React State

개요

State 변수는 일반 JavaScript 변수처럼 보인다. 하지만 state는 스냅샷처럼 동작한다. state 변수를 설정하여도 이미 갖고 있는 state 변수는 변경되지 않고 리렌더링이 발동된다.

rendering

렌더링이란 React가 컴포넌트를 호출한다는 뜻이다. 해당 함수에서 반환하는 JSX는 시간에 따른 UI snapshot과 같다. prop, event handler, local variation은 모두 렌더링 당시의 state를 사용해 계산된다. UI snapshot은 대화형이다. 여기에 입력에 대한 응답으로 어떤 일이 일어날지를 결정하는 핸들러와 같은 로직이 포함된다. React는 스냅샷과 일치하도록 화면을 업데이트하고 이벤트 핸들러를 연결한다. 결과적으로 버튼을 누르면 JSX의 핸들러가 동작한다.

React가 컴포넌트를 다시 렌더링할 때

- React가 컴포넌트를 다시 호출한다.
- 컴포넌트가 새로운 JSX 스냅샷을 반환한다.
- React가 컴포넌트가 반환한 스냅샷과 일치하도록 화면을 업데이트 한다.

컴포넌트의 메모리로써 state는 함수가 반환된 후 사라지는 일반 변수와 다르다. state는 React 자체에 메모리된다. React가 컴포넌트를 호출하면 특정 렌더링에 대한 state의 스냅샷을 제공한다. 컴포넌트는 해당 렌더링의 state 값을 사용해 계산된 새로운 props와 event handler가 포함된 UI의 스냅샷을 JSX에 반환한다.

다음 예제를 보면 `onClick`으로 `setNumber`함수를 세 번 호출하도록 전달하고 있다.

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 1);
        setNumber(number + 1);
        setNumber(number + 1);
      }}>+3</button>
    </>
  )
}
```

한 번에 클릭으로 +3만큼 `number`가 증가하길 기대하고 작성한 코드로 보이지만 실제로는 +1만큼 증가한다. **그 이유는 state를 설정하면 다음 렌더링에 대해서만 변경되기 때문이다.** 순차적으로 보면 다음과 같다.

- 사용자가 버튼을 클릭하여 `onClick`에 전달된 `setNumber(number + 1)`이 호출됐다.

- React는 다음 렌더링에서 `number`를 +1만큼 증가시킬 준비를 한다. (누르는 즉시 `number`가 +1만큼 증가하여 다음 코드에 바로 적용되는 것이 아니다.)

결론적으로 `<button>` 안에 코드를 풀어서 보면 다음과 같이 생겼다고 볼 수 있다. 각 렌더링의 state 값은 고정되어 있다.

```
<button onClick={() => {
  setNumber(0 + 1);
  setNumber(0 + 1);
  setNumber(0 + 1);
}}>+3</button>
```

타이머를 설정하여 컴포넌트가 다시 렌더링 된 후에 동작하도록 만들어도 결과는 마찬가지다.

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
        setTimeout(() => {
          alert(number);
        }, 3000);
      }}>+5</button>
    </>
  )
}
```

그 이유는 React에 저장된 state는 `alert`가 호출될 때 변경된 상태일 수도 있지만, 사용자가 상호작용한 시점에 state snapshot을 사용하기 때문에 이미 예약된 state로 호출된다.

```
setNumber(0 + 5);
setTimeout(() => {
  alert(0);
}, 3000);
```

React State batches

state 변수를 설정하면 다음 렌더링 큐에 들어간다. 그러나 때에 따라 다음 렌더링 큐에 넣기 전에 값에 대해 여러 작업을 수행하고 싶을 때도 있다.

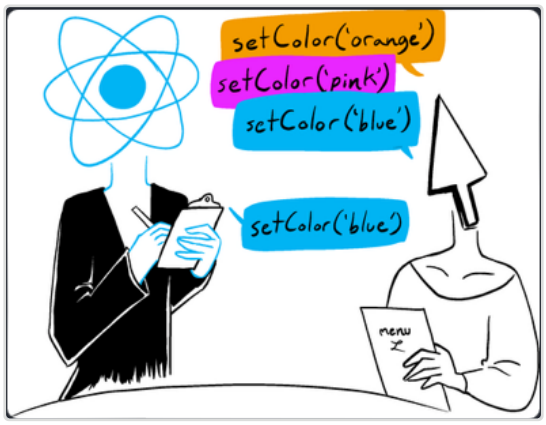
```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 1);
        setNumber(number + 1);
```

```
      setNumber(number + 1);
    }}>+3</button>
  </>
 )
}
```

위 예제를 다시 보면 각 렌더링의 state 값은 고정되어 있기 때문에 `setNumber(number + 1)`을 여러 번 호출 해도 `setNumber(0 + 1)`과 같다.

