[React-deep-dive]-10장.리액트 17 과 18의 변경사항

리액트 16 → 리액트 17
    점진적인 업그레이드
    이벤트 위임 방식의 변경
    새로운 JSX transform
    이벤트 풀링 제거
    `useEffect` 클린업 함수의 비동기 실행 (동기→ 비동기)
    컴포넌트의 undefined 반환에 대한 일관적인 처리

리액트 16 → 리액트 17

결론 : 새롭게 추가된 기능 x, 버전 업그레이드시 코드의 수정을 필요로 하는 변경사항을 최소화 했다.

점진적인 업그레이드

보통 버전의 업그레이드는 **유의적 버전 전략** 이라는 것을 따른다.

https://velog.io/@rkddl6803/유의적-버전Semantic-Versioning

유의적 버전(Semantic Versioning, Semver)

노드js, nvm, npm 등에는 버전이 항상 xx.xx.x 혹은 x.xx.x 등으로 표기된다.
이와 같은 표기는 유의적 버전 표기인데 이것은 버전에 의미를 부여해서 구분하고 해석할 수 있다는 뜻이다.
버전 각자리 숫자의 의미는 다음과 같다.

> Major.Minor.Patch
>
> **Major** : 기존 버전과 호환되지 않는 새로운 버전
> **Minor** : 기존 버전과 호환되는 새로운 기능이 추가된 버전
> **Patch** : 기존 버전과 호환되는 버그 및 오타 등이 수정된 버전

만약 아래와 같이 버전 맨 앞에 캐롯 기호가 붙어있다면 다음과 같은 의미이다.

> **^Major.Minor.Patch**
>
> **^** : Major 버전 안에서 가장 최신 버전으로 업데이트 가능

예 : 16.1.0이 깔려있는데 ^16.1.0라고 뜬다면 16.1.0 버전 이상의 16 버전의 가장 최신 버전으로 업데이트 가능하다는 뜻이다.

17부터는 점진적 업그레이드를 지원하기 위한 리액트의 일부 컴포넌트 변경

18버전 부터는 한 어플리케이션의 두개의 리액트가 존재할수 있음, **즉, 17은 17,18이 공존하기 위한 중간 절차**

리액트에서 권장하는 방법은 아니지만, 규모가 커 한꺼번에 새로운 버전으로 업그레이드 하기 부담스러울때 유용한 시나리오

https://velog.io/@typo/how-airbnb-smoothly-upgrades-react

이벤트 위임 방식의 변경

일반 버튼의 추가 방식

- 직접 DOM을 참조해서 가져온 뒤, DOM의 onclick 이벤트를 추가

```
const buttonRef = useRef<HTMLButtonElement>(null)

useEffect(()⇒{
   if (buttonRef.current) {
      buttonRef.current.onClick = function click () {
         alert("hi")
         }
      }
},[])

...
<button ref={buttonRef}>hi</button>
```

리액트 버튼의 이벤트 추가 방식

- 리액트 어플리케이션에서 DOM에 이벤트를 추가하는 방식
- 이벤트 핸들러를 해당 DOM요소에 부탁하는 것이 아니라, 이벤트 위임을 통해 루트에 부착한다
- 이벤트 위임이란 이벤트의 구성단계의 원리를 활용해 이벤트를 상위 컴포넌트에만 붙이는것을 의미한다

```
function hello () {
   alert("hello")
}
...
<button onClick={hello} >hello</button>
```

