11. 컴퍼넌트 최적화 하기

11.1 대량의 데이터 랜더링

- 대량의 데이터 생성후 성능 테스트하기

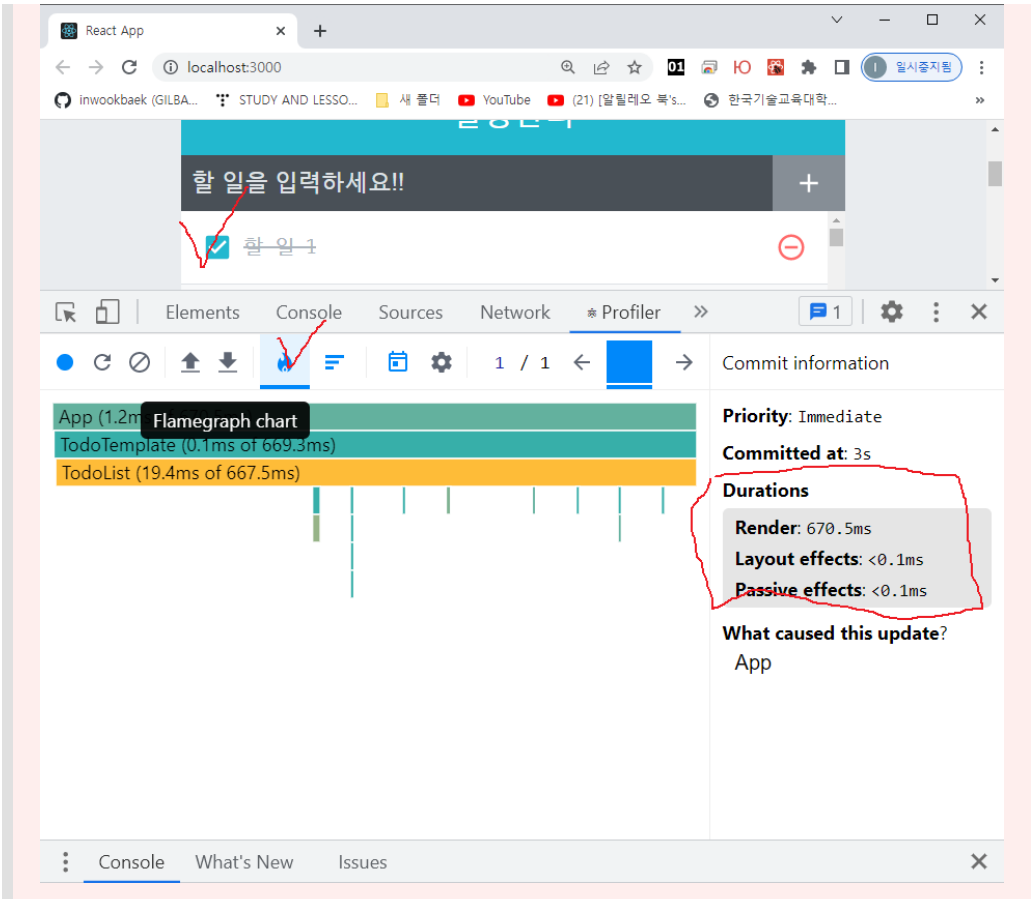
App.js

```
import { useCallback, useRef, useState } from "react";
import TodoTemplate from "./components/TodoTemplate";
import TodoInsert from "./components/TodoInsert";
import TodoList from "./components/TodoList";

function createBulkTodos() {
  const array = [];
  for (let i = 1; i <= 2500; i++) {
    array.push({
      id: i,
      text: `할 일 ${i}`,
      checked: false,
    });
  }
  return array;
}

const App = () => {

  const [todos, setTodos] = useState(createBulkTodos);

  const nextId = useRef(2501);

  const onInsert = useCallback(
    text => {
      const todo = {
        id: nextId.current,
        text,
        checked: false
      };
      setTodos(todos.concat(todo));
      nextId.current += 1;
    }, [todos]
  )

  const onRemove = useCallback(
    id => {
      setTodos(todos.filter(todo => todo.id !== id));
    }, [todos]
  )

  const onToggle = useCallback(
    id => {
      setTodos(todos.map(todo => todo.id === id ? { ...todo, checked:
!todo.checked} : todo));
```

