코드 수선 모임

1월 1일(목)

React 문서
상호작용성 더하기

https://ko.react.dev/learn/adding-interactivity

이벤트에 응답하기
State: 컴포넌트의 기억저장소
렌더링 그리고 커밋
스냅샷으로서의 State
state 업데이트 큐
~~객체 State 업데이트하기~~
~~배열 State 업데이트하기~~

이벤트 핸들러

화면의 일부 요소는 상호작용 기능이 붙어 사용자의 입력(click, hover, input ...)에 따라 업데이트

JSX

```jsx
export default function Button() {
  function handleClick() {
    alert('You clicked me!');
  }

  return (
    <button onClick={handleClick}>
      Click me
    </button>
  );
}
```

button 태그에 handleClick 이벤트 핸들러 붙이기

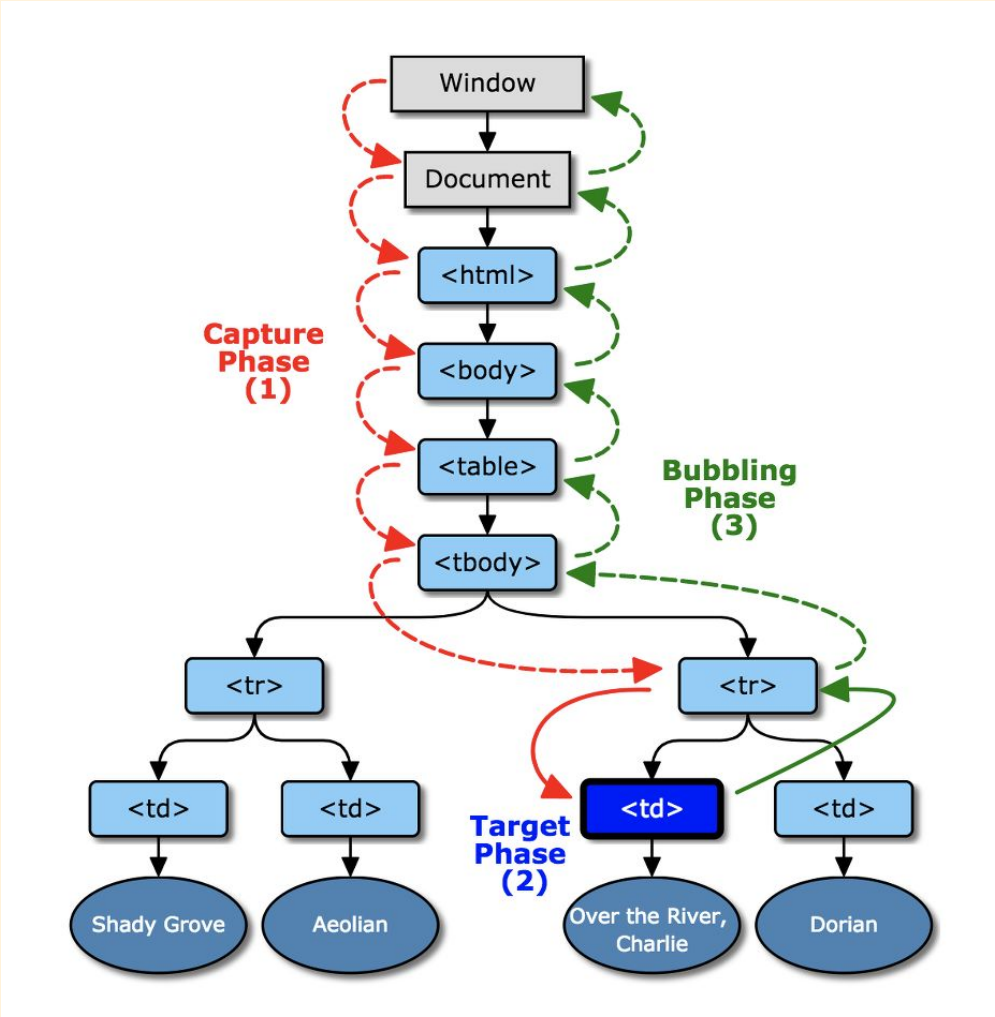
이벤트 전파

특정 이벤트가 발생하면
컴포넌트 트리의 하단부터 시작해서 해당 이벤트가 전파 (bubbling) 된다.

