useEffect

`useEffect` 는 외부 시스템과 컴포넌트를 동기화하는 React Hook입니다.

```
useEffect(setup, dependencies?)
```

- 레퍼런스
  - `useEffect(setup, dependencies?)`
- 사용방법
  - 외부 시스템과 연결
  - 커스텀 Hook을 Effect로 감싸기
  - React로 작성되지 않은 위젯 제어하기
  - Effect를 이용한 데이터 페칭
  - 반응형값 의존성 지정
  - Effect에서 이전 state를 기반으로 state 업데이트하기
  - 불필요한 객체 의존성 제거하기
  - 불필요한 함수 의존성 제거하기
  - Effect에서 최신 props와 state를 읽기
  - 서버와 클라이언트에서 다른 컨텐츠를 표시하기
- 트러블 슈팅
  - Effect가 컴포넌트 마운트 시 2번 동작합니다.
  - Effect가 매 리렌더링마다 실행됩니다.
  - Effect가 무한 반복됩니다.
  - 컴포넌트가 마운트 해제되지 않았음에도 정리 함수가 실행됩니다.
  - Effect가 시각적인 작업을 수행하며, 실행되기 전에 깜빡임이 보입니다.

레퍼런스

useEffect(setup, dependencies?)

컴포넌트의 최상위 레벨에서 `useEffect` 를 호출하여 Effect를 선언할 수 있습니다.

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

아래에서 더 많은 예시를 보세요.

매개변수

- `setup(설정)` : Effect의 로직이 포함된 함수입니다. 설정 함수는 선택적으로 *clean up(정리)* 함수를 반환할 수 있습니다. React는 컴포넌트가 DOM에 추가된 이후에 설정 함수를 실행합니다. 의존성의 변화에 따라 컴포넌트가 리렌더링이 되었을 경우, (설정 함수에 정리 함수를 추가했었다면) React는 이전 렌더링에 사용된 값으로 정리 함수를 실행한 후 새로운 값으로 설정 함수를 실행합니다. 컴포넌트가 DOM에서 제거된 경우에도 정리 함수를 실행합니다.

- `dependencies` **선택사항** : `설정` 함수의 코드 내부에서 참조되는 모든 반응형 값들이 포함된 배열로 구성됩니다. 반응형 값에는 props와 state, 모든 변수 및 컴포넌트 body에 직접적으로 선언된 함수들이 포함됩니다. 린터가 React 환경에 맞게 설정되어 있을 경우, 린터는 모든 반응형 값들이 의존성에 제대로 명시되어 있는지 검증할 것입니다. 의존성 배열은 항상 일정한 수의 항목을 가지고 있어야 하며 `[dep1, dep2, dep3]` 과 같이 작성되어야 합니다. React는 각각의 의존성들을 `Object.is` 비교법을 통해 이전 값과 비교합니다. 의존성을 생략할 경우,

Effect는 컴포넌트가 리렌더링될 때마다 실행됩니다. 인수에 의존성 배열을 추가했을 때, 빈 배열을 추가했을 때, 의존성을 추가하지 않았을 때의 차이를 확인해 보세요.

반환값

`useEffect`는 `undefined`를 반환합니다.

