 변경된 속성으로부터 루트로 거슬러올라가는 경로상에 있지 않은 것
console.log(quiz === quiz2);
console.log(quiz.quizlist === quiz2.quizlist);
console.log(quiz.quizlist[0] === quiz2.quizlist[0]);
console.log(quiz.quizlist[0].options[0] === quiz2.quizlist[0].options[0]);
console.log(quiz.quizlist[0].options[0].option ===
                                        quiz2.quizlist[0].options[0].option);
console.log(quiz.students === quiz2.students);
```

```
false
false
false
false
false
true
```

3. immer 라이브러리

❖immer를 반드시 사용해야 하는가?

- 그렇지 않다. 하지만 바람직함.
  - 간단한 객체는 Spread 연산자(...)을 이용할 수 있음
- 특히 UI 렌더링 성능 최적화를 위해서는 반드시 필요함.

❖이 과정에서는 immer를 사용하여 예제를 작성함.

- 상태 데이터에 대해 불변성을 확보하는 것이 중요하다는 점을 인식하도록 하자.

4. React.memo() 고차 함수

❖React.memo()란?

- 리액트가 기본 제공하는 고차 함수
- 컴포넌트가 동일한 상태, 속성을 가지고 있다면 불필요한 렌더링을 방지할 수 있도록 함
  - shallow compare 로 비교하여 렌더링 최적화
  - 따라서 렌더링 최적화를 위해서는 불변성을 가진 상태 변경이 필수
- 사용 방법

```
const Child = (props:Props) => {

}

export default React.memo(Child);
- 또는 -
const Child = React.memo((props:PropsType)=> {

});

