10장 리액트 17과 18의 변경사항 살펴보기

React 17 & 18

240414 두선아

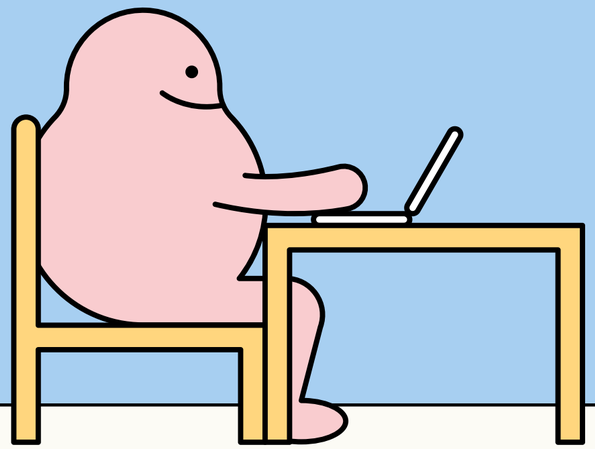
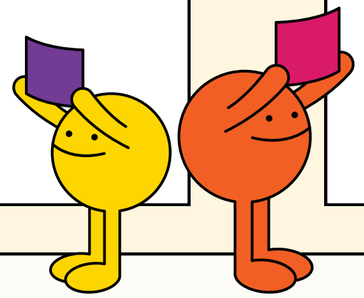


table of contents



React는?

npm 다운로드

리액트는 2013년 5월 29일 출시 이후
npm stats 기준으로 15억 7천 다운로드

npm-stats

react

1,570,813,991

TOTAL DOWNLOADS

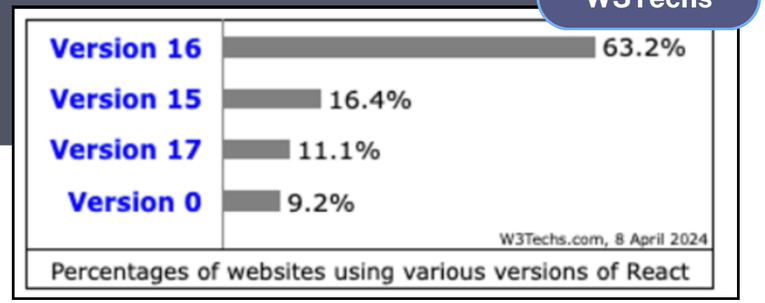
1st January 2013 - 13th April 2024

PERIOD

버전

현재 리액트: v18.2.0
인터넷에서 가장 많이 사용되는 버전: v16

W3Techs



유명 웹사이트

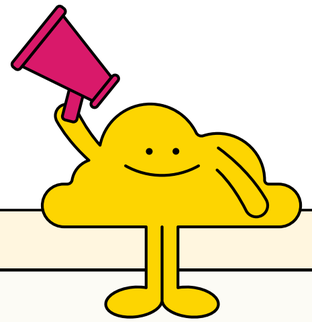
에어 비앤비 Airbnb
넷플릭스 Netflix

Airbub

```
rendered by
    legacy - App
    react-dom@18.2.0
```

Netflix

```
rendered by
    react-dom@18.2.0
```



Goal

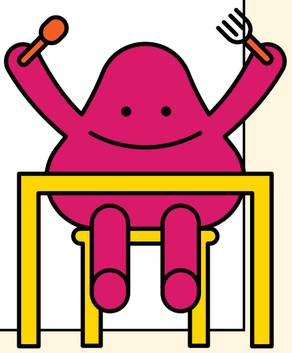
가장 많이 사용되는 버전인 16
따라서 16.8 이후 고착화된 서비스가 많을 것

버전 업그레이드를 왜 해야 할까?