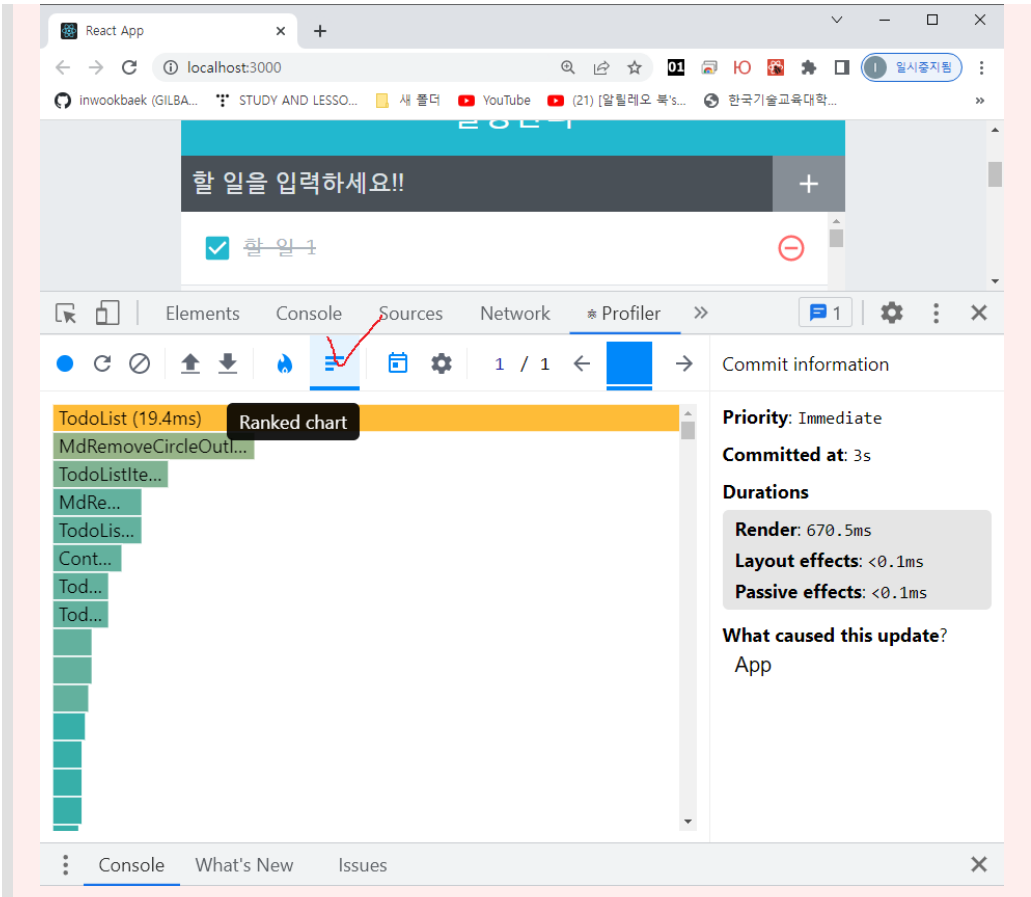
```
    }, [todos]
  )

  return (
    <TodoTemplate>
      <TodoInsert onInsert={onInsert} />
      <TodoList todos={todos} onRemove={ onRemove } onToggle={ onToggle } />
    </TodoTemplate>
  );
};

export default App;
```