export default Child;
```

4. React.memo() 고차 함수

❖React.memo()만으로 렌더링 최적화가 가능한가?

- 불가능하다.
- 여러가지 기법들이 결합되어야 함
  - React.memo() 고차함수
  - useCallback(), useMemo() 메모이제이션 훅
  - 적절한 컴포넌트 분할
  - useRef() 훅 활용

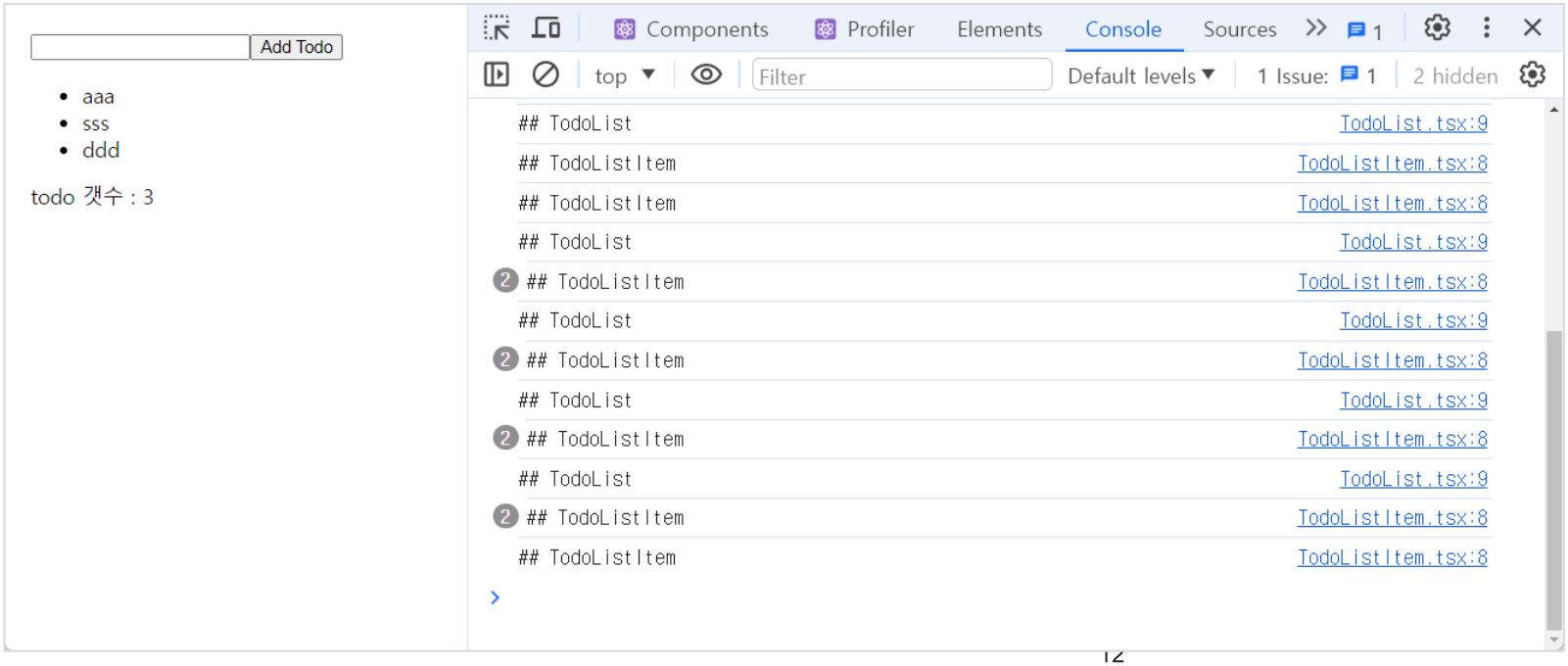
5. 렌더링 최적화 예제 시작

❖미리 제공되는 예제 : render-perf-optimize-0 예제

- App – TodoList – TodoListItem 컴포넌트
- TodoList, TodoListItem 에 렌더링 여부를 확인하기 위한 console.log() 문이 포함되었음

❖실행 결과

- App 컴포넌트의 상태가 바뀔 때마다 모든 컴포넌트가 다시 렌더링됨
  - 예) 새로운 todo 추가를 위해 입력필드에 todo 명을 타이핑할 때도 다시 렌더링됨



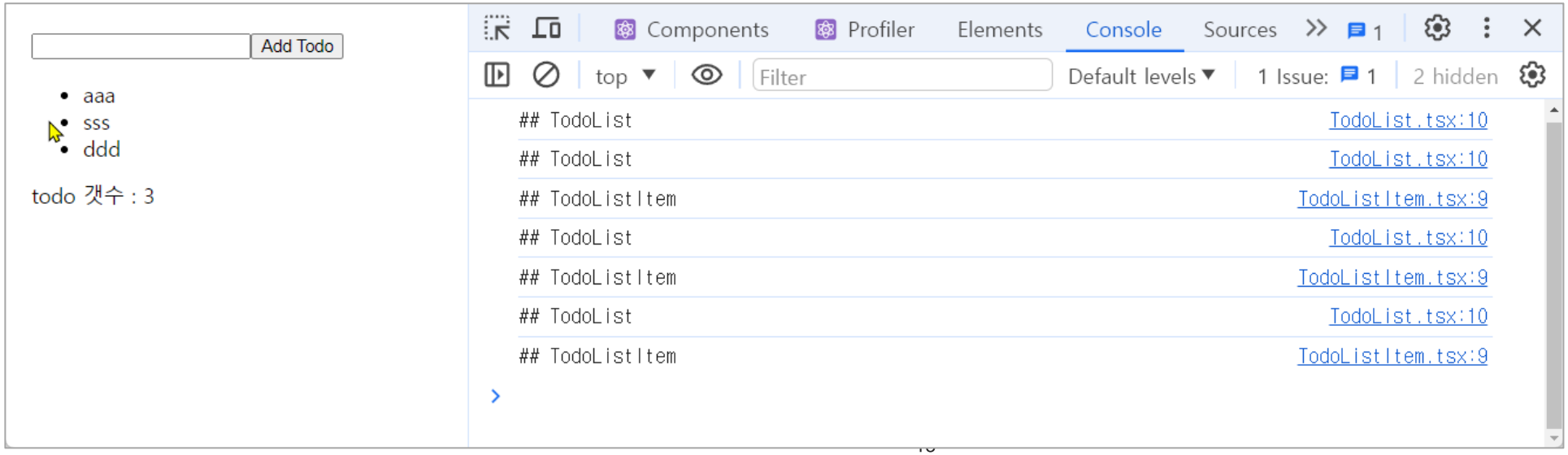
6. 렌더링 최적화 1단계

❖React.memo() 고차함수 적용