- 모든 라이브러리가 그렇듯, 리액트도 더 나은 기능과 성능을 위해 버전업
- 버전 16.8의 Hook처럼 획기적인 업그레이드가 있을 수도 있음
- 리액트에 의존하는 라이브러리를 사용할 시, peerDependencies를 고려해야 함

리액트 16으로부터 18까지 업데이트 내용 살펴보기

- 17, 18에서 어떤 변화가 있었는 지 살펴보기



React 17

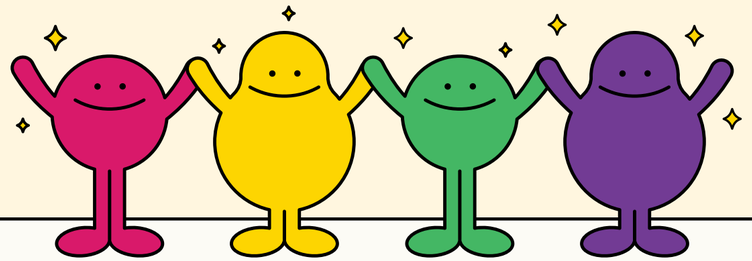
점진적 업그레이드

이벤트 위임 방식의 변경

event pooling 제거

useEffect 클린업 함수의 비동기 실행

컴포넌트의 undefined 반환에 대한 일관적 처리



Gradual Upgrades

- semantic version을 기반으로 업데이트
  => 기존 버전과 호환되지 않기 때문에, major 버전을 올리기 힘들다.
- React 17 버전부터 점진적 업그레이드를 지원한다.
  => 일부 트리, 컴포넌트에 대해서만 버전을 업그레이드
- mordern(17)과 legacy(16, lazy loading)로 구성된 예제 repository
  => 랜더링 과정에서 버전 불일치로 인한 에러 발생x
  => 두 개의 react root: 컴포넌트, 훅, Context를 서로 불러와서 사용할 수 있음

For most apps, upgrading all at once is still the best solution

---

Gradual Upgrades: https://legacy.reactjs.org/blog/2020/10/20/react-v17.html#gradual-upgrades
demo source code: https://github.com/reactjs/react-gradual-upgrade-demo/

event delegation 방식의 변경

- handler()

```jsx
import { useEffect, useRef } from "react";

export default function Button() {
  const buttonRef = useRef<HTMLButtonElement | null>(null);

  useEffect(() => {
    if (buttonRef.current) {
      buttonRef.current.onclick = function 자갈치() {
        alert("🐙");
      };
    }
  }, []);

  function 오사쯔() {
    alert("📣");
  }

  return (
    <>
      {/* 리액트 애플리케이션에서 DOM에 onClick 이벤트를 추가하는 방식 */}
      {/* handler: noop(), no operation, 암것도 하지 않음 */}
      <button onClick={오사쯔}>리액트 버튼</button>

      {/* DOM을 참조한 다음, DOM의 onclick에 직접 함수 추가하는 고전적인 이벤트 핸들러 추가 방식 */}
      {/* handler: 자갈치() */}
      <button ref={buttonRef}>DOM 버튼</button>
    </>
  );
}
```

noop()

자갈치()

React & Event Handler

- 리액트의 경우, 최초 릴리즈부터 event delegation을 적극적으로 사용
  - 해당 이벤트 핸들러를 추가한 각각의 DOM 요소에 부착하지 않음
  - 이벤트 타입당 하나의 핸들러를 root에 부착

handler를 root에 부착

버전 16: document
버전 17: 리액트 컴포넌트 최상단 트리 (root element)

이벤트 구성 단계

단계	내용
capture	이벤트 핸들러가 트리 최상단 요소에서부터, 실제 이벤트가 발생한 타깃 요소까지 내려가는 것
target	이벤트 핸들러가 타깃 노드에 도달 / 이벤트 호출
bubbling	이벤트가 발생한 요소에서부터 최상위 요소까지 올라가는 것

🌳 document의 handler와 점진적 업그레이드

- 이벤트를 document가 아닌, 해당 리액트 컴포넌트 트리 수준으로 격리 (이벤트 버블링 혼선을 막음)
- JQuery 등 다른 라이브러리에도 통용됨

😮 버전 16버전에서 root element 에 event stopPropagation 하면 모든 이벤트를 막을 수 있었다~

- 이 때 document.addEventListener로 리액트의 모든 이벤트를 확인하는 코드가 있었다면?