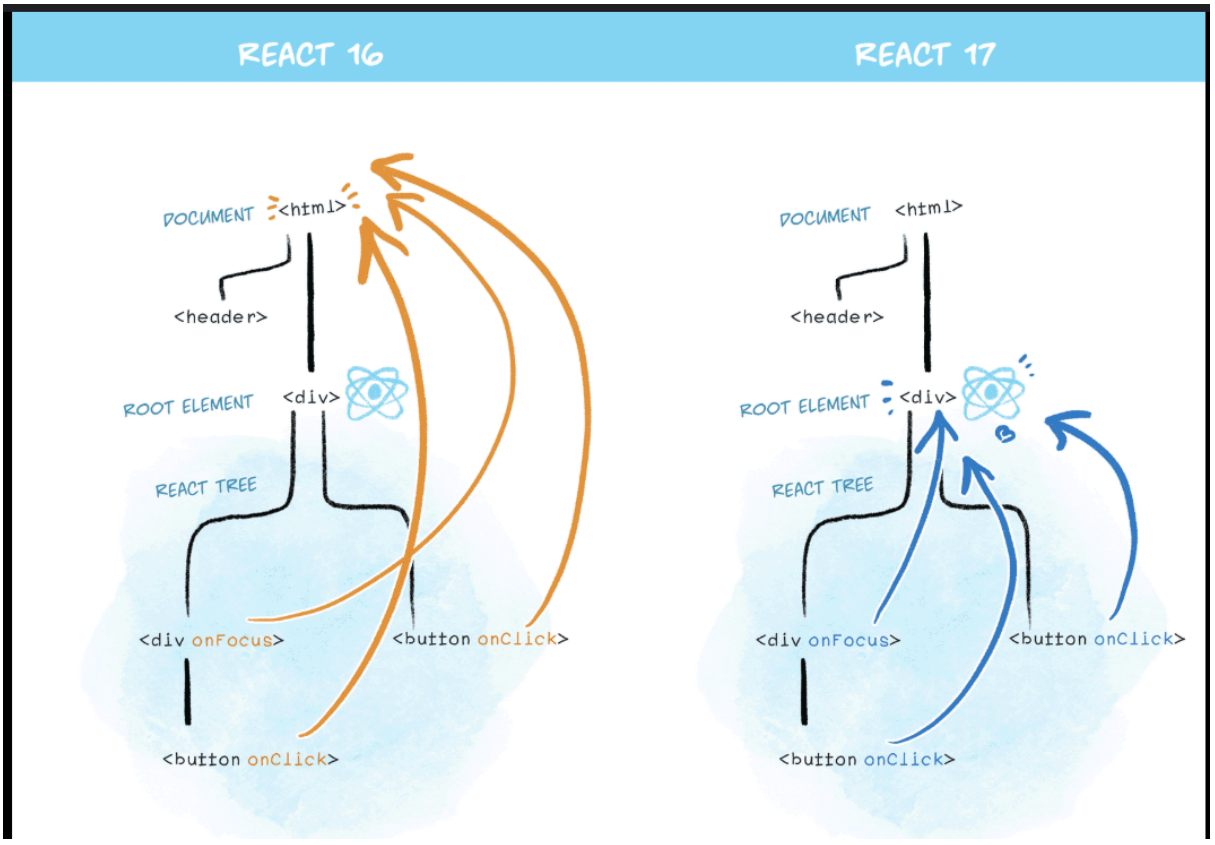
이벤트 구성단계

1. **캡처**
    - 이벤트 핸들러가 트리 최상단 요소에서 부터 시작해서 실제 이벤트가 발생한 타깃 요소까지 내려가는 것을 의미한다.

2. **타깃**
    - 이벤트 핸들러가 타깃 노드에 도달하는 단계다. 이 단계에서 이벤트가 호출된다.
3. **버블링**
    - 이벤트가 발생한 요소에서 부터 시작해 최상위 요소까지 다시 올라간다.

17에서의 이벤트 위임

결론 : 점진적 업그레이드 지원을 위해 document 에서 이벤트 위임 수행에서 리액트 컴포넌트 최상단 트리, 즉 루트 요소에서 수행으로 바뀌었다.



React 16

이미 모든 이벤트가 document로 올라가 있는 상태기 때문에 `e.stopPropagation()` 을 실행해도 이벤트 전파를 막을수 없었다.

React 17

각 이벤트가 해당 리액트 컴포넌트 트리 수준으로 격리 되므로 이벤트 버블링으로 인한 혼선을 방지

점진적인 업그레이드 지원을 위해, 다른 바닐라 자바스크립트 코드 또는 jQuery등이 혼재돼 있는 경우 혼란을 방지하기 위해

새로운 JSX transform

```
const Component = (
 <div>
  <span>hello world</span>
 </div>
)
```

```
//구버전
//React.createElement 를 수행할 때 import React from 'react'가 필요 했다.
  var Component = React.createElement(
  "div",
  null,
  React.createElement("span", null, "hello world")
);
```

```
//17버전
//JSX를 변환할 때 필요한 모듈인 react/jsx-runtime을 불러오는 require 구문도 같이
var _jsxRuntime = require('react/jsx-runtime')

var Component = (0,_jsxRumtime.jsx)('div',{
  children:(0,_jsxRuntime.jsx)('span',{
    children:"hello world"
  })
})
```

→ 컴포넌트 작성의 간결화, 번들링 크기 약간 줄임 `react-codemode` 명령어로 지울수 있음

이벤트 풀링 제거

결론 : 리액트 16의 이벤트 풀링 개념이 삭제 되었다.

이벤트 풀링 (In React 16)

리액트는 한번 래핑한 이벤트를 사용하기 때문에 이벤트가 발생할 때마다 이 이벤트를 새로 만들어야 했고, 그 과정에서 야기되는 메모리 누수 해결을 위해 이벤트를 주기적으로 해제해야 하는 번거로움이 있다.