11.2 크롬 개발자도구로 성능 모니터링

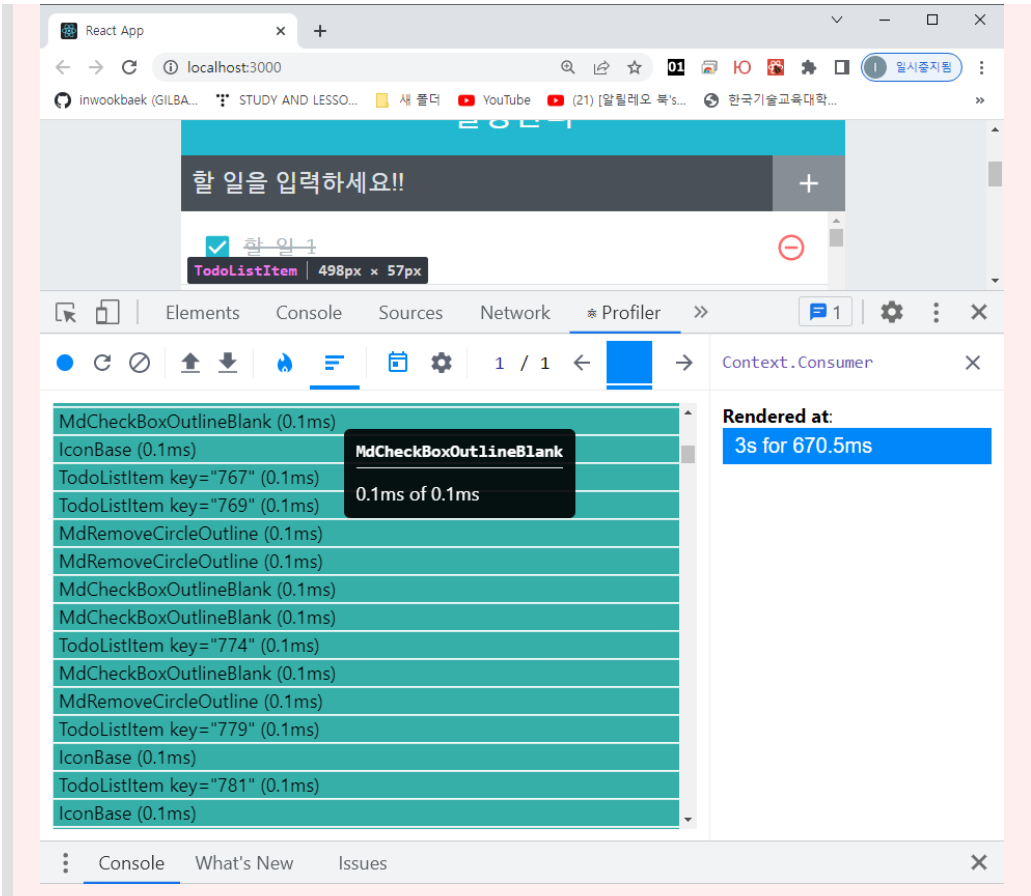
1. 프로그램시작 : `yarn start`
2. 500개의 데이터 자동생성 확인
3. 크롬 개발자도구에서 `Profiler` 텝에서 좌측 상단에 파란색 녹화버튼 클릭
4. 1항목 체크후 녹화버튼 다시 클릭(중지)



- Render Duration : 리렌더링에 소요된 시간
- 불꽃모양 아이콘 우측의 `랭크차트` 아이콘 클릭



- 상기화면은 리렌더링된 컴퍼넌트를 오래 걸린 순으로 정렬
- 작은 초록색 박스를 클릭하면 크기가 늘어나 내용을 확인할 수 있다.



- 내용을 보면 관계없는 컴퍼넌트들도 리렌더링된 것을 확인할 수 있다.

- 한 개의 데이터만 업데이트함에도 많은 소요시간이 걸린 것을 알 수 있는데 이를 최적화하는 방법을 알아보자

11.3 느려지는 원인 분석

- 컴퍼넌트는 아래와 같은 상황에서 리랜더링을 한다.

  1. 전달받은 props가 변경될 때
  2. state가 변경될 때
  3. 부모 컴퍼넌트가 리랜더링될 때
  4. forceUpdate함수가 실행될 때

- 첫 번째 항목을 수정할 경우 App컴퍼넌트의 state가 변경되면서 리렌더링이 되면서
- 자식 컴퍼넌트 즉, TodoList가 리랜더링되고 그 안의 컴퍼넌트도 리랜더링이 되면서 느려지게 된다.
- 즉, 2~2500번째 데이터도 모두 리랜더링이 되기 때문에 느려지는 원인이 된다.