주의 사항

- `useEffect`는 Hook이므로 컴포넌트의 최상위 또는 커스텀 Hook에서만 호출할 수 있습니다. 반복문이나 조건문에서는 사용할 수 없습니다. 필요한 경우 새로운 컴포넌트를 추출하고 해당 컴포넌트로 state를 이동해서 사용할 수 있습니다.
- 외부 시스템과 컴포넌트를 동기화할 필요가 없는 경우, Effect를 선언할 필요가 없을 수 있습니다.
- Strict Mode를 사용할 경우, React는 실제 첫 번째 설정 함수가 실행되기 이전에 **개발 모드에만 한정하여 한 번의 추가적인 설정 + 정리 사이클을 실행합니다.** 이는 정리 로직이 설정 로직을 완벽히 "반영"하고 설정 로직이 수행하는 작업을 중단하거나 취소할 수 있는지를 확인하는 스트레스 테스트입니다. 이에 따라 문제가 생길 경우, 정리 함수를 구현하십시오.
- 만약 의존성이 객체이거나 컴포넌트 내부에 선언된 함수일 경우에는 Effect가 필요 이상으로 재실행될 수 있습니다. 이를 수정하려면 불필요한 객체 의존성이나 함수 의존성을 제거하세요. 또는 state 업데이트를 추출하거나 Effect 밖으로 비 반응형 로직을 빼낼 수 있습니다.
- Effect가 사용자 상호작용(클릭 등)에 의해 발생하지 않았다면, React는 일반적으로 **Effect를 실행하기 전에 브라우저가 업데이트된 화면을 먼저 렌더링하도록 합니다.** 만약 Effect가 시각적인 작업을 수행하고 (예: 툴팁의 위치 조정), 이에 따라 지연이 눈에 띄게 나타난다면 (예: 깜빡임 현상), `useEffect` 대신 `useLayoutEffect`를 사용하세요.
- Effect가 사용자 상호작용(클릭 등)으로 인해 발생한 경우, **React는 화면이 업데이트되어 브라우저가 화면을 그리기 전에 Effect를 실행할 수 있습니다.** 이것이 Effect의 결과를 이벤트 시스템이 관찰할 수 있도록 보장합니다. 이는 대개 예상대로 작동하지만, `alert()`와 같이 작업을 브라우저가 화면을 그린 후로 미뤄야 하는 경우 `setTimeout`을 활용할 수 있습니다. 자세한 내용은 reactwg/react-18/128을 참조하세요.
- Effect가 사용자 상호작용(클릭 등)에 의해 발생했더라도, **React는 때로 Effect 내부의 상태 업데이트를 처리하기 전에 브라우저가 화면을 다시 그리도록 허용할 수 있습니다.** 이는 대개 예상대로 작동하지만, 브라우저가 화면을 다시 그리지 않도록 막아야 하는 상황이라면 `useEffect` 대신 `useLayoutEffect`를 사용해야 합니다.
- Effect는 **client 환경에서만 동작합니다.** 서버 렌더링에서는 동작하지 않습니다.

사용방법

외부 시스템과 연결

몇몇 컴포넌트들은 페이지에 표시되는 동안 네트워크나 브라우저 API, 또는 서드파티 라이브러리와의 연결이 유지되어야 합니다. React에 제어되지 않는 이러한 시스템들을 *외부 시스템 (external)* 이라 부릅니다.

컴포넌트를 외부 시스템과 연결하려면 컴포넌트의 최상위 레벨에서 `useEffect` 를 호출해야 합니다.

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

`useEffect` 는 2개의 인수가 필요합니다.

1. 외부 시스템과 컴포넌트를 연결하는 설정 코드 가 포함된 *설정 함수*
   - 외부 시스템과의 연결을 해제하는 정리 코드 가 포함된 *정리 함수*를 반환할 수 있습니다.
2. 위 함수 내부에서 사용하는 컴포넌트에서 비롯된 반응형 값들을 포함하는 의존성 배열

React는 설정 함수와 정리 함수가 필요할 때마다 호출할 수 있으며, 이는 여러 번 호출될 수 있습니다.

1. 컴포넌트가 화면에 추가되었을 때 설정 코드 가 동작합니다 *(마운트 시)*.

2. 의존성 이 변경된 컴포넌트가 리렌더링 될 때마다 아래 동작을 수행합니다.
   - 먼저 정리 코드 가 오래된 props와 state와 함께 실행됩니다.
   - 이후, 설정 코드 가 새로운 props와 state와 함께 실행됩니다.
3. 컴포넌트가 화면에서 제거된 이후에 정리 코드 가 마지막으로 실행됩니다 *(마운트 해제 시)*.

위의 예시를 통해 순서를 설명해 보겠습니다.