```
// TodoList, TodoListItem 컴포넌트를 다음과 같이 React.memo()로 감싸줌
const TodoList = React.memo(({ todoList }: PropsType) => {
    ...(생략)
});
```

❖적용 결과

- 입력필드에 타이핑하는 동안 TodoList, TodoListItem 컴포넌트 re-render 되지 않음
- 새로운 항목을 추가하더라도 모든 TodoListItem 이 아닌 추가된 Item만 render 됨



6. 렌더링 최적화 2단계

❖React.memo() 만으로 최적화가 완료된 걸까?

- 메서드(함수)가 속성으로 전달되는 상황에서는 어떨까?
  - 기존 코드에 할 일 아이템 삭제기능을 추가해 봄 → 어떤 문제점이 발생하는지 살펴보자
- src/App.tsx 변경

```
.....(생략)
const App = () => {
  .....(생략)
  const deleteTodo = (id: number) => {
    const index = todoList.findIndex((item) => item.id === id);
    const newTodoList = produce(todoList, (draft) => {
      draft.splice(index, 1);
    });
    setTodoList(newTodoList);
  };
  return (
    <div className="boxStyle">
      .....(생략)
      <TodoList todoList={todoList} deleteTodo={deleteTodo} />
      <div>todo 갯수 : {todoList.length}</div>
    </div>
  );
};
export default App;
```

6. 렌더링 최적화 2단계

- src/TodoList.tsx 변경

```
import React from "react";
import { TodoListItemType } from "./App";
import TodoListItem from "./TodoListItem";

type PropsType = {
  todoList: TodoListItemType[];
  deleteTodo: (id: number) => void;
};

const TodoList = React.memo(({ todoList, deleteTodo }: PropsType) => {
  console.log("## TodoList");
  return (
    <ul>
      {todoList.map((item) => (
        <TodoListItem key={item.id} todoListItem={item} deleteTodo={deleteTodo} />
      ))}
    </ul>
  );
});

export default TodoList;
```

6. 렌더링 최적화 2단계

- src/TodoListItem.tsx 변경

```
import React from "react";
import { TodoListItemType } from "./App";

type PropsType = {
  todoListItem: TodoListItemType;
  deleteTodo: (id: number) => void;
};

const TodoListItem = React.memo(({ todoListItem, deleteTodo }: PropsType) => {
  console.log("## TodoListItem");
  return (
    <li>
      <span>{todoListItem.todo}</span>   
      <span style={{ cursor: "pointer", color: "blue" }} onClick={() => deleteTodo(todoListItem.id)}>
        삭제
      </span>
    </li>
  );
});