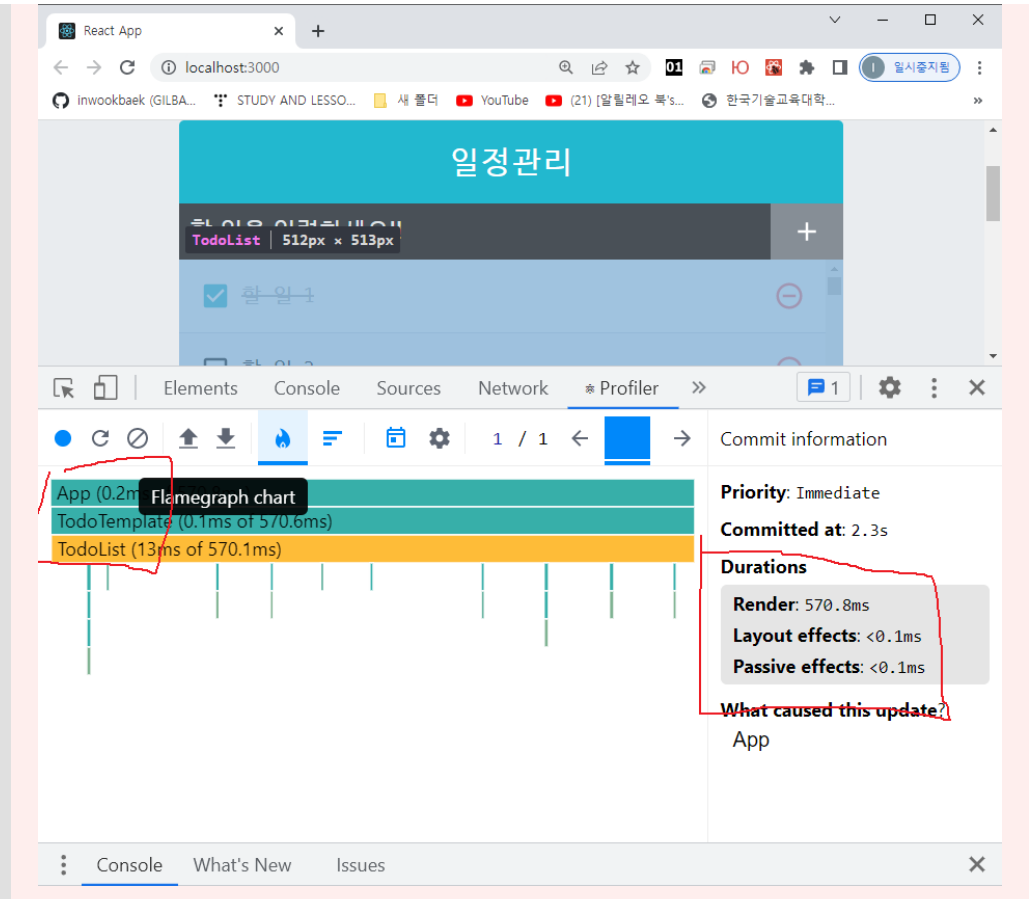
1.4 React.memo를 사용하여 최적화하기

- 컴퍼넌트의 리랜더링을 방지할 때 shouldComponentUpdate라는 함수를 사용하면 되는데
- 이 함수는 클래스형 컴퍼넌트의 함수로 `함수형 컴퍼넌트에서는 사용할 수 없다.
- 그 대신에 `React.memo 함수를 사용` 한다.
- React.memo의 사용법은 매우 간단하다. 컴퍼넌트를 만들고 나서 감싸주기만 하면 된다.

.js

```
import React from 'react';
import {
  MdCheckBoxOutlineBlank,
  MdCheckBox,
  MdRemoveCircleOutline,
} from 'react-icons/md';
import cn from 'classnames'
import './TodoListItem.scss';

const TodoListItem = ({ todo, onRemove, onToggle }) => {

  const {id, text, checked} = todo;

  return (
    <div className="TodoListItem">
      <div className={cn('checkbox', {checked})} onClick={() => onToggle(id)}>
        {checked ? <MdCheckBox />  : <MdCheckBoxOutlineBlank /> }
        <div className="text">{ text }</div>
      </div>
      <div className="remove" onClick={() => onRemove(id)}>
        <MdRemoveCircleOutline />
      </div>
    </div>
  );
```

```
};
//=========================================================
export default React.memo(TodoListItem);
//=========================================================
```



11.5 onToggle, onRemove함수 변경 방지

- 현재 프로젝트에서는 todos가 변경되면 onToggle, onRemove도 변경이 된다.
- onToggle함수는 최신 상태의 todos를 참조하기 때문에 배열상태가 업데이트되면서 함수가 새로 만들어 진다.
- 함수가 새롭게 만들어지는 것을 방지하는 방법은
  1. useState의 함수형 업데이트 기능을 사용하는 것과
  2. useReducer를 사용하는 것이다.