위의 `ChatRoom` 컴포넌트가 화면에 추가되면 초기 `serverUrl` 과 `roomId` 를 이용해 채팅방과 연결될 것입니다. 리렌더링에 의해 `serverUrl` 또는 `roomId` 가 변경된다면 (예를 들어 사용자가 드롭다운 메뉴를 이용해 다른 채팅방을 선택할 경우) *Effect는 이전 채팅방과의 연결을 해제하고 다음 채팅방과 연결합니다.* `ChatRoom` 컴포넌트가 화면에서 제거된다면 Effect는 마지막 채팅방과 이뤄진 연결을 해제할 것입니다.

React는 **버그를 발견하기 위해 개발모드에서 설정 이 실행되기 전에 설정 과 정리 를 한 번 더 실행시킵니다.** 이는 스트레스 테스트의 하나로써 Effect의 로직이 정확하게 수행되고 있는지를 검증합니다. 만약 가시적인 이슈가 보인다면 정리 함수의 로직에 놓친 부분이 있는 것입니다. 정리 함수는 설정 함수의 어떠한 동작이라도 중지하거나 실행 취소를 할 수 있어야 하며, 사용자는 *설정* 함수가 한 번 호출될 때와 *설정* → *정리* → *설정* 순서로 호출될 때의 차이를 느낄 수 없어야 합니다.

각각의 Effect를 독립적인 프로세스로 작성하고 정확한 설정/정리 사이클을 고려하세요. 컴포넌트의 마운트, 업데이트, 마운트 해제 여부는 중요하지 않아야 합니다. 정리 로직이 설정 로직과 정확하게 "미러링"될 때, Effect는 설정과 정리를 필요한 만큼 견고하게 처리합니다.

중요합니다!

Effect는 (채팅 시스템과 같은) 외부 시스템과 컴포넌트가 동기화를 유지할 수 있도록 합니다. *외부 시스템*은 React에 의해 컨트롤되지 않는 모든 코드를 의미합니다. 예를 들어:

- `setInterval()` 에 의해 관리되는 타이머 또는 `clearInterval()` .
- `window.addEventListener()` 을 이용한 이벤트 구독 또는 `window.removeEventListener()` .
- `animation.start()` 와 같은 서드 파티 애니메이션 라이브러리 API 또는 `animation.reset()` .

만약 외부 시스템과 React를 연결할 필요가 없다면 Effect를 사용할 필요가 없을 수 있습니다.

외부 시스템과 연결 예시

1. 채팅 서버와 연결
2. 전역 브라우저 이벤트 감시하기
3. 애니메이션 동작시키기
4

예시 1 of 5: 채팅 서버와 연결

이 예시에서는 `ChatRoom` 컴포넌트의 Effect를 통해 `chat.js` 로 정의된 외부 시스템과 연결을 유지합니다. “Open chat”을 누르면 `ChatRoom` 컴포넌트가 나타납니다. 이 샌드박스는 개발 모드에서 동작하므로 추가적인 연결-연결해제 사이클이 동작합니다. 드롭다운 메뉴나 input을 이용해 `roomId` 또는 `serverUrl` 를 변경하고 어떻게 Effect가 chat을 재연결하는지 확인해 보세요. “Close chat”을 눌러 Effect가 마지막에 연결되었던 chat을 연결 해제하는 것도 확인해 보세요.

App.js chat.js 새로고침 Clear 포크

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
```

자세히 보기

다음 예시

커스텀 Hook을 Effect로 감싸기

Effect는 “탈출구” 입니다. “React 바깥으로 나가야 할 때”와 유즈케이스에 필요한 빌트인 솔루션이 없을 때 사용합니다. 만약 Effect를 자주 작성해야 한다면 컴포넌트가 의존하고 있는 공통적인 동작들을 커스텀 Hook으로 추출해야 한다는 신호일 수 있습니다.

예시로 아래의 `useChatRoom` 커스텀 Hook은 Effect의 로직을 조금 더 선언적인 API로 보일 수 있도록 숨겨줍니다.