여기에 한 가지 더 이야기 하자면, **React는 state 업데이트를 하기 전에 Event handler의 모든 코드가 실행될 때까지 기다린다는 점**이다. 이렇게 동작하면 너무 많은 리렌더링을 방지할 수 있다.



반대로 각각의 `setNumber()`에 값을 전달하여 예제에서 예상한대로 3만큼 증가하도록 하는 것을 batch라고 한다.

다음 렌더링 전에 동일한 state 변수 업데이트

다음 렌더링 전에 동일한 state 변수를 업데이트 하고 싶다면 `setNumber(number + 1)`을 `setNumber(n => n + 1)`와 같이 Queue의 state를 기반으로 다음 state를 계산하는 함수를 전달할 수 있다. 이는 단순히 state 값을 대체하는 것이 아니라 React에 state 값으로 무언가를 하도록 지시하는 방법이다. `n => n + 1`은 업데이터 함수라고 부른다.

```
export default function Counter() {
 const [number, setNumber] = useState(0);

 return (
  <>
   <h1>{number}</h1>
   <button onClick={() => {
    setNumber(n => n + 1);
    setNumber(n => n + 1);
    setNumber(n => n + 1);
   }}>+3</button>
  </>
 )
}
```

React가 이벤트 핸들러를 수행하는 동안 여러 코드를 통해 작동하는 방식은 다음과 같다.

1. `setNumber(n => n + 1)` : `n => n + 1` 함수를 Queue에 추가한다. (세 번)
2. 다음 렌더링 중에 `useState`를 호출하면 React는 Queue를 순회한다. 이전 `number` state는 0이었기 때문에 React는 첫 번째 업데이터 함수에 `n` 인수로 0을 전달한다.
3. React는 이전 업데이터 함수의 반환 값을 가져와서 다음 업데이터 함수에 전달하는 식으로 반복한다.

queued update	n	returns
`n => n + 1`	`0`	`0 + 1 = 1`
`n => n + 1`	`1`	`1 + 1 = 2`
`n => n + 1`	`2`	`2 + 1 = 3`

state를 교체한 후 업데이트

만약 `setNumber(n + 5)`와 같은 함수를 호출한 다음 업데이터 함수를 호출하면 어떻게 될까?

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
        setNumber(n => n + 1);
      }}>Increase the number</button>
    </>
  )
}
```

React가 이벤트 핸들러에 작업을 지시한다.

1. `setNumber(number + 5)` : `number`가 0이므로 `setNumber(0 + 5)`이다. React는 Queue에 'number 5만큼 증가' 를 추가한다.
2. `setNumber(n => n + 1)` : `n => n + 1`은 업데이터 함수이다. React는 해당 함수를 Queue에 추가한다.
3. 다음 렌더링을 하는 동안 React는 state queue를 순회한다.

queued update	n	returns
"replace with `5`"	`0` (unused)	`5`
`n => n + 1`	`5`	`5 + 1 = 6`

최종적으로 React는 6을 저장하고 `useState`에서 반환한다.

업데이트 후 state를 변경

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
        setNumber(n => n + 1);
```

```
      setNumber(number + 42);
    }}>Increase the number</button>
  </>
 )
}
```

1. `setNumber(number + 5)` : `number`는 0이므로 `setNumber(0 + 5)`이다. React는 'number 값 5만큼 증가'를 Queue에 추가한다.
2. `setNumber(n => n + 1)` : React가 `n => n + 1` 업데이터 함수를 Queue에 추가한다.
3. `setNumber(number + 42)` : React가 'number 값 42만큼 증가'을 Queue에 추가한다.

queued update	n	returns
"replace with `5`"	`0` (unused)	`5`
`n => n + 1`	`5`	`5 + 1 = 6`
"replace with `42`"	`6` (unused)	`42`

최종적으로 React는 42를 저장하고 `useState`에서 반환한다.

요약하면 다음과 같이 정리할 수 있다.