JSX transfrom

- JSX는 브라우저가 이해할 수 있는 코드가 아님
  - 바벨, 타입스크립트 컴파일이 필요함
- 16까지는 JSX 변환을 위해 'react' import가 필요했고, 없으면 에러
  - react.createElement
- 17부터는 바벨과 협력해 이러한 import 구문 없이도 JSX 변환
  - _jsxRuntime
- import 삭제하니까 번들링 크기 조금이나마 줄임
  - 컴포넌트 작성 간결화. 내부 로직또한 간결하다.

그러므로 import ‘react’ 지우고, tsconfig.json의 jsx는 "react-jsx"!

event pooling 제거

- 리액트 16의 이벤트 풀링
  - SyntheticEvent 풀을 만들어서, 이벤트가 발생할 때마다 가져옴
- (메모리 할당) 한 번 래핑한 이벤트를 사용하기 때문에, 이벤트가 발생할 때마다 이벤트를 새로 만들어야 함
  - 메모리 누수를 방지하기 위해 주기적으로 해제하는 번거로움
- 이벤트가 종료되자마자 초기화하는 방식은 직관적이지 않았다😢
  - 이벤트 풀링 방식으로, 이벤트 객체를 사용한다.
  - 재사용 사이, 모든 이벤트 필드를 null로 변경(재사용을 위해 초기화)
  - 초기화된 이후에 e에 접근하면 null
    - e.persist()같은 별도 처리가 필요하다

SyntheticEvent

브라우저의 기본 이벤트를 한 번 더 감싼 이벤트 객체

이벤트 풀링 시스템

1. 이벤트 핸들러가 이벤트를 발생 시킴
2. 합성 이벤트 풀에서 합성 이벤트 객체에 대한 참조를 가져옴
3. 이벤트 정보를 합성 이벤트 객체에 넣는다
4. 유저가 지정한 이벤트 리스너 실행
5. 이벤트 객체 초기화 => 다시 이벤트 풀~

즉, 별도 메모리 공간에 합성 이벤트 객체를 할당해야 한다.
성능 향상에 도움이 되지 않으므로 event pooling 개념이 삭제되었다.
또한 모던 브라우저 성능이 개선되어, 이벤트 핸들러 내부에서 이벤트 객체에 접근할 때 비동기든 동기든 상관없음

useEffect 클린업 함수의 비동기 실행

- useEffect의 클린업 함수는 16버전까지는 동기
  - 불필요한 성능 저하가 발생
- 17부터는 화면이 완전히 업데이트 된 이후, 클린업 함수가 비동기적으로 실행
  - 클린업 함수는 컴포넌트의 커밋 단계가 완료될 때까지 지연
- "화면의 업데이트가 완전히 끝난 이후"에 실행되도록 바뀌었다.
  - 성능적 이점 (Profiler로 commitTime을 확인할 수 있다)

Profiler

https://react.dev/reference/react/Profiler

'<Profiler>' lets you measure rendering performance of a React tree programmatically.

컴포넌트의 undefined 반환

- undefined에 대한 일관적 처리

react	케이스	에러 여부
16, 17	컴포넌트가 undefined를 반환	O
16	memo, forwardRef가 undefined를 반환	X
17	memo, forwardRef가 undefined를 반환	O
18	컴포넌트가 undefined를 반환, memo, forwardRef가 undefined를 반환	X

- 17.0.0 더 살펴보기
  - https://github.com/facebook/react/releases/tag/v17.0.0
- 리액트 16 애플리케이션이 있다면 17로 업데이트 하는 것이 좋다
  - 점진적 업그레이드 대비
  - 리액트팀 피셜, 16 -> 17 업데이트의 공수가 크기 않음
    - 10만 개의 컴포넌트 중 호환성이 깨지는 변경 사항은 20개 미만이라 한다