따라서 이벤트 풀링 이라는 것으로 `SyncthеticEvent` 라는 이벤트 객체로 풀을 만들어서 이벤트가 발생할 때마다 가져오는것을 의미한다.

이벤트 풀링 시스템에서의 이벤트 발생 플로우

1. 이벤트 핸들러가 이벤트를 발생
2. 합성 이벤트 풀에서 합성 이벤트 객체에 대한 참조를 가져온다.
3. 이 이벤트 정보를 합성 이벤트 객체에 넣어준다.
4. 유저가 지정한 이벤트 리스너가 실행된다.
5. 이벤트 객체가 초기화(null) 되고 다시 이벤트 풀로 돌아간다.

이벤트 풀링 개념의 단점

이벤트가 종료 되자마자 초기화 되므로, 비동기 코드 내부에서 event 객체의 속성값이 null 이 되는 문제가 발생,

`event.persist()` 로 처리해주어야 했었다.

`useEffect` 클린업 함수의 비동기 실행 (동기→ 비동기)

```
useEffect(()⇒ {
   어쩌구 동작~
   return ()⇒ {
      클린업~~
   }
},[])
```

리액트 16 버전까지 useEffect 의 클린업 함수가 동기적으로 처리되었다.

이는 클린업 함수가 완료되기 전까지 다른 작업을 방해하기 때문에 불필요한 성능 저하로 이어지는 문제가 존재했다.

리액트 17버전부터는 화면이 완전히 업데이트된 이후에 클린업 함수가 비동기적으로 실행된다.

컴포넌트의 undefined 반환에 대한 일관적인 처리

일단 컴포넌트에서 `undefined` 를 반환하면 에러가 발생하는데,

16에서는 `forwardRef` , `memo` 에서 `undefined` 를 반환하는 경우에는 에러가 발생하지 않는 문제가 있었다.

17에서는 일관적으로 에러가 발생한다.

결론 : 16(`forwardRef` , `memo` 제외) → 17(일관적으로 에러) → 18에서는 갑자기 아예 에러로 취급되진 않는다.

▼

리액트 18

새로 추가된 훅

useId

컴포넌트별로 유니크한 값을 생성하는 새로운 훅이다.

같은 컴포넌트여도 서로 인스턴스가 다르면 다른 랜덤한 값을 만들어 내며

서버사이드와 클라이언트 간에 동일한 값이 생성되어 하이드레이션 이슈도 발생하지 않는다.

Math.random() 값을 서버사이드 렌더링 시 사용하게 된다면 하이드레이션 이슈가 발생한다.

```
const id = useId();
```

useTransition

UI 변경을 가로막지 않고 상태를 업데이트할 수 있는 리액트 훅이다.

이 훅은 상태 업데이트를 긴급하지 않은 것으로 간주해 무거운 렌더링 작업을 조금 미룰 수 있으며, 사용자에게 조금 더 나은 사용자 경험을 제공할 수 있다.

렌더링이 오래걸리는 컴포넌트의 렌더링을 동기적으로 기다리지 않고 비동기적으로 렌더링 할 수 있게 해주는 훅이다.

사용법

`useTransition`은 매개변수를 받지 않는다.

`useTransition`은 정확히 두 개의 항목이 있는 배열을 반환하는데

- 보류 중인 트랜지션이 있는지 여부를 알려주는 `isPending` 플래그
- state 업데이트를 트랜지션으로 표시할 수 있는 `startTransition` 함수

```
const [isPending, startTransition] = useTransition();
```

useDeferredValue

리액트 컴포넌트 트리에서 리렌더링이 급하지 않은 부분을 지연할 수 있게 도와주는 훅이다.

useTransition과의 차이점은 감싸는 부분이 useTransition 은 state값을 업데이트하는 함수이고 useDeferredValue 는 state값 자체만을 감싸서 사용한다는 것이다.

방식에서만 차이가 있을 뿐 상황에 맞는 방법을 선택하여 사용하면 된다.

```
const deferredText = useDeferredValue(text);
```

useSyncExternalStore

useSubsription이 리액트 18버전에서 useSuncExternalStore로 대체되었다.

리액트 17에서는 일어날 여지가 없었던 tearing 현상 (하나의 state값이 서로 다른 값으로 렌더링되는 현상) 이 18버전에서 useTransition, useDeferredValue의 훅처럼 렌더링을 일시 중지하거나 뒤로 미루는 등의 최적화가 가능해지며 동시성 이슈가 발생할 수 있게 되었고 리액트 밖의 값에 대해서 (innerWidth 같은 경우) 해당 현상에 대한 이슈 해결을 위해 등장하게 된 훅이다.