```
function useChatRoom({ serverUrl, roomId }) {
  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
      roomId: roomId
    };
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]);
}
```

이제 이 커스텀 Hook을 어떤 컴포넌트에서도 이용할 수 있습니다.

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useChatRoom({
    roomId: roomId,
    serverUrl: serverUrl
  });
  // ...
```

또한 React 생태계에는 각종 목적에 맞는 훌륭한 커스텀 Hook들도 많이 존재합니다.

이 링크를 통해 커스텀 Hook에 대해 더 많이 공부해보세요.

커스텀 Hook에서 Effect를 활용하는 예시

1. 커스텀 `useChatRoom` Hook
2. 커스텀 `useWindowListener` Hook
3. 커스텀 `use`

예시 1 of 3: 커스텀 `useChatRoom` Hook

이 예시는 이전 예시 중 하나와 동일하지만 로직이 커스텀 Hook으로 추출되었습니다.

App.js | useChatRoom.js | chat.js — 새로고침 | Clear | 포크

```
import { useState } from 'react';
import { useChatRoom } from './useChatRoom.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useChatRoom({
    roomId: roomId,
    serverUrl: serverUrl
  });

  return (
```

자세히 보기

다음 예시

React로 작성되지 않은 위젯 제어하기

가끔은 컴포넌트의 prop 또는 state를 외부 시스템과 동기화해야할 때가 있습니다.

예를 들어 React 없이 작성된 서드 파티 지도 위젯이나 비디오 플레이어 컴포넌트가 있다면 이 컴포넌트의 state를 현재 React 컴포넌트의 state와 일치하도록 하기 위해 Effect를 사용할 수 있습니다. 이 Effect는 `map-widget.js` 에 정의된 `MapWidget` 클래스의 인스턴스를 생성합니다. `Map` 컴포넌트의 `zoomLevel` prop을 변경할 때, Effect는 해당 클래스 인스턴스의 `setZoom()` 을 호출하여 동기화를 유지합니다.

App.js Map.js map-widget.js 새로고침 Clear 포크

```
import { useRef, useEffect } from 'react';
import { MapWidget } from './map-widget.js';
```

```
export default function Map({ zoomLevel }) {
  const containerRef = useRef(null);
  const mapRef = useRef(null);

  useEffect(() => {
    if (mapRef.current === null) {
      mapRef.current = new MapWidget(containerRef.current);
    }
```

자세히 보기

이 예시에서는 정리 함수가 필요하지 않습니다. 이는 `MapWidget` 클래스가 클래스에 전달된 DOM 노드만 관리하기 때문입니다. `Map` 컴포넌트가 트리에서 제거된 후, 브라우저의 자바스크립트 엔진에 의해 DOM 노드와 `MapWidget` 클래스 인스턴스 모두가 자동으로 가비지 컬렉션에 의해 정리됩니다.

Effect를 이용한 데이터 페칭

You can use an Effect to fetch data for your component. Note that if you use a framework, using your framework's data fetching mechanism will be a lot more efficient than writing Effects manually.

만약 직접 Effect를 작성하여 데이터를 페칭하고 싶다면, 코드는 다음과 같을 수 있습니다.

```
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);

  useEffect(() => {
    let ignore = false;
    setBio(null);
    fetchBio(person).then(result => {
      if (!ignore) {
        setBio(result);
      }
    });
    return () => {
      ignore = true;
    };
  }, [person]);

  // ...
```

`ignore` 변수의 초기값이 `false`로 설정되고 정리 함수 동작 중에 `true`로 설정되는 것에 주목하세요. 이 로직은 코드가 "경쟁 상태(race conditions)"에 빠지지 않도록 보장해 줍니다. 네트워크 요청을 보낸 순서와 응답을 받는 순서가 다르게 동작할 수 있기 때문에 이러한 처리가 필요합니다.