export default TodoListItem;
```

6. 렌더링 최적화 2단계

- 삭제 기능 추가 후 실행 결과
  - 이전과 동일하게 TodoList, TodoListItem이 매번 렌더링됨(필드에 입력을 하는 중에도)
  - 이유는 무엇인가?
    - App 컴포넌트가 Re-render 되면서 함수가 매번 새롭게 생성되었기 때문에
  - 실행 흐름
    - App 컴포넌트에서 할일을 입력필드에 타이핑합니다.
    - App 컴포넌트의 상태가 변경됩니다.
    - App 컴포넌트가 re-render 되면서 deleteTodo 함수가 새롭게 생성됩니다.
    - 새롭게 생성된 deleteTodo 함수가 TodoList를 거쳐 TodoListItem 컴포넌트까지 속성을 통해 전달됩니다.
    - TodoList, TodoListItem 컴포넌트의 기존 deleteTodo 함수와 얕은 비교의 결과가 false 이므로 매번 렌더링됩니다.
  - 어떻게 해결할 것인가?
    - useCallback() 고차 함수

7. 렌더링 최적화 3단계

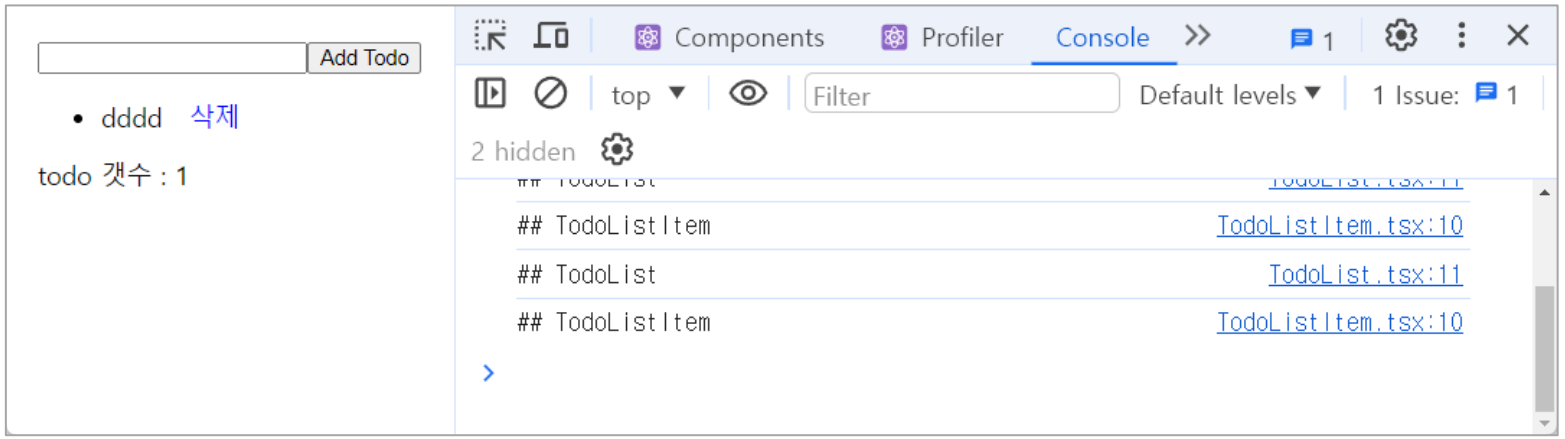
❖ useCallback 훅 적용

- 렌더링할 때마다 함수를 매번 생성하는 것을 막아줌
- 의존 객체를 정확하게 지정해야 함
- src/App.tsx 변경
  - 주의사항 : 반드시 의존값(상태)을 depsList 에 지정해야 함 -〉 지정하지 않으면 클로저 트랩에 빠짐

```
const addTodo = useCallback((todo: string) => {
  const newTodoList = produce(todoList, (draft) => {
    draft.push({ id: new Date().getTime(), todo: todo });
  });
  setTodoList(newTodoList);
  setTodo("");
}, [todoList]);

const deleteTodo = useCallback((id: number) => {
  const index = todoList.findIndex((item) => item.id === id);
  const newTodoList = produce(todoList, (draft) => {
    draft.splice(index, 1);
  });
  setTodoList(newTodoList);
}, [todoList]);
```

7. 렌더링 최적화 3단계

- useCallback 훅 적용 결과
  - 필드에 입력할 때 re-render 되는 현상은 사라졌음
  - 하지만 할일을 추가할 때마다 TodoListItem 모두가 re-render됨
    - 이유는 useCallback의 의존 객체인 todoList가 변경되면 deleteTodo 함수가 다시 만들어지 때문에...
    - useCallback는 메서드가 생성될 때 의존 객체 값을 사용하기 때문에 이 때의 re-render는 꼭 필요한 것임.
    - 이 문제까지 해결하려면 useRef 훅을 응용하는 방법을 사용해야 함
- 만일 useCallback에서 depsList를 빈 배열로 지정하면 어떤 상황이 벌어지는가?
  - addTodo의 useCallback의 depsList를 빈배열로 지정한 후의 실행 결과
    - 여러번 할일을 추가해도 마지막에 추가한 하나의 할일만 남음
    - 클로저 트랩으로 인해서 발생한 문제



8. 렌더링 최적화 4단계

❖useRef + useCallback + useEffect 조합으로 클로저 트랩 우회

- 3단계에서 TodoListItem이 모두 re-render되는 현상은 useCallback 훅의 depsList에 todoList가 지정되었기 때문임
- 이 depsList를 빈 배열로 지정할 수 있도록 한다면 동일한 함수를 매번 속성으로 전달하는 것이므로 문제 해결
- src/App.tsx 변경