- 업데이터 함수가 Queue에 추가된다.
- 일반 함수가 Queue에 추가되며, 이미 Queue에 대기 중인 항목을 무시한다.

객체 State 업데이트

State는 객체를 포함한 모든 종류의 JavaScript 값을 가질 수 있다. 하지만 React state가 갖고 있는 객체를 직접 변경하는 것은 좋지 않다. 객체를 업데이트 할 때는 새로운 객체를 생성(혹은 기존 객체 복사)하여 state가 복사본을 사용하도록 하는 것이 좋다.

예시

```
const [x, setX] = useState(0);
```

숫자, 문자열, 불리언 값은 변경할 수 없거나 read only를 의미하는 Immutable한 상태를 갖는다.

```
setX(5);
```

x state는 0에서 5로 변경되지만, 숫자 0 자체가 변경되는 것은 아니다. JavaScript의 primary 값들은 변경이 불가능하다.

```
const [position, setPosition] = useState({ x: 0, y: 0 });
position.x = 5;
```

그러나 사실 객체 안에 정의되어 있는 프로퍼티는 기술적으로 변경이 가능하다. 그러나 객체도 Primary 값들처럼 불변성을 가진 것으로 다루는 것이 좋다. 객체를 직접 변경하는 것 대신 교체하는 것을 추천한다.

Use State like read only

```
const [position, setPosition] = useState({
    x: 0,
    y: 0
  });
  ...
onPointerMove={e => {
  position.x = e.clientX;
  position.y = e.clientY;
}}
```

예를 들어 위와 같은 코드는 정상적으로 동작하지 않는다. 그 이유는 React는 state Setter 함수가 없으면 객체가 변경되었는지 알 수 없기 때문이다. 따라서 리렌더링이 일어나지 않는다. 리렌더링을 발생시키려면 새 객체를 생성하고 state의 Setter 함수에 전달해야 한다.

```
onPointerMove={e => {
  setPosition({
    x: e.clientX,
    y: e.clientY
  });
}}
```

위 코드는 새로운 무명 객체를 생성하여 `setPosition`에 전달한 형태이기 때문에 리렌더링이 정상적으로 동작한다.

전개 문법으로 객체 복사

객체의 프로퍼티 중 하나 혹은 일부만 변경하는 경우가 있을 수 있다. 이런 경우 전개 문법 `...`을 사용할 수 있다.

```
setPerson({
  firstName: e.target.value, // input의 새로운 first name
  lastName: person.lastName,
  email: person.email
});
```

```
setPerson({
  ...person, // 이전 필드를 복사
  firstName: e.target.value // 새로운 부분은 덮어쓰기
});
```

`...` 전개 문법은 한 레벨 깊이의 내용만 복사하기 때문에 '얕다'는 점이 중요하다. 중첩된 프로퍼티를 업데이트 하려면 한 번 이상 사용해야 한다.

`<input>` DOM element의 `name` 프로퍼티를 사용하면 이런 식으로 표현할 수도 있다.

```
function handleChange(e) {
    setPerson({
      ...person,
      [e.target.name]: e.target.value
    });
  }
```

중첩된 객체 갱신

```
const [person, setPerson] = useState({
```

```
  name: 'Niki de Saint Phalle',
  artwork: {
    title: 'Blue Nana',
    city: 'Hamburg',
    image: 'https://i.imgur.com/Sd1AgUOm.jpg',
  }
});
```

만약 위 코드에서 `city`를 변경하기 위해서는 새로운 `artwork`객체를 생성한 뒤 `artwork`를 포함하는 `person`객체를 만들어야 한다.

```
const nextArtwork = { ...person.artwork, city: 'New Delhi' };
const nextPerson = { ...person, artwork: nextArtwork };
setPerson(nextPerson);
```

혹은 `...`전개 문법을 활용할 수도 있다.

```
setPerson({
  ...person, // 다른 필드 복사
  artwork: { // artwork 교체
    ...person.artwork, // 동일한 값 사용
    city: 'New Delhi' // 하지만 New Delhi!
  }
});
```

객체 중첩이라는 표현을 사용했지만 실제로는 중첩되어 있다기 보다는 감싸고 있는 객체가 프로퍼티 객체를 가리키고 있다고 하는 것이 정확한 표현이다.