JSX

```jsx
export default function Toolbar() {
  return (
    <div className="Toolbar" onClick={() => {
      alert('You clicked on the toolbar!');
    }}>
      <button onClick={() => alert('Playing!')}>
        Play Movie
      </button>
      <button onClick={() => alert('Uploading!')}>
        Upload Image
      </button>
    </div>
  );
}
```

1. onClick 이벤트 발생
2. button의 onClick 이벤트 핸들러 호출
3. div의 onClick 이벤트 핸들러 호출


Window
Document
<html>
<body>
<table>
<tbody>
<tr>
<tr>
<td>
<td>
<td>
<td>
Shady Grove
Aeolian
Over the River, Charlie
Dorian
Capture Phase (1)
Bubbling Phase (3)
Target Phase (2)

W3C

이벤트 전파 멈추기

부모 컴포넌트의 이벤트 핸들러는 동작하지 않도록 막으려면 자식 컴포넌트에서 e.stopPropagation()을 호출하여 이벤트가 더 이상 bubbling 되지 않도록 방지!

JSX

```jsx
function Button({ onClick, children }) {
  return (
    <button onClick={e => {
      e.stopPropagation();
      onClick();
    }}>
      {children}
    </button>
  );
}

export default function Toolbar() {
  return (
    <div className="Toolbar" onClick={() => {
      alert('You clicked on the toolbar!');
    }}>
      <Button onClick={() => alert('Playing!')}>
        Play Movie
      </Button>
      <Button onClick={() => alert('Uploading!')}>
        Upload Image
      </Button>
    </div>
  );
}
```

이벤트 핸들러

이벤트 핸들러는 컴포넌트의 부수효과(사이드 이펙트)를 관리하는 역할을 한다.

- 컴포넌트 함수는 순수 함수여야 한다.
- 이벤트 핸들러가 사용자의 입력을 받고, 외부의 상태, 컴포넌트의 상태를 변경하는 것에 특화되어 있다.

Important

컴포넌트에서 **반환하는 JSX는 항상 순수**해야한다.
https://ko.react.dev/learn/keeping-components-pure#

- 동일한 입력이 주어지면 항상 동일한 JSX를 반환해야한다.
- 렌더링 이전에 존재했던 객체나 변수를 변경해서는 안된다. (렌더링이 발생할때마다 계산을 반복하여 값을 변경시킨다.)
    - 사이드 이펙트는 대부분 이벤트 핸들러를 사용해야한다. (렌더링 중에는 실행되지 않도록)
    - 최후의 수단으로 `useEffect` 를 사용한다.
        - `useEffect` 의 콜백함수는 렌더링 이후에 실행된다.

https://ko.react.dev/learn/keeping-components-pure

State : 컴포넌트별 메모리

컴포넌트는 사용자와의 상호 작용의 결과로 화면의 내용을 변경해야 한다.

현재 입력값, 현재 지정된 이미지, 선택한 아이템등을 "기억"해야 할 필요가 있고 state는 컴포넌트별 메모리다.

`state`는 각 컴포넌트의 지역 상태다.

state는 각 컴포넌트의 지역 상태다.

동일한 컴포넌트를 두번 사용하더라도, 각 컴포넌트는 격리된 상태를 가진다.

- 부모 컴포넌트는 자식 컴포넌트의 `state`를 변경할 수 없다.
- 복수의 자식 컴포넌트가 같은 `state`를 공유해야 한다면, `상태 끌어올리기`를 해야한다.