App.js

새로고침 Clear 포크

```
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);
  useEffect(() => {
    let ignore = false;
    setBio(null);
    fetchBio(person).then(result => {
```

```
        if (!ignore) {
          setBio(result);
```

자세히 보기

또한 `async` / `await` 구문을 사용하여 코드를 다시 작성할 수 있지만 여전히 정리 함수를 제공해야 합니다.

App.js

새로고침 Clear 포크

```
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);
  useEffect(() => {
    async function startFetching() {
      setBio(null);
      const result = await fetchBio(person);
      if (!ignore) {
        setBio(result);
```

자세히 보기

Effect에서 직접 데이터 페칭 로직을 작성하면 나중에 캐싱 기능이나 서버 렌더링과 같은 최적화를 추가하기 어려워집니다. 자체 제작된 커스텀 Hook이나 커뮤니티에 의해 유지보수되는 Hook을 사용하는 편이 더 간단합니다.

자세히 살펴보기

Effect에서 데이터를 페칭하는 좋은 대안은 무엇인가요?

자세히 보기

반응형값 의존성 지정

Effect의 의존성을 "선택"할 수 없다는 점에 유의하세요. Effect 코드에서 사용하는 모든 반응형 값 은 의존성으로 선언되어야 합니다. Effect의 의존성 배열은 코드에 의해 결정됩니다.

```
function ChatRoom({ roomId }) { // 이것은 반응형 값입니다
  const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // 이것도 반응
```

```
  useEffect(() => {
    const connection = createConnection( serverUrl , roomId ); // 이 Effect는 이 반응형 값
    connection.connect();
    return () => connection.disconnect();
  }, [ serverUrl , roomId ]); // ✅ 그래서 이 값들을 Effect의 의존성으로 지정해야 합니다
  // ...
}
```

`serverUrl` 또는 `roomId` 가 변경될 때마다 Effect는 새로운 값을 이용해 채팅을 다시 연결할 것입니다.

반응형 값 에는 props와 컴포넌트 내부에 선언된 모든 변수나 함수들이 포함됩니다. `roomId` 와 `serverUrl` 은 반응형 값이므로 이들을 의존성에서 제거하면 안 됩니다. 이들을 누락했을 때 린터가 React 환경에 맞게 설정되어 있었다면 린터는 이것을 수정해야 하는 실수로 표시합니다.

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // 🔴 React Hook useEffect has missing dependencies: 'roomId' and 'serverUrl
  // ...
}
```

의존성을 제거하려면 그것이 의존성이 되지 않아야 함을 린터에 증명해야 합니다. 예를 들어, `serverUrl` 을 컴포넌트 밖으로 이동하여 그것이 반응적이지 않고 리렌더링될 때 변경되지 않을 것임을 증명할 수 있습니다.

```
const serverUrl = 'https://localhost:1234'; // 더 이상 반응형 값이 아님

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
```

```
    return () => connection.disconnect();
  }, [roomId]); // ✅ 모든 의존성이 선언됨
  // ...
}
```

이제 `serverUrl` 은 반응형 값이 아니며 (리렌더링될 때 변경되지 않을 것이므로), 의존성에 추가할 필요가 없습니다. **Effect의 코드가 어떤 반응형 값도 사용하지 않는다면 그 의존성 목록은 비어있어야 합니다. (`[]`)**

```
const serverUrl = 'https://localhost:1234'; // 더 이상 반응형 값이 아님
const roomId = 'music'; // 더 이상 반응형 값이 아님

function ChatRoom() {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // ✅ 모든 의존성이 선언됨
  // ...
}
```

의존성이 비어있는 Effect는 컴포넌트의 props나 state가 변경되도 다시 실행되지 않습니다.

주의하세요!

기존의 코드 베이스에서 아래와 같이 린터를 억제하고 있는 일부 Effect가 있을 수 있습니다.