Immer로 간결한 로직 작성

Immer는 복사본 생성을 도와주는 라이브러리이다. 코드가 더욱 간결해지고 직관적이다. Immer를 사용하기 위해서는 Package에 추가해야 한다.

```
const [person, updatePerson] = useImmer({
    name: 'Niki de Saint Phalle',
    artwork: {
      title: 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgUOm.jpg',
    }
  });

  function handleNameChange(e) {
    updatePerson(draft => {
      draft.name = e.target.value;
    });
  }
```

Immer는 draft라고 하는 Proxy 객체를 생성하고 draft를 변경한 후 수정된 draft를 새로운 객체로 반환하는 방식으로 동작한다. Immer를 사용하면 상태를 불변하게 가져가면서도 객체를 변경할 수 있다.

배열 state 업데이트

JavaScript에서 배열은 변경이 가능하지만 state로 저장할 때에는 변경할 수 없도록 처리하는 것이 좋다.

변경 없이 배열 업데이트

React state 내에서 배열을 다룰 땐 오른쪽 열에 있는 함수들로 변경하는 것이 좋다. 예를 들면, 새 배열을 state Setter 함수에 전달할 때 `filter()`, `map()`과 같이 원본 배열은 변경시키지 않으면서 새로운 배열을 반환하는 함수가 있다.

	비선호 (배열을 변경)	선호 (새 배열을 반환)
추가	`push`, `unshift`	`concat`, `[...arr]` 전개 연산자 (예시)
제거	`pop`, `shift`, `splice`	`filter`, `slice` (예시)
교체	`splice`, `arr[i] = ...` 할당	`map` (예시)
정렬	`reverse`, `sort`	배열을 복사한 이후 처리 (예시)

배열에 항목 추가

```
setArtists([
  ...artists,
  { id: nextId++, name: name }
]);
```

배열 전개 구문 `...`을 사용하여 배열 항목에 요소를 추가하는 `push()`와 같은 작업을 할 수 있다. 전개 구문을 뒤에 선언하면 `unshift()` 함수와 같은 작업 역시 가능하다.

```
export default function List() {
  const [name, setName] = useState('');
  const [artists, setArtists] = useState([]);

  return (
    <>
      <h1>Inspiring sculptors:</h1>
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
      <button onClick={() => {
        setName('');
        setArtists([
          ...artists,
          { id: nextId++, name: name }
        ]);
      }}>Add</button>
      <ul>
        {artists.map(artist => (
          <li key={artist.id}>{artist.name}</li>
```

```
      ))}
    </ul>
   </>
  );
 }
```

배열에 항목 제거

배열에 항목을 제거하는 쉬운 방법 중 하나는 `filter()` 함수를 사용하는 것이다. 기존 배열을 변경시키지 않으면서 새로운 배열을 반환한다.

```
setArtists(
 artists.filter(a => a.id !== artist.id)
);
```

```
let initialArtists = [
 { id: 0, name: 'Marta Colvin Andrade' },
 { id: 1, name: 'Lamidi Olonade Fakeye'},
 { id: 2, name: 'Louise Nevelson'},
];

export default function List() {
 const [artists, setArtists] = useState(
  initialArtists
 );

 return (
  <>
   <h1>Inspiring sculptors:</h1>
   <ul>
    {artists.map(artist => (
     <li key={artist.id}>
      {artist.name}{' '}
      <button onClick={() => {
       setArtists(
        artists.filter(a =>
         a.id !== artist.id
        )
       );
      }}>
       Delete
      </button>
     </li>
    ))}
   </ul>
  </>
 );
}
```

배열 변환

배열의 전체 항목을 변경하고자 할 때는 `map()` 을 사용해 새로운 배열을 만들 수 있다.

```
const nextShapes = shapes.map(shape => {
```

```
  if (shape.type === 'square') {
    // 변경시키지 않고 반환합니다.
    return shape;
  } else {
    // 50px 아래로 이동한 새로운 원을 반환합니다.
    return {
      ...shape,
      y: shape.y + 50,
    };
  }
});
// 새로운 배열로 리렌더링합니다.
setShapes(nextShapes);
```

배열 내 항목 교체

배열 내 하나의 항목을 교체할때도 `map()` 함수를 사용할 수 있다.

```
let initialCounters = [
  0, 0, 0
];

export default function CounterList() {
  const [counters, setCounters] = useState(
    initialCounters
  );

  function handleIncrementClick(index) {
    const nextCounters = counters.map((c, i) => {
      if (i === index) {
        // 클릭된 counter를 증가시킵니다.
        return c + 1;
      } else {
        // 변경되지 않은 나머지를 반환합니다.
        return c;
      }
    });
    setCounters(nextCounters);
  }

  return (
    <ul>
      {counters.map((counter, i) => (
        <li key={i}>
          {counter}
          <button onClick={() => {
            handleIncrementClick(i);
          }}>+1</button>
        </li>
      ))}
    </ul>
  );
}
```