JSX

```jsx
return (
    <div className="Page">
      <Gallery />
      <Gallery />
    </div>
  );
```

state vs. 일반 변수

```
let index = 0;

function handleClick() {
    index = index + 1;
}
```

- 지역 변수는 렌더링 간에 유지되지 않는다.
- 지역 변수를 변경해도 렌더링을 일으키지 않는다.

```
const [index, setIndex] = useState(0);

function handleClick() {
    setIndex(index + 1);
}
```

- state 변수 : 렌더링 간에 데이터를 유지하기 위한
- state setter 함수 : 변수를 업데이트하고 React가 컴포넌트를 다시 렌더링 하도록 한다.

state를 설정하면 렌더링이 동작한다.

- 인터페이스가 이벤트에 반응하려면 state를 업데이트해야 한다.
- 이벤트(클릭) -> UI 변경 (X)
- 이벤트(클릭) -> state 변경 -> 렌더링 -> UI변경 (O)

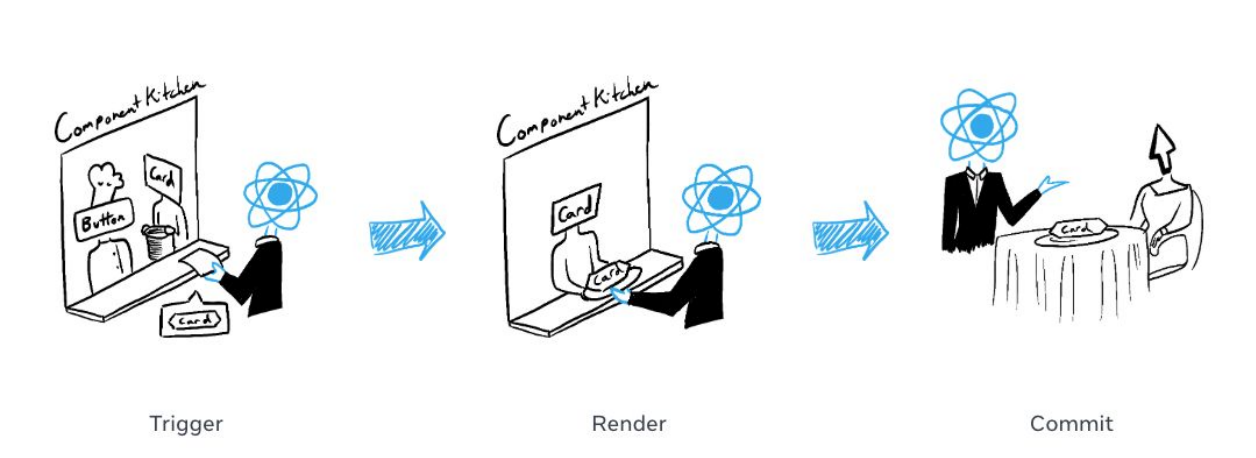
렌더링 조건

- 초기 렌더링 (처음 화면에 진입하면 렌더링)
- useState의 setState가 실행되는 경우
- useReducer의 dispatch가 실행되는 경우
- 컴포넌트의 key props가 변경되는 경우
- 부모 컴포넌트가 리렌더링 되는 경우

렌더링 그리고 커밋

UI를 요청하고 제공하는 3가지 단계

- 1단계: 렌더링 트리거 (손님의 주문을 주방으로 전달)
- 2단계: 컴포넌트 렌더링 (주방에서 주문 준비하기)
- 3단계: DOM에 커밋 (테이블에 주문한 요리 내놓기)



1단계: 렌더링 트리거

1. 컴포넌트의 초기 렌더링인 경우

➢ createRoot를 호출하고 해당 컴포넌트로 `render`를 호출하는 과정이 내부 동작에 있다

```
const root = createRoot(document.getElementById('root'))
root.render(<Image />);
```

2. 리렌더링인 경우: 컴포넌트의 state가 업데이트, 부모 컴포넌트가 호출된 경우

➢ 컴포넌트의 상태를 업데이트하면 렌더링 대기열에 추가된다.