2022.3

React 18

Concurrent Rendering

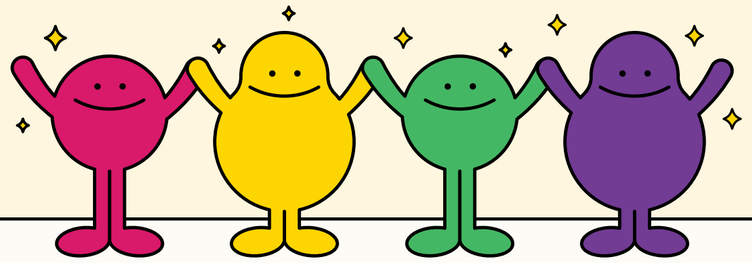
react-dom/client

react-dom/server

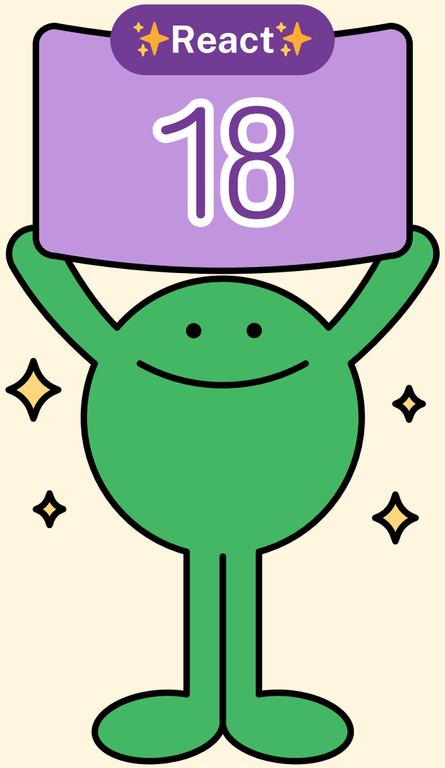
Automatic Batching

Strict Mode - Updated!

Suspense - Updated!



Concurrent Rendering



- 리액트 버전 18: 성능 향상과 렌더링 엔진 개선
- 동시성 지원!
  - 한 번에 둘 이상의 작업이 동시에 진행되는 것
- 리액트 18 이전의 렌더링은 개입할 수 없는 하나의 동기적인 처리였다.

The most important addition in React 18📌
is something we hope you never have to think about: “concurrency”

---

https://legacy.reactjs.org/blog/2022/03/29/react-v18.html#what-is-concurrent-react

useId

컴포넌트별로 유니크한 값을 생성

- 유니크한 ID
  - 컴포넌트 트리의 컴포넌트 ID는 고유성을 유지해야함
  - 서버 사이드 랜더링 환경에서 하이드레이션 시 서버와 동일한 값을 가져야 함
- useId가 생성한 값, :값:
  - CSS selector, querySelector 작동 x
- 생성 알고리즘
  - id는 현재 트리에서 자신의 위치를 나타내는 32글자의 이진 문자열
    - R은 서버사이드 생성 id
    - r은 클라이언트 생성 id

```
function mountId(): string {
  ...
  const root = ((getWorkInProgressRoot(): any): FiberRoot);
  const identifierPrefix = root.identifierPrefix;

  let id;
  if (getIsHydrating()) { // 하이드레이션이라면? 즉 server-generated 이라면?
    const treeId = getTreeId();

    // Use a captial R prefix for server-generated ids.
    id = ':' + identifierPrefix + 'R' + treeId;
    // Unless this is the first id at this level, append a number at the end
    // that represents the position of this useId hook among all the useId ✅
    const localId = localIdCounter++;
    if (localId > 0) {
      id += 'H' + localId.toString(32); // "H + 32글자-이진문자열"
    }

    id+=':'; // 마지막 : 붙이기
  } else {
    // Use a lowercase r prefix for client-generated ids.
    const globalClientId = globalClientIdCounter++;
    id = ':' + identifierPrefix + 'r' + globalClientId.toString(32) + ':';
  }

  hook.memoizedState = id;
  return id;
}
```