구성

- 첫번째 인수는 subscribe로, 콜백 함수를 받아 스토어에 등록하는 용도로 사용된다. 스토어의 값이 변경되면 이 콜백이 호출되어야 한다.
- 두번째 인수는 컴포넌트에 필요한 현재 스토어의 데이터를 반환하는 함수다.

  스토어가 변경되지 않았다면 매번 함수를 호출할 때마다 동일한 값을 반환해야 한다.
- 마지막인수는 옵셔널 값으로 서버 사이드 렌더링 시에 내부 리액트를 하이드레이션하는 도중에만 사용된다. 서버사이드에서 렌더링하는 훅이라면 반드시 이 값을 넘겨줘야 하며, 클라이언트의 값과 불일치가 발생할 경우 오류가 발생한다.

useInsertionEffect

기본적인 훅 구조는 useEffect와 동일하다.

다른 차이점은 실행 시점인데, useInsertionEffect는 DOM이 실제로 변경되기 전에 동기적으로 실행된다.

이러한 차이는 브라우저가 다시 스타일을 입혀서 DOM을 재계산하지 않아도 된다는 점에서 매우 크다.

하지만 라이브러리를 작성하는 경우가 아니라면 참고만 하고 실제 애플리케이션 코드에는 가급적 사용하지 말도록 하자

---

react-dom/client

createRoot

기존의 react-dom에 있던 render 메서드를 대체할 새로운 메서드다.

```
const root = ReactDOM.createRoot(container)
root.render(<App/>)
```

hydrateRoot

서버 사이드 렌더링 애플리케이션에서 하이드레이션을 하기 위한 새로운 메서드다.

```
const root = ReactDOM.hydrateRoot(container)
```

서버 사이드 렌더링을 자체적으로 구현해서 사용할 경우 이 부분을 수정해야 한다.

자동배치(Automatic Batching)

automatic batching 은 여러개의 상태변화를 하나의 리렌더링으로 묶어서 성능을 향상시키는 방법을 말한다.

리액트 17버전에서는 settimeout , promise 같은 비동기 이벤트에서는 자동배치가 이뤄지고 있지 않았기 때문에 해당 부분에서 자동배치가 일어나지 않았지만

리액트 18버전에서 부터는 루트 컴포넌트를 createRoot를 사용해서 만들며 모든 업데이트가 배치 작업으로 최적화 할 수 있게 되었다.

더욱 엄격해진 엄격 모드

리액트의 엄격모드

리액트 엄격모드는 리액트에서 제공하는 컴포넌트 중 하나로 개발을 하는 동안 잠재적인 버그를 찾아내는데 도움이 되는 컴포넌트다.

해당 모드는 개발모드에서만 작동하고 프로덕션 모드에서는 작동하지 않는다.

- 더 이상 안전하지 않은 특정 생명주기를 사용하는 컴포넌트에 대한 경고
- 문자열 ref 사용 금지
- findDOMNode 에 대한 경고 출력

    - findDOMNode 는 클래스 컴포넌트 인스턴스에서 DOM요소에 대한 참조를 가져올 수 있는 현재는 사용하지 않는 것을 권장하는 메서드다.
- 구 Context API 사용 시 발생하는 경고
    - childContextTypes 와 getChildContext를 사용하는 구 contextApi 를 사용하면 에러가 발생한다.
- 예상치 못한 부작용 검사
    - 리액트 엄격 모드 내부에서는 아래의 내용을 의도적으로 이중으로 호출한다.

      이는 리액트 함수가 항상 순수한 결과물을 내고 있는지 개발자에게 알려주기 위함이다.

        - 클래스 컴포넌트의 constructor, render, shouldComponentUpdate, getDerivedStateFromProps
        - 클래스 컴포넌트의 setState의 첫번째 인수
        - 함수 컴포넌트의 body
        - useState, useMemo, useReducer 에 전달되는 함수

Suspense 기능 강화

suspense는 컴포넌트를 동적으로 가져올 수 있게 도와주는 기능이다.

인수로 두 개를 받게 되는데 하나는 fallback prop 으로 지연시켜 불러온 컴포넌트가 아직 불러와지지 않았을 때 보여주는 fallback을 나타낸다.

그리고 children 으로는 React.lazy로 선언한 지연 컴포넌트를 받는다.

즉 지연 컴포넌트를 로딩하기 전에 fallback prop을 보여주고 로딩이후에는 지연 컴포넌트를 보여주는 것이다.

리액트 18버전에서는 Next에서도 해당 suspense 기능을 사용할 수 있게 공식적으로 지원이 되고 마운트 되기 이전에 effect가 실행되는 버그가 수정되었다 .