배열 중간에 항목 삽입

중간에 삽입할 땐 `...` 전개 구문과 `slice()` 함수를 함께 사용해서 구현할 수 있다. `slice()`를 사용하면 배열의 일부분을 잘라낼 수 있다.

```
let nextId = 3;
const initialArtists = [
  { id: 0, name: 'Marta Colvin Andrade' },
  { id: 1, name: 'Lamidi Olonade Fakeye'},
  { id: 2, name: 'Louise Nevelson'},
];

export default function List() {
  const [name, setName] = useState('');
  const [artists, setArtists] = useState(
    initialArtists
  );

  function handleClick() {
    const insertAt = 1; // 모든 인덱스가 될 수 있습니다.
    const nextArtists = [
      // 삽입 지점 이전 항목
      ...artists.slice(0, insertAt),
      // 새 항목
      { id: nextId++, name: name },
      // 삽입 지점 이후 항목
      ...artists.slice(insertAt)
    ];
    setArtists(nextArtists);
    setName('');
  }

  return (
    <>
      <h1>Inspiring sculptors:</h1>
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
      <button onClick={handleClick}>
        Insert
      </button>
      <ul>
        {artists.map(artist => (
          <li key={artist.id}>{artist.name}</li>
        ))}
      </ul>
    </>
  );
}
```

배열 내부 객체 업데이트

배열 객체 안에 각 항목이 가리키는 객체는 별도의 값이다. (다른 곳에서 해당 값을 참조하고 있을 수 있다.) 그렇기 때문에 **중첩된 state를 업데이트할 때, 업데이트 하려는 지점부터 최상위 레벨까지의 복사본을 만들어야 한다.**

```
setMyList(myList.map(artwork => {
  if (artwork.id === artworkId) {
```

```
    // 변경된 *새* 객체를 만들어 반환합니다.
    return { ...artwork, seen: nextSeen };
  } else {
    // 변경시키지 않고 반환합니다.
    return artwork;
  }
}));
```

Immer로 업데이트

Immer를 사용하면 제공된 `draft`를 변경하는 것이기 때문에 `artwork.seen = nextSeen`과 같이 변경해도 괜찮다. 원본 state를 변경하는 것이 아니기 때문이다.

```
updateMyTodos(draft => {
  const artwork = draft.find(a => a.id === artworkId);
  artwork.seen = nextSeen;
});
```

```
import { useState } from 'react';
import { useImmer } from 'use-immer';

let nextId = 3;
const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
  const [myList, updateMyList] = useImmer(
    initialList
  );
  const [yourArtworks, updateYourList] = useImmer(
    initialList
  );

  function handleToggleMyList(id, nextSeen) {
    updateMyList(draft => {
      const artwork = draft.find(a =>
        a.id === id
      );
      artwork.seen = nextSeen;
    });
  }

  function handleToggleYourList(artworkId, nextSeen) {
    updateYourList(draft => {
      const artwork = draft.find(a =>
        a.id === artworkId
      );
      artwork.seen = nextSeen;
    });
  }

  return (
```

```
    <>
      <h1>Art Bucket List</h1>
      <h2>My list of art to see:</h2>
      <ItemList
        artworks={myList}
        onToggle={handleToggleMyList} />
      <h2>Your list of art to see:</h2>
      <ItemList
        artworks={yourArtworks}
        onToggle={handleToggleYourList} />
    </>
  );
}

function ItemList({ artworks, onToggle }) {
  return (
    <ul>
      {artworks.map(artwork => (
        <li key={artwork.id}>
          <label>
            <input
              type="checkbox"
              checked={artwork.seen}
              onChange={e => {
                onToggle(
                  artwork.id,
                  e.target.checked
                );
              }}
            />
            {artwork.title}
          </label>
        </li>
      ))}
    </ul>
  );
}
```

참고 자료 :

- 스냅샷으로서의 State – React
- state 업데이트 큐 – React
- 객체 State 업데이트하기 – React
- 배열 State 업데이트하기 – React