11.5.1 useState의 함수형 업데이트

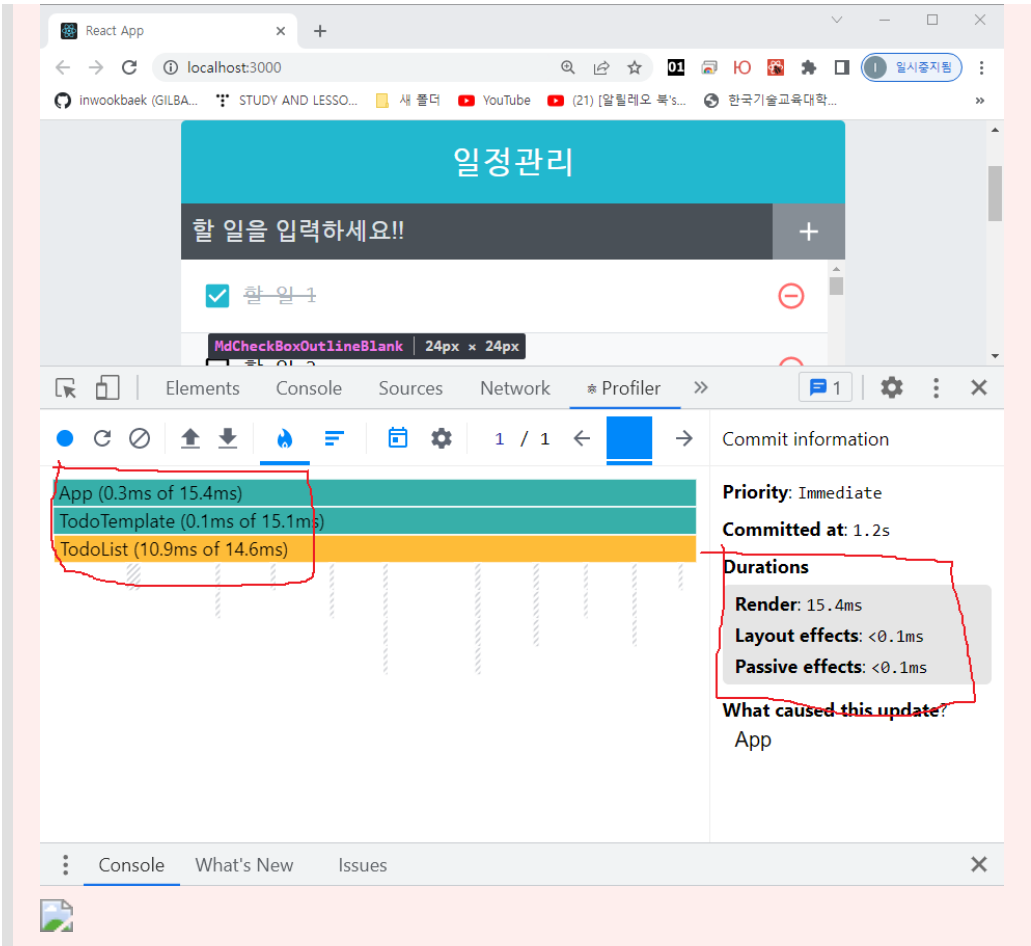
- 기존에 setTodos함수를 사용할 때는 새로운 상태를 파라미터로 전달 했다.
- 대신, 상태 업데이트를 어떻게 할지를 정의해 주는 함수를 전달할 수 있다. 이를 함수형업데이트라고 한다.
- setTodos를 사용할 때 'totos =>' 만 넣어 준다 ##### App.js

```
import { useCallback, useRef, useState } from "react";
import TodoTemplate from "./components/TodoTemplate";
import TodoInsert from "./components/TodoInsert";
import TodoList from "./components/TodoList";

function createBulkTodos() {
  const array = [];
```

```
  for (let i = 1; i <= 2500; i++) {
    array.push({
      id: i,
      text: `할 일 ${i}`,
      checked: false,
    });
  }
  return array;
}

const App = () => {

  const [todos, setTodos] = useState(createBulkTodos);

  const nextId = useRef(2501);

  const onInsert = useCallback(
    text => {
      const todo = {
        id: nextId.current,
        text,
        checked: false
      };
      // setTodos(todos.concat(todo));
      // 함수형업데이트
      setTodos(todos => todos.concat(todo));
      nextId.current += 1;
    }, []
  )

  const onRemove = useCallback(
    id => {
      // 함수형업데이트
      setTodos(todos => todos.filter(todo => todo.id !== id));
    }, []
  )

  const onToggle = useCallback(
    id => {
      // 함수형업데이트
      setTodos(todos=> todos.map(todo => todo.id === id ? { ...todo, checked:
!todo.checked} : todo));
    }, []
  )

  return (
    <TodoTemplate>
      <TodoInsert onInsert={onInsert} />
      <TodoList todos={todos} onRemove={ onRemove } onToggle={ onToggle } />
    </TodoTemplate>
  );
};

export default App;
```