useTransition

UI 변경을 가로막지 않고 상태를 업데이트

- ✨ 동시성 concurrency을 다룰 수 있는 새로운 훅
  - 랜더링이 무거운 작업에 가로막히는 현상을 개선
- 예시
  - 느린 랜더링 과정일 때 로딩 화면
  - 랜더링을 버리고 새로운 상태값으로 다시 랜더링
- 주의점
  - 내부에는 setState과 같은 상태 업데이트 함수 작업만!
  - startTransition 상태 업데이트는
    다른 동기 상태 업데이트로 실행이 지연될 수 있다.
  - 반드시 동기 함수만!
    작업을 지연시키는 startTransition 작업과
    비동기 실행은 불일치이기 때문

```tsx
import { useState, useTransition } from "react";

type Tab = "about" | "posts" | "contact";

export default function TabContainer() {
  const [isPending, startTransition] = useTransition(); // 인수x
  // isPending: 상태 업데이트가 진행 중인지 확인
  // startTransition: 긴급하지 않은 상태 업데이트로 간주한 set 함수를 인수로 받는 함수
  // 여러 개의 setter를 넣어줄 수도 있다.
  const [tab, setTab] = useState<Tab>("about");

  function selectTab(nextTab: Tab) {
    startTransition(() => {
      setTab(nextTab);
    });
  }

  return (
    <>
      {isPending ? (
        "로딩 중"
      ) : (
        <>
          {tab === "about" && <About />}
          {tab === "posts" && <Posts />}
          {tab === "contact" && <Contact />}
        </>
      )}
    </>
  );
}
```

useDeferredValue

UI 변경을 가로막지 않고 상태를 업데이트

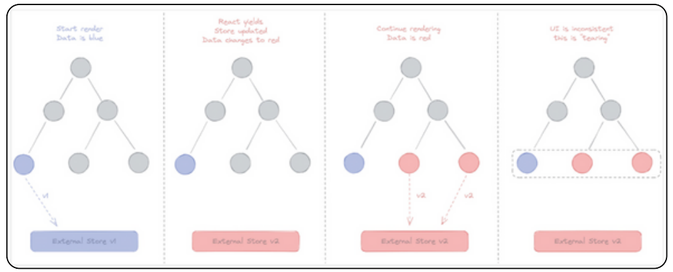
- useTransition과 useDeferredValue
  - useTransition: 상태 업데이트 코드에 접근
  - useDeferredValue: 값에만 접근 ex) props
- debounce는? 고정된 지연 시간이 필요
  useDeferredValue는?
  => 첫 번째 랜더링이 완료된 후
  지연된 랜더링을 수행
- 지연된 랜더링은 중단할 수 있고
  유저 인터렉션을 차단하지 않는다

```jsx
export default function Input() {
  const [text, setText] = useState(""); // 잦은 변경
  const deferredText = useDeferredValue(text); // 급하지 않음

  const list = useMemo(() => {
    const arr = Array.from({ length: deferredText.length }).map(
      (_) => deferredText
    );

    return (
      <ul>
        {arr.map((str, index) => (
          <li key={index}>{str}</li>
        ))}
      </ul>
    );
  }, [deferredText]); // deferredText를 의존하는 list 메모이제이션

  function handleChange(e: ChangeEvent<HTMLInputElement>) {
    setText(e.target.value);
  }

  return <><input value={text} onChange={handleChange}>{list}</input>;
}
```