```
import { useCallback, useEffect, useRef, useState } from "react";
import TodoList from "./TodoList";
import { produce } from "immer";

export type TodoListItemType = { id: number; todo: string };

const App = () => {
  const [todoList, setTodoList] = useState<TodoListItemType[]>([]);
  const [todo, setTodo] = useState<string>("");

  //함수에 대한 메모리 참조를 가지도록 useRef()훅 이용
  //App 컴포넌트가 마운트될 때는 undefined 로 시작함
  const addTodoRef = useRef<(todo: string) => void | undefined>();
  const deleteTodoRef = useRef<(id: number) => void | undefined>();
```

8. 렌더링 최적화 4단계

- src/App.tsx 변경 (이어서1)

```
// 컴포넌트의 상태가 변경될 때마다 매번 effectCallback이 실행되도록  depsList 지정하지 않음
// ---> 상태가 바뀌면 useRef 참조에 변경된 상태를 참조하는 새로운 함수가 매번 생성되어 할당됨.
useEffect(() => {
  addTodoRef.current = (todo: string) => {
    const newTodoList = produce(todoList, (draft) => {
      draft.push({ id: new Date().getTime(), todo: todo });
    });
    setTodoList(newTodoList);
    setTodo("");
  };

  deleteTodoRef.current = (id: number) => {
    const index = todoList.findIndex((item) => item.id === id);
    const newTodoList = produce(todoList, (draft) => {
      draft.splice(index, 1);
    });
    setTodoList(newTodoList);
  };
});
```

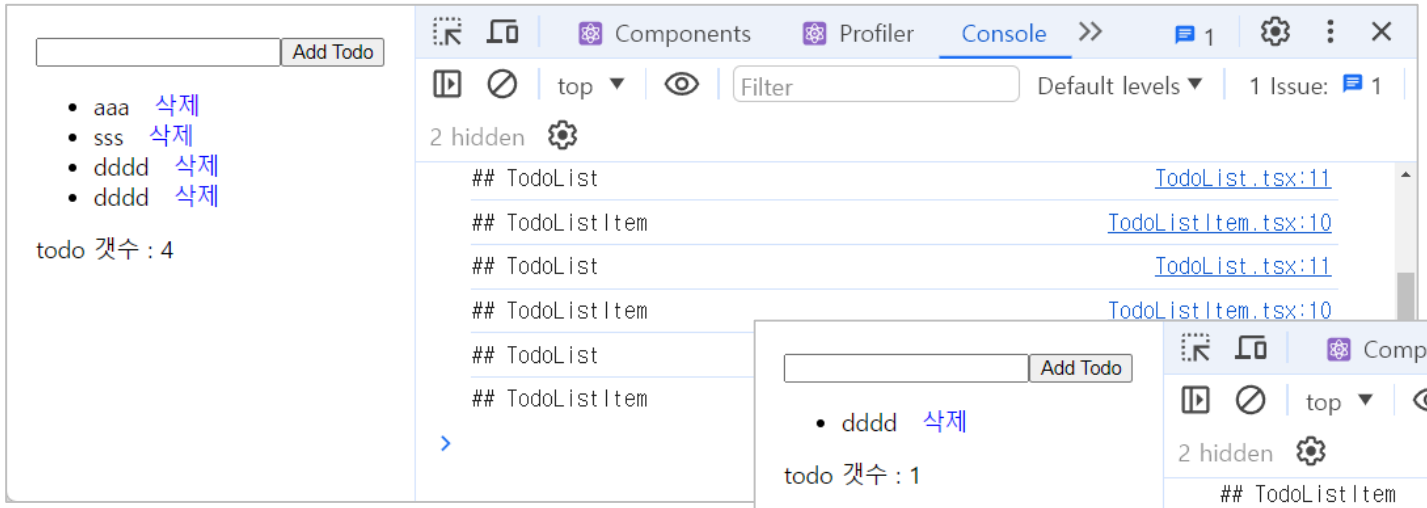
8. 렌더링 최적화 4단계

- src/App.tsx 변경 (이어서2)