```
useEffect(() => {
  // ...
  // 🔴 Avoid suppressing the linter like this:
  // eslint-ignore-next-line react-hooks/exhaustive-deps
}, []);
```

의존성이 코드와 일치하지 않을 때 버그가 도입될 위험이 큽니다. 린터를 억제함으로써 Effect가 의존하는 값에 대해 React가 '거짓말'을 하게 됩니다. 린터를 속이는 대신 이러한 값들이 불필요하다는 것을 증명하세요.

반응형 값을 의존성으로 추가하는 예시

1. 의존성 배열 전달
2. 빈 의존성 배열 전달
3. 의존성 배열을 전달하지 않았을 때

예시 1 of 3: 의존성 배열 전달

의존성을 명시하면 Effect는 **초기 렌더링 후 *그리고* 의존성 값 변경과 함께 리렌더링이 된 후 동작합니다.**

```
useEffect(() => {
  // ...
}, [a, b]); // a나 b가 다르면 다시 실행됨
```

아래 예시에서는 `serverUrl` 와 `roomId` 은 반응형 값이므로 둘 다 의존성으로 지정해야 합니다. 결과적으로 드롭다운에서 다른 방을 선택하거나 서버 URL 입력을 편집하면 채팅이 다시 연결됩니다. 그러나 `message` 는 Effect에서 사용되지 않으므로(의존성이 아니므로), 메세지를 편집해도 대화가 다시 연결되지 않습니다.

App.js chat.js 새로고침 Clear 포크

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');
  const [message, setMessage] = useState('');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
```

```
        connection.disconnect();
```

자세히 보기

다음 예시

Effect에서 이전 state를 기반으로 state 업데이트하기

Effect에서 이전 state를 기반으로 state를 업데이트하려면 문제가 발생할 수 있습니다.

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(count + 1); // 초마다 카운터를 증가시키고 싶습니다...
    }, 1000)
    return () => clearInterval(intervalId);
  }, [count]); // 🚩 ... 하지만 'count'를 의존성으로 명시하면 항상 인터벌이 초기화됩니다.
  // ...
```

`count` 가 반응형 값이므로 반드시 의존성 배열에 추가해야 합니다. 그러나 `count` 가 변경되는 것은 Effect가 정리된 후 다시 설정되는 것을 야기하므로 `count` 는 계속 증가할 것입니다. 이상적이지 않은 방식입니다.

이러한 현상을 방지하려면 `c => c + 1` state 변경함수를 `setCount` 에 추가하세요.

App.js

다운로드 새로고침 Clear 포크

```
import { useState, useEffect } from 'react';

export default function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(c => c + 1); // ✅ State 업데이터를 전달
    }, 1000);
    return () => clearInterval(intervalId);
  }, []); // ✅ 이제 count는 의존성이 아닙니다
```

`c => c + 1`을 `count + 1` 대신 전달하고 있으므로, Effect는 더 이상 `count`에 의존하지 않습니다. 이 수정으로 인해 `count`가 변경될 때마다 Effect가 정리 및 설정을 다시 실행할 필요가 없게 됩니다.

불필요한 객체 의존성 제거하기

Effect가 렌더링 중에 생성된 객체나 함수에 의존하는 경우, 너무 자주 실행될 수 있습니다. 예를 들어 이 Effect는 매 렌더링 후에 다시 연결됩니다. 이는 렌더링마다 `options` 객체가 다르기 때문입니다.

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const options = { // 🚩 이 객체는 재 렌더링 될 때마다 새로 생성됩니다
    serverUrl: serverUrl,
    roomId: roomId
  };

  useEffect(() => {
    const connection = createConnection(options); // 객체가 Effect 안에서 사용됩니다
    connection.connect();
    return () => connection.disconnect();
  }, [options]); // 🚩 결과적으로, 의존성이 재 렌더링 때마다 다릅니다
  // ...
```

렌더링 중에 생성된 객체를 의존성으로 사용하는 것을 피하세요. 대신 객체를 Effect 내에서 생성하세요.