useSyncExternalStore

외부 데이터 소스와 동시성 처리

- Tearing 현상
  - 하나의 state를 놓고
    서로 다른 값을 기준으로 랜더링되는 현상



- 랜더링을 중단, 미루기가 가능해지기 때문에
  동시성 이슈가 발생할 수 있다!
- 리액트 scope 외부의 데이터 예시
  - Global Varible
  - document.body, window.innerWidth, DOM
  - 외부 상태 관리 라이브러리

```tsx
import { useSyncExternalStore } from "react";

function subscribe(callback: (this: Window, ev: UIEvent) => void) {
  window.addEventListener("resize", callback);
  return () => {
    window.removeEventListener("resize", callback);
  };
}

function useWindowWidth() {
  return useSyncExternalStore(
    subscribe,
    () => window.innerWidth,
    () => 0 // 서버 사이드 렌더링 시 제공되는 기본값
  );
}

export default function App() {
  const windowSize = useWindowWidth();
  return (
    <>
      {windowSize} {window.innerWidth}
    </>
  );
}
```

useInsertionEffect

DOM 변경 작업 이전에 실행되는 Effect 훅

- CSS-in-js
  - styled-components가 사용하는 스타일을 모아서
    서버 사이드 랜더링 이전에 <style> 태그에 삽입하는 작업은
    모든 리액트 컴포넌트 랜더링에 영향을 미칠 수도 있는 매우 무거운 작업
- 서버 사이드에서 스타일 코드를 삽입할 수 있는 훅
- DOM이 실제로 변경되기 전에 동기적으로 실행됨
  - 브라우저가 레이아웃을 계산하기 전에 실행될 수 있게 한다

훅	실행 시점
useLayoutEffect	DOM 변경 작업이 끝난 이후에 실행
useInsertionEffect	DOM 변경 작업 이전에 실행된다
useEffect	컴포넌트 Commit 이후에 실행된다

react-dom/client

react-dom 변경점 : client

- 클라이언트에서 리액트 트리를 만들 때 사용하는 API 변경점
  - index.{tlj}sx에서 내용 변경
- createRoot

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

- hydrateRoot

서버 사이드 랜더링 애플리케이션에서
하이드레이션을 하기 위한 메서드

  - 프레임워크에 의존 => 수정할 일이 거의 없다.

```
// before
import ReactDOM from "react-dom";
import App from "App";

const container = document.getElementById("root");

ReactDOM.hydrate(<App />, container);

// 18
const root = ReactDOM.hydrateRoot(container, <App />);
```

- onRecoverableError
  - createRoot, hydrateRoot의 옵션
  - 과정에서 에러가 발생했을 때 발생하는 콜백 함수
    => reportError, console.error 또는 원하는 내용

react-dom/server

react-dom 변경점 : server

renderToPipeableStream

- 리액트 컴포넌트를 HTML로 랜더링하는 메서드, 스트림을 지원
  - HTML을 점진적으로 랜더링, 클라이언트는 중간에 script를 삽입하는 등의 작업을 할 수 있음
  - 서버에서 Suspence를 사용해, 빠르게 랜더링 필요한 부분 랜더링(비싼 연산 나중에)
- 첫 번째 로딩을 빠르게 수행하기
  - hydrateRoot를 호출해 서버에서는 HTML 랜더링 + 클라이언트의 리액트에는 이벤트만 추가
  - 기존 renderToNodeStream 문제점 -> 무조건 순서대로, 순서에 의존적 / 블락킹, 병목
- <Suspense />와 같은 코드 분할, 지연 랜더링을 서버사이드에서 사용
  - 실제로 프레임워크가 아니라 renderToPipableStream으로 서버사이드 랜더링을 만드는 경우 거의 없지만
    -> 사용하는 프레임워크에서 리액트 18을 사용하고 싶다면? 개념과 메서드 지원 여부 확인 필요

renderToReadableStream

- 웹 스트림을 사용하는 모던 엣지 런타임 환경에서 사용하는 메서드 (일반적x)
  - renderToPipeableStream: Node.js
  - renderToReadableStream: web stream

자동 배치 Automatic Batching

- 리액트가 여러 상태를 하나의 리랜더링으로 묶어서 성능을 향상시키는 방법
  - 사용자 액션이 한 번에 두 개 이상의 state을 동시에 업데이트 한다?
    - 하나의 리랜더링으로 묶어서 수행

- Promise, setTimeout과 같은 비동기 이벤트
  - 17의 경우? 두 번 리랜더링
  - 18에서는 자동 배치로 한 번 리랜더링(모든 업데이트가 배치 작업으로 최적화되도록 바뀜)
  - 그 외의 이벤트는 동일하게 1번

- 자동 배치를 안할 경우 > flushSync를 사용하면 된다.