2단계: React 컴포넌트 렌더링

1. 컴포넌트의 초기 렌더링인 경우

- 루트 컴포넌트를 호출한다.
- 자식 컴포넌트들이 연이어서 호출된다.
- 렌더링 단계에서 Virtual DOM 노드를 생성한다.

2. 리렌더링인 경우

- state가 업데이트된 컴포넌트와 자식 컴포넌트들을 연이어서 호출
- React는 현재 Virtual DOM 트리와 이전 Virtual DOM 트리를 비교하여 차이점만을 계산(diff), 이 과정을 reconciliation

3단계: React가 DOM에 변경사항을 커밋

1. 컴포넌트의 초기 렌더링인 경우

➢ appendChild() DOM API를 사용하여 생성한 모든 DOM 노드를 화면에 표시.

2. 리렌더링인 경우

➢ 계산한 diff 결과(이전 렌더링과 비교하여 변경된 DOM)를 화면에 반영

브라우저 페인트

- ➢ 렌더링이 완료되고 React가 DOM을 업데이트한 후, 브라우저는 화면을 다시 그린다.
- ➢ 이를 **브라우저 렌더링**이라고 하지만, React의 렌더링과 구분하기 위해 **페인팅**이라고 부른다 (React docs)

스냅샷으로서의 State

- 컴포넌트, 함수에서 반환하는 JSX는 UI의 스냅샷 (그 시점의 모습)
- prop, 이벤트 핸들러, 지역 변수는 모두 **렌더링 시점의 state**를 사용해 계산된다

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

- 하나의 렌더링 사이클 안에서 **number 변수의 값은 고정되어 있다**.
- 따라서 세 번 모두 동일하게 **0 + 1**을 수행한다.

State 업데이트 큐

- **batch**: 컴포넌트에서 상태 업데이트가 발생할 때마다 업데이트를 바로 적용하지 않고 업데이트 큐에 추가. 이벤트 핸들러의 모든 코드가 실행될 때까지 기다린다.

- 여러 상태 업데이트를 단일 리렌더링으로 그룹화하여 성능을 향상 ✅

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

batch처리 때문에 다음 렌더링 전까지 number의 상태가 갱신되지 않음

업데이트 함수

- ➢ 하나의 이벤트에서 일부 **state**를 여러 번 업데이트하려면 업데이트 함수를 사용한다.
- ➢ 이전 큐의 **state**를 기반으로 다음 **state**를 계산

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

queued update	n	반환
n => n + 1	0	0 + 1 = 1
n => n + 1	1	1 + 1 = 2
n => n + 1	2	2 + 1 = 3

업데이트 큐를 처리하고 최종 state를 얻는 로직

```tsx
export function getFinalState(
  baseState: number,
  queue: (number | ((n: number) => number))[]
) {
  let finalState = baseState;

  for (let update of queue) {
    if (typeof update === 'function') {
      // 큐의 인자 함수일경우
      finalState = update(finalState);
    } else {
      // 큐의 인자가 숫자일 경우
      finalState = update;
    }
  }

  return finalState;
}
```

```tsx
// 현재 number = 0 가정
setNumber(number + 1); // 0 + 1 -> 큐: [1]
setNumber(number + 1); // 0 + 1 -> 큐: [1, 1]
setNumber(number + 1); // 0 + 1 -> 큐: [1, 1, 1]
// 결과 -> 1
```

```tsx
// 현재 number = 0 가정
setNumber(n => n + 1);
setNumber(n => n + 1);
setNumber(n => n + 1);
// 큐: [f, f, f] (각각 n => n + 1 함수)
// 결과 -> 3
```

```tsx
// 현재 number = 0 가정
setNumber(number + 5); // 0 + 5 -> 큐: [5]
setNumber(n => n + 1); // 업데이트 함수 전달 -> 큐: [5, n => n + 1]
setNumber(42);         // 숫자 42 전달 -> 큐: [5, n => n + 1, 42]
// 결과 -> 42
```