App.js chat.js 새로고침 Clear 포크

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

const serverUrl = 'https://localhost:1234';
```

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
```

자세히 보기

이제 `options` 객체를 Effect 내에서 생성하면, Effect 자체는 roomId 문자열에만 의존합니다.

이 수정으로 입력란에 텍스트를 입력하더라도 채팅이 다시 연결되지 않습니다. 객체와는 달리 `roomId` 와 같은 문자열은 다른 값으로 설정하지 않는 한 변경되지 않습니다. 의존성 제거에 관한 자세한 내용은 여기를 참고하세요.

불필요한 함수 의존성 제거하기

Effect가 렌더링 중에 생성된 객체나 함수에 의존하는 경우, 너무 자주 실행될 수 있습니다. 예를 들어 이 Effect는 매 렌더링 후에 다시 연결됩니다. 이는 렌더링마다 `createOptions` 함수가 다르기 때문입니다.

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  function createOptions() { // 🚩 이 함수는 재 렌더링 될 때마다 새로 생성됩니다
    return {
      serverUrl: serverUrl,
      roomId: roomId
    };
  }

  useEffect(() => {
    const options = createOptions(); // 함수가 Effect 안에서 사용됩니다
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // 🚩 결과적으로, 의존성이 재 렌더링 때마다 다릅니다
  // ...
```

리렌더링마다 함수를 처음부터 생성하는 것 그 자체로는 문제가 되지 않고, 이를 최적화할 필요는 없습니다. 그러나 이것을 Effect의 의존성으로 사용하는 경우 Effect가 리렌더링 후마다 다시 실행되게 합니다.

렌더링 중에 생성된 함수를 의존성으로 사용하는 것을 피하세요. 대신 Effect 내에서 함수를 선언하세요.

App.js chat.js 새로고침 Clear 포크

```
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    function createOptions() {
      return {
        serverUrl: serverUrl,
```

자세히 보기

`createOptions` 함수가 Effect 내부에 선언되었으므로, Effect 자체는 `roomId` 문자열에만 의존합니다. 이 수정을 통해 입력란에 입력하는 것만으로 채팅이 다시 연결되지 않습니다. `roomId`와 같은 문자열은 다른 값으로 설정하지 않는 한 변경되지 않기 때문입니다. 의존성 제거에 대한 자세한 내용은 여기를 참고하세요.

Effect에서 최신 props와 state를 읽기

By default, when you read a reactive value from an Effect, you have to add it as a dependency. This ensures that your Effect “reacts” to every change of that value. For most dependencies, that’s the behavior you want.

그러나 때로는 Effect에서 최신 props와 state를 ‘반응’하지 않고 읽고 싶을 수 있습니다. 예를 들어 페이지 방문마다 쇼핑 카트에 담긴 항목 수를 기록하고 싶다고 가정해 보겠습니다.

```
function Page({ url, shoppingCart }) {
  useEffect(() => {
    logVisit(url, shoppingCart.length);
  }, [url, shoppingCart]); // ✅ 모든 의존성이 선언됨
```

```
  // ...
}
```

What if you want to log a new page visit after every `url` change, but *not* if only the `shoppingCart` changes? You can't exclude `shoppingCart` from dependencies without breaking the reactivity rules. However, you can express that you *don't want* a piece of code to "react" to changes even though it is called from inside an Effect. Declare an *Effect Event* with the `useEffectEvent` Hook, and move the code reading `shoppingCart` inside of it:

```
function Page({ url, shoppingCart }) {
  const onVisit = useEffectEvent(visitedUrl => {
    logVisit(visitedUrl, shoppingCart.length)
  });