수정후 성능테스트

- 소요시간과 리랜더링된 컴퍼넌트의 수를 확인해 보면 몇개 없는 것을 화인할 수 있다.
- 흰색 빗금박스들은 React.memo를 통해 리랜더링이 되지 않은 컴퍼넌트를 나타낸다.



11.5.2 useReducer 사용하기

- 함수형업데이트대신에 useReducer를 사용해도 동일한 문제를 해결할 수 있다.
- useReducer를 사용할 때 원래 두 번째에는 초기상태를 넣어 주어야 하는데 대신에 undefined를 넣었고
- 세 번째 파라미터에 초기상태를 만들어 주는 createBulkTodos를 넣어 주었다. 이럴 경우 처음 랜더링할 때문 함수가 호출된다.

App.js

```
import { useCallback, useReducer, useRef } from "react";
import TodoTemplate from "./components/TodoTemplate";
import TodoInsert from "./components/TodoInsert";
import TodoList from "./components/TodoList";

function createBulkTodos() {
  const array = [];
  for (let i = 1; i <= 2500; i++) {
    array.push({
      id: i,
      text: `할 일 ${i}`,
      checked: false,
    });
  }
  return array;
```

```
}

function todoReducer(todos, action) {

  switch(action.type) {
    case 'INSERT':
      // {type: 'INSERT', todo: {id:1, text: 'todo...', checked:false}}
      return todos.concat(action.todo);
    case 'REMOVE':
      // {type: 'REMOVE', todo: {id:1}}
      return todos.filter(todo => todo.id !== action.id);
    case 'TOGGLE':
      // {type: 'TOGGLE', todo: {id:1}}
      return todos.map(todo => todo.id === action.id ? { ...todo, checked:
!todo.checked} : todo)
    default:
      return todos;
  }
}

const App = () => {

  const [todos, dispatch] = useReducer(todoReducer, undefined,
createBulkTodos);

  const nextId = useRef(2501);

  const onInsert = useCallback(
    text => {
      const todo = {
        id: nextId.current,
        text,
        checked: false
      };
      // setTodos(todos.concat(todo));
      // 함수형업데이트
      // setTodos(todos => todos.concat(todo));
      // useReducer
      dispatch({type:'INSERT', todo})
      nextId.current += 1;
    }, []
  )

  const onRemove = useCallback(
    id => {
      // 함수형업데이트
      // setTodos(todos => todos.filter(todo => todo.id !== id));
      // useReducer
      dispatch({type: 'REMOVE', id})
    }, []
  )

  const onToggle = useCallback(
    id => {
      // 함수형업데이트
      // setTodos(todos=> todos.map(todo => todo.id === id ? { ...todo,
checked: !todo.checked} : todo));
```

```
      // useReducer
      dispatch({type: 'TOGGLE', id});
    }, []
  )

  return (
    <TodoTemplate>
      <TodoInsert onInsert={onInsert} />
      <TodoList todos={todos} onRemove={ onRemove } onToggle={ onToggle } />
    </TodoTemplate>
  );
};

export default App;
```

11.6 불변성의 중요성

- 기존의 값을 직접 수정하지 않으면서 `새로운 값을 만들어 내는 것을 불변성을 지킨다` 라고 한다.
- 불변성이 지켜지지 않으면 객체 내부의 값이 새로워져도 바뀐 것을 감지하지 못한다.
- 그러면 React.memo에서 서로 비교하여 최적화하는 것이 불가능하게 된다.
- `전개 연산자를 사용하여 복사하면 얕은 복사`가 된다.
- 배열, 객체구조가 복잡해 지면 불변성을 유지하기가 어렵게 된다. `불변성을 도와주는 라이브러리가 immer`이다.