```
  //매번 함수를 생성하지 않도록 useCallback()을 이용해 함수 캐싱
  //직접적으로 이용하는 상태, 속성이 없으므로 depsList에 지정하지 않아도 됨
  //즉 속성으로 전달하는 함수는 항상 동일한 함수이므로 re-render를 일으키지 않음
  const addTodo = useCallback((todo: string) => {
    addTodoRef.current && addTodoRef.current(todo);
  }, []);

  const deleteTodo = useCallback((id: number) => {
    deleteTodoRef.current && deleteTodoRef.current(id);
  }, []);

  return (
      .....(생략)
  );
};

export default App;
```

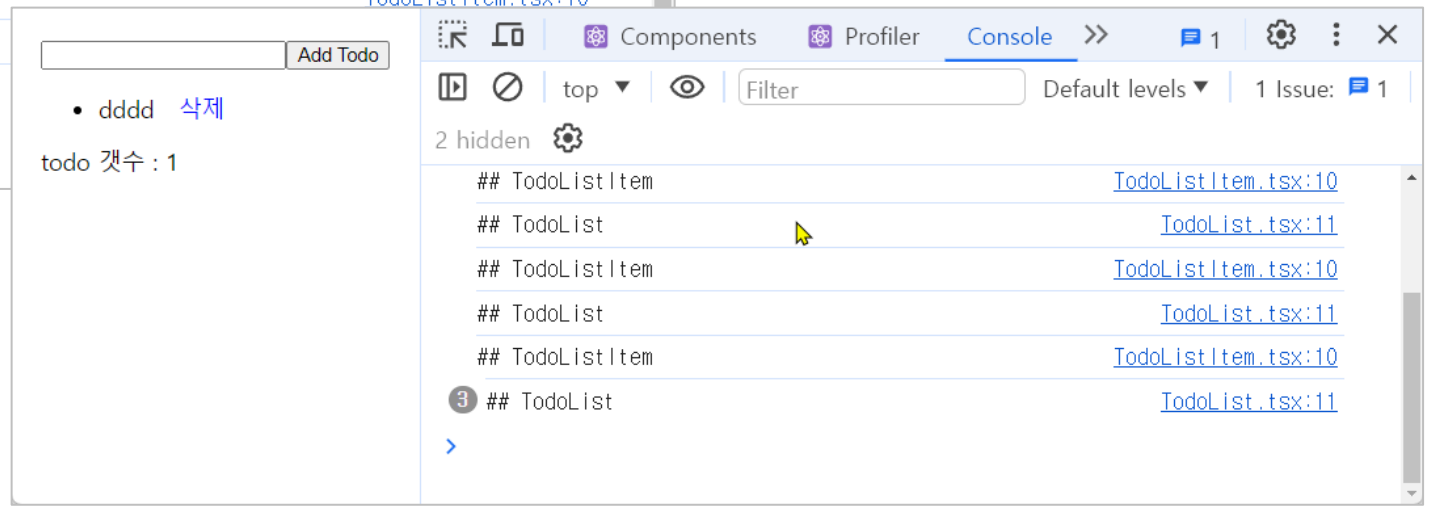
8. 렌더링 최적화 4단계

- useRef + useCallback + useEffect 조합 실행 결과
  - 할일을 추가할 때는 추가한 TodoListItem만 렌더링됨
  - 삭제할 때는 모든 TodoListItem 컴포넌트는 렌더링되지 않음
- 하지만 이 방법은 조금 번거롭고 까다로움



<추가할 때>

<삭제할 때>



9. 렌더링 최적화 5단계

❖컴포넌트 분할을 이용한 최적화

- 리액트 앱에서 React.memo() 로 얕은 비교후 렌더링할지 말지를 결정하는 단위는 **컴포넌트**임
  - 입자가 큰 컴포넌트의 상태, 속성이 변경되면 컴포넌트 전체를 re-render함.
- 따라서 적절히 컴포넌트를 분할하는 것은 성능 개선에 큰 도움을 줌

❖예제 준비

- render-perf-optimize-3 예제를 이용해서 시작
  - 3단계 적용 예제 : useCallback() 까지만 적용된 예제
- 최적화할만한 부분
  - 할일을 입력하는 입력필드 UI 영역을 별도의 컴포넌트로 분리
  - TodoListItem을 매번 바뀌는 속성을 전달받는 부분과 그렇지 않은 부분으로 분리

9. 렌더링 최적화 5단계

- src/AddTodo.tsx 작성

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

type PropsType = {
  addTodo: (todo: string) => void;
};

const AddTodo = ({ addTodo }: PropsType) => {
  const [todo, setTodo] = useState<string>("");

  const addTodoHandler = () => {
    addTodo(todo);
    setTodo("");
  };

  return (
    <div>
      <input type="text" value={todo} onChange={(e) => setTodo(e.target.value)} />
      <button onClick={addTodoHandler}>Add Todo</button>
    </div>
  );
};

export default AddTodo;
```