문자열 ref 금지

```
<input type="text" ref="myInput" />; // 문자열을 ref로 사용함
console.log(this.refs.myInput); // deprecated
```

- 문자열 ref의 문제점, 문자열 ref 사용 금지
  - 여러 컴포넌트에 걸쳐 사용될 수 있으므로 충돌의 여지 있음
  - 단순 만자열로 존재하기 때문에, 어떤 ref에서 참조되고 있는 지 파악하기 어려움
  - 리액트가 계속해서 현재 랜더링되고 있는 컴포넌트의 ref 값을 추적하기 때문에 성능 이슈가 있음

findDOMNode 사용에 대한 경고

❌ 클래스 컴포넌트 인스턴스에서 실제 DOM 요소에 대한 참조를 가져올 수 있는 지금은 권장되지 않는 메서드

- 부모가 특정 자식만 별도로 랜더링하는 것이 가능했다. 리액트가 추구하는 트리 추상화 구조와 맞지 않음

구 Context API 사용 시 경고

❌ (엄격 모드) childContextType, getChildContext 사용 시 경고

예상치 못한 side-effects에 대한 검사

- (엄격 모드) 다음을 이중 호출
  - 클래스 컴포넌트의 constructor.render, shoudComponentUpdate.getDerivedStateFromProps
  - 클래스 컴포넌트의 setState의 첫 번째 인수
  - 함수 컴포넌트의 body
  - useState, useMemo, useReducer에 전달되는 함수
- 왜?
  - 함수형 프로그래밍 원칙에 따라 모든 컴포넌트는 항상 순수하다.
    - 정말 순수한 결과물을 내고 있는지 개발자에게 확인 시켜주기 위해 두 번 실행한다.
  - 입력 값이 변경되지 않으면, 항상 같은 결과물을 반환한다.
    - 즉, state, props, context가 변경되지 않으면? 항상 동일한 JSX를 반환해야 한다.

console

- 17에서는 두 번 log 안함
- 18에서는 두 번째 log는 글자색상 회색

https://react.dev/blog/2022/03/08/react-18-upgrade-guide#updates-to-strict-mode

stric mode's update

- 향후 리액트에서, 컴포넌트가 마운트 해제된 상태에서 내부의 상태값을 유지하는 기능을 제공할 예정 <- 큰 일
  - ex) 뒤로 가기 후 돌아왔을 때 이전 상태 유지
  - 컴포넌트가 처음 마운트될 때마다, 모든 컴포넌트를 자동으로 마운트 해제 및 재마운트해, 두 번째 마운트 이전 상태로 복원

```
* React mounts the component.
  * Layout effects are created.
  * Effect effects are created.
* React simulates unmounting the component.
  * Layout effects are destroyed.
  * Effects are destroyed.
* React simulates mounting the component with the previous state.
  * Layout effect setup code runs
  * Effect setup code runs
```

Suspense 기능 강화

- Suspense는?
  - 16.6에서 실험 버전으로 도입 / 컴포넌트 동적으로 가져올 수 있게 도와주는 기능
  - lazy와 Suspense는 한 쌍으로 사용됐고, 애플리케이션에서 상대적으로 중요하지 않은 컴포넌트 분할
    - 초기 렌더링 속도를 향상시킴

```
const DynamicSampleComponent = lazy(() => import("./SampleComponent"));

export default function App() {
  return (
    <Suspense fallback={<>...loading</>}>
      <DynamicSampleComponent />
    </Suspense>
  );
}
```