11.7 TodoList 최적화하기

TodoList.js

```
import TodoListItem from './TodoListItem';
import './TodoList.scss';

const TodoList = ({ todos, onRemove, onToggle }) => {
  return (
    <div className="TodoList">
      {todos.map(todo => (
        <TodoListItem todo={todo} key={todo.id} onRemove={onRemove} onToggle=
{onToggle}/>
      ))}
    </div>
  );
};
/=============================================
export default React.memo(TodoList);
/=============================================
```

- React.momo로 현재 프로젝트 성능에 전형 영향을 주지 않는다.
- 부모인 APp가 리렌더링되는 유일한 이유가 todos배열이 업데이트 될 때 이다.
- 즉, TodoList컴퍼넌트는 불필요한 리랜더링이 발생하지 않는다.
- 하지만 App에 다른 state가 추가 되어 해당 값이 업데이트될 때 TodoList가 불필요한 리랜더링이 된다.

- 그렇기 때문에 사저에 React.memo로 최적화해 준 것이다.

11.8 react-virtualized를 사용한 리랜더링

- react-virtualized를 사용하면 스크롤되기 전에 보이지 않는 컴퍼넌트는 랜더링하지 않고 크기만 차지하게끔 할 수 있다.
- 만약 스크롤하게 되면 해당 위치에 해당되는 컴퍼넌트만 조회하게 해 준다.

11.8.1 최적화준비

- 패키지설치 : `yarn add react-virtualized`
- 최적화하기 전에 각 항목의 실제 크기의 px단위를 알아 내는 것이다. (크롬 개발자 도구를 이용)

11.8.2 TodoList 수정

TodoList.js

```
import React, { useCallback } from 'react';
import TodoListItem from './TodoListItem';
import './TodoList.scss';
import { List } from 'react-virtualized';

const TodoList = ({ todos, onRemove, onToggle }) => {

  const rowRenderer = useCallback(
    ({ index, key, style }) => {
      const todo = todos[index];
      return (
        <TodoListItem
          todo = {todo}
          key = {key}
          onRemove = {onRemove}
          onToggle = { onToggle }
          style = {style}
        />
      )
    }, [onRemove, onToggle, todos]
  );

  return (
    <List
      className="TodoList"
      width={512} // 전체 크기
      height={350} // 전체 높이
      rowCount={todos.length} // 항목 개수
      rowHeight={57} // 항목 높이
      rowRenderer={rowRenderer} // 항목을 렌더링할 때 쓰는 함수
      list={todos} // 배열
      style={{ outline: 'none' }} // List에 기본 적용되는 outline 스타일 제거
    />
  );
};
```

```
export default React.memo(TodoList);
```