9. 렌더링 최적화 5단계

- src/App.tsx 변경
  - todo 상태를 AddTodo 컴포넌트로 이동

```
import { useCallback, useState } from "react";
import TodoList from "./TodoList";
import { produce } from "immer";
import AddTodo from "./AddTodo";

export type TodoListItemType = { id: number; todo: string };

const App = () => {
  const [todoList, setTodoList] = useState<TodoListItemType[]>([]);
  ......(생략)

  return (
    <div className="boxStyle">
      <AddTodo addTodo={addTodo} />
      <br />
      <TodoList todoList={todoList} deleteTodo={deleteTodo} />
      <div>todo 갯수 : {todoList.length}</div>
    </div>
  );
};

export default App;
```

9. 렌더링 최적화 5단계

- src/TodoListItemBody.tsx 작성

```
import React from "react";
import { TodoListItemType } from "./App";

type PropsType = {
  todoListItem: TodoListItemType;
};

const TodoListItemBody = React.memo(({ todoListItem }: PropsType) => {
  console.log("## TodoListItemBody");
  return <span>{todoListItem.todo}</span>;
});

export default TodoListItemBody;
```

9. 렌더링 최적화 5단계

- src/TodoListItemDelete.tsx 작성

```
import React from "react";

type PropsType = {
  id: number;
  deleteTodo: (id: number) => void;
};

const TodoListItemDelete = React.memo(({ id, deleteTodo }: PropsType) => {
  console.log("## TodoListItemDelete");
  return (
    <span style={{ cursor: "pointer", color: "blue" }} onClick={() => deleteTodo(id)}>
      삭제
    </span>
  );
});

export default TodoListItemDelete;
```