- 마운트 되기 직전에 effect가 빠르게 실행되는 문제 수정!
  => 이제 컴포넌트가 실제로 화면에 노출될 때 effect 실행

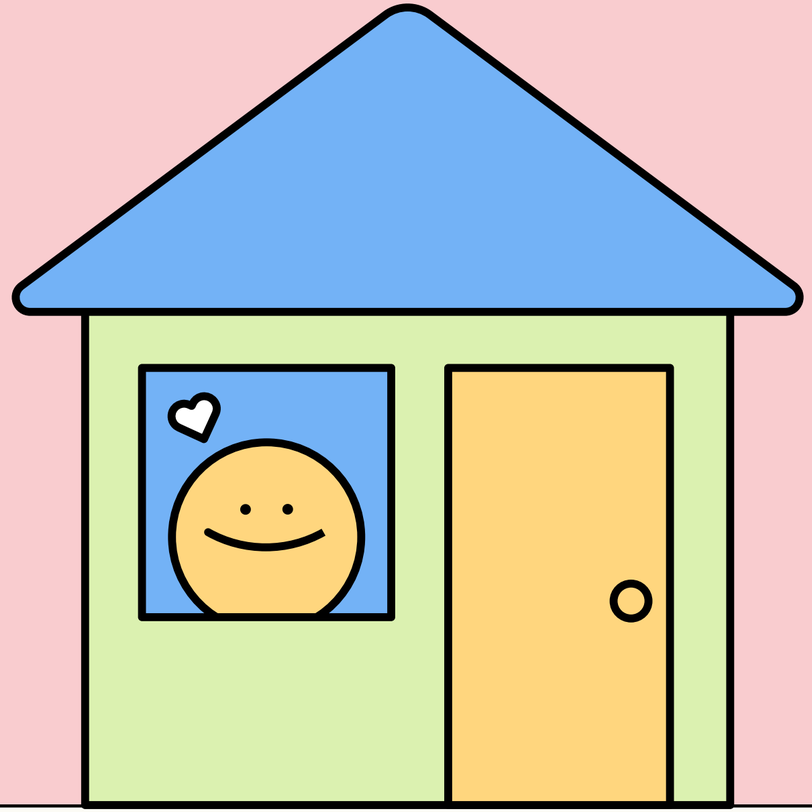
- Supense로 인해 컴포넌트가 보이거나 사라질 때: effect 정상적으로 실행
  이전에는 컴포넌트 스스로가 Suspense에 의해 현재 보여지고 있는지 숨겨져 있는지 알 수 있는 방법이 없었다.
  - 지금은? useLayoutEffect의 effect와 cleanUp이 정상적으로 실행된다.
    - ✅ effect (componentDidMount)
    - ✅ cleanUp (componentWillUnmount)
  - Suspense를 서버에서도 실행할 수 있음
  - Suspense에 쓰로틀링이 추가되었다. 중첩된 Suspense의 fallback이 있다면 자동으로 throttle 된다

- 18.0.0 더 살펴보기
  - https://github.com/facebook/react/releases/tag/v18.0.0
  - 17 버전업보다 많은 Breaking Changes 📚
- 리액트 18의 핵심은 동시성 랜더링
  - 일시중지, 랜더링 포기 등의 매커니즘을 도입
    - 랜더링 과정이 복잡해졌다.
    - 메인 스레드를 차단하지 않고 일관적인 UI 표시를 보장한다

😃

10장 리액트 17과 18의 변경사항 살펴보기

감사합니다



reference

모던 리액트 Deep Dive 10장 리액트 17과 18의 변경사항 살펴보기

[17] **점진적 업그레이드** https://legacy.reactjs.org/blog/2020/10/20/react-v17.html#gradual-upgrades

[18] **동시성** https://legacy.reactjs.org/blog/2022/03/29/react-v18.html#what-is-concurrent-react

Thank you!