11.8.3 TodoListItem.js 수정

TodoListItem .js

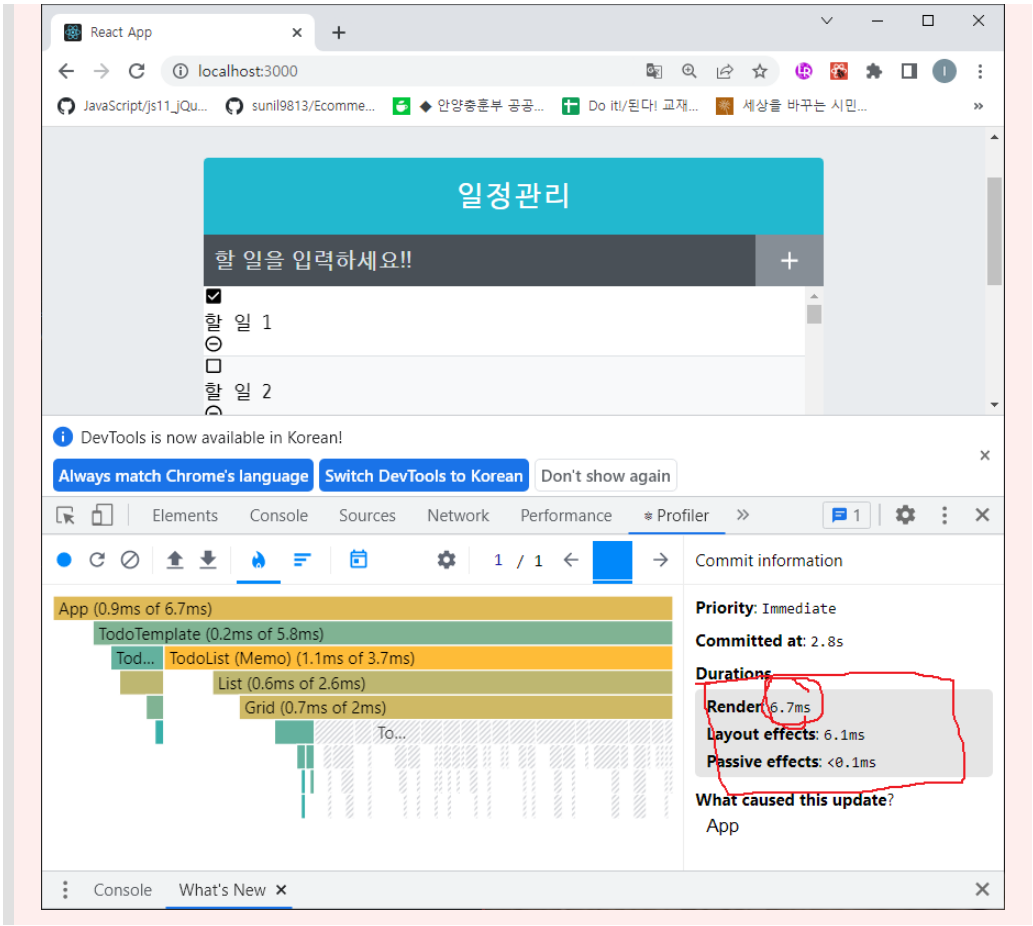
```
import React from 'react';
import {
  MdCheckBoxOutlineBlank,
  MdCheckBox,
  MdRemoveCircleOutline,
} from 'react-icons/md';
import cn from 'classnames'
import './TodoListItem.scss';

const TodoListItem = ({ todo, onRemove, onToggle, style }) => {

  const {id, text, checked} = todo;

  return (
    <div className="TodoListItem-virtualized" style={style}>
      <div className={cn('checkbox', {checked})} onClick={() => onToggle(id)}>
        {checked ? <MdCheckBox />  : <MdCheckBoxOutlineBlank /> }
        <div className="text">{ text }</div>
      </div>
      <div className="remove" onClick={() => onRemove(id)}>
        <MdRemoveCircleOutline />
      </div>
    </div>
  );
};

export default React.memo(TodoListItem);
```

TodoListItem.scss

```
.TodoListItem-virtualized {
  & + & {
    border-top: 1px solid #dee2e6;
  }
  &:nth-child(even) {
    background: #f8f9fa;
  }
}

.TodoListItem {
  padding: 1rem;
  display: flex;
  align-items: center; // 세로 중앙 정렬
  &:nth-child(even) {
    background: #f8f9fa;
  }
  .checkbox {
    cursor: pointer;
    flex: 1; // 차지할 수 있는 영역 모두 차지
    display: flex;
    align-items: center; // 세로 중앙 정렬
```

```
      svg {
        // 아이콘
        font-size: 1.5rem;
      }
      .text {
        margin-left: 0.5rem;
        flex: 1; // 차지할 수 있는 영역 모두 차지
      }
      // 체크되었을 때 보여줄 스타일
      &.checked {
        svg {
          color: #22b8cf;
        }
        .text {
          color: #adb5bd;
          text-decoration: line-through;
        }
      }
    }
    .remove {
      display: flex;
      align-items: center;
      font-size: 1.5rem;
      color: #ff6b6b;
      cursor: pointer;
      &:hover {
        color: #ff8787;
      }
    }

    // 엘리먼트 사이사이에 테두리를 넣어줌
    & + & {
      border-top: 1px solid #dee2e6;
    }
  }
```

- scss적용시 ui가 깨진다.(해결안하고 넘어감)
- 성능이 많이 좋아 졌다.


React App
localhost:3000
일정관리
할 일을 입력하세요!!
할 일 1
할 일 2
DevTools is now available in Korean!
Always match Chrome's language
Switch DevTools to Korean
Don't show again
Elements
Console
Sources
Network
Performance
Profiler
Commit information
App (0.9ms of 6.7ms)
TodoTemplate (0.2ms of 5.8ms)
TodoList (Memo) (1.1ms of 3.7ms)
List (0.6ms of 2.6ms)
Grid (0.7ms of 2ms)
Priority: Immediate
Committed at: 2.8s
Durations
Render 6.7ms
Layout effects: 6.1ms
Passive effects: <0.1ms
What caused this update?
App
Console
What's New