9. 렌더링 최적화 5단계

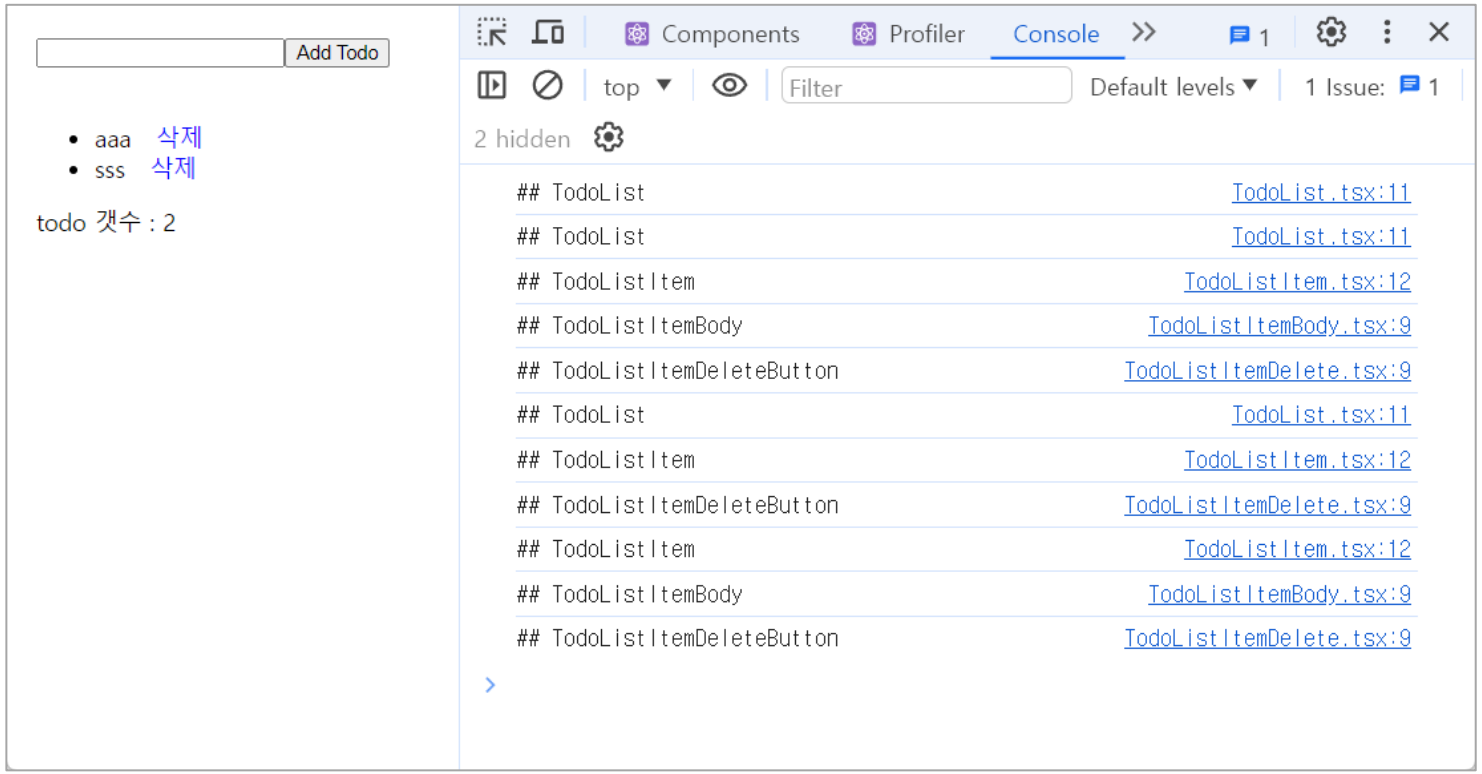
- src/TodoListItem.tsx 변경

```
import React from "react";
import { TodoListItemType } from "./App";
import TodoListItemBody from "./TodoListItemBody";
import TodoListItemDelete from "./TodoListItemDelete";

type PropsType = {
  todoListItem: TodoListItemType;
  deleteTodo: (id: number) => void;
};

const TodoListItem = React.memo(({ todoListItem, deleteTodo }: PropsType) => {
  console.log("## TodoListItem");
  return (
    <li>
      <TodoListItemBody todoListItem={todoListItem} />
         
      <TodoListItemDelete deleteTodo={deleteTodo} id={todoListItem.id} />
    </li>
  );
});

export default TodoListItem;
```

9. 렌더링 최적화 5단계

❖실행 결과

- 새롭게 추가한 아이템은 Body, DeleteButton 모두 렌더링
- 나머지 아이템들은 새로운 todoList를 참조하는 deleteTodo 속성(함수)를 전달받는 DeleteButton만 렌더링



10. 정리

❖렌더링 최적화를 모든 컴포넌트에 적용해야 하는가?

- 그렇지 않음. 반드시 필요한 컴포넌트에만 적용할 것
  - 렌더링 최적화를 필요로 하는 컴포넌트는 의외로 적음
  - 렌더링 최적화를 위해 캐싱해야 하므로 메모리를 더 사용할 수도 있음
  - 렌더링 최적화를 위해 개발자의 공수가 추가로 소요됨
- 렌더링 최적화가 바람직한 경우
  - 정적인 데이터를 반복적으로 렌더링하는 컴포넌트
  - 동일한 속성(props)를 이용해 반복적으로 렌더링하는 컴포넌트