  useEffect(() => {
    onVisit(url);
  }, [url]); // ✅ 모든 의존성이 선언됨
  // ...
}
```

Effect 이벤트는 반응적이지 않으며 Effect의 의존성에서 배제되어야 합니다. Effect 이벤트에는 비 반응형 코드(Effect 이벤트 로직은 최신 props와 state를 읽을 수 있음)를 배치할 수 있습니다. `onVisit` 내의 `shoppingCart` 를 읽음으로써 `shoppingCart` 의 변경으로 인한 Effect의 재실행을 방지합니다.

Effect 이벤트가 어떻게 반응형 및 비 반응형 코드를 분리하는 데 도움이 되는지에 대한 자세한 내용은 여기를 읽어보세요.

서버와 클라이언트에서 다른 컨텐츠를 표시하기

If your app uses server rendering (either directly or via a framework), your component will render in two different environments. On the server, it will render to produce the initial HTML. On the client, React will run the rendering code again so that it can attach your

event handlers to that HTML. This is why, for hydration to work, your initial render output must be identical on the client and the server.

드물게 클라이언트에서 다른 내용을 표시해야 할 수 있습니다. 예를 들어 앱이 `localStorage` 에서 일부 데이터를 읽는 경우, 이를 서버에서 구현할 수 없습니다. 다음은 이것을 구현하는 방법입니다.

```
function MyComponent() {
  const [didMount, setDidMount] = useState(false);

  useEffect(() => {
    setDidMount(true);
  }, []);

  if (didMount) {
    // ... 클라이언트 전용 JSX 반환 ...
  }  else {
    // ... 초기 JSX 반환 ...
  }
}
```

앱이 로딩 중인 동안 사용자는 초기 렌더링 출력을 볼 것입니다. 그다음 로딩 및 hydration이 완료되면 Effect가 실행되어 `didMount` 를 `true` 로 설정하면서 다시 렌더링이 동작합니다. 이로써 클라이언트 전용 렌더링 출력으로 전환됩니다. Effect는 서버에서 실행되지 않으므로 초기 서버 렌더링 중의 `didMount` 는 `false` 가 됩니다.

이 패턴은 적절히 사용해야 합니다. 느린 연결 환경을 가진 사용자는 초기 렌더링 화면을 상당한 시간 동안 볼 것이므로 컴포넌트의 모양을 급변시키지 않는 것이 좋습니다. 많은 경우에는 CSS를 사용하여 조건부로 다양한 것들을 표시하는 방법으로 대처할 수 있습니다.

트러블 슈팅

Effect가 컴포넌트 마운트 시 2번 동작합니다.

개발 환경에서 Strict Mode가 활성화되면 React는 실제 설정 이전에 설정과 정리를 한번 더 실행합니다.

이것은 Effect의 로직이 올바르게 구현되었는지 확인하는 스트레스 테스트입니다. 이에 따라 눈에 띄는 문제가 발생한다면 정리 함수에 어떤 로직이 누락되었을 수 있습니다. 정리 함수는 설정 함수가 수행한 것을 중지하거나 되돌릴 수 있어야 합니다. 일반적인 지침으로는 사용자가 설정이 한번 호출되는 것(배포 환경과 같이)과 설정 → 정리 → 설정 순서로 호출되는 것을 구별할 수 없어야 한다는 것입니다.

이것이 버그를 찾는 데 어떻게 도움이 되며, 로직을 어떻게 수정하는지에 자세히 알아보려면 여기를 읽어보세요.

Effect가 매 리렌더링마다 실행됩니다.

먼저 의존성 배열에 값을 추가했는지 확인해 보세요.

```
useEffect(() => {
  // ...
}); // 🚩 의존성 배열이 없음. 재 렌더링 될 때마다 재실행됨!
```

의존성 배열을 명시했음에도 Effect가 여전히 반복해서 실행된다면 의존성이 렌더링마다 다르기 때문입니다.

이 문제를 해결하기 위해 콘솔에 의존성을 수동으로 기록하는 방법으로 디버깅할 수 있습니다.

```
  useEffect(() => {
    // ..
  }, [serverUrl, roomId]);

  console.log([serverUrl, roomId]);
```

그다음 콘솔에서 기록된 다른 렌더링 배열을 마우스 오른쪽 버튼으로 클릭하고 두 배열 모두에 대해 전역 변수로 저장을 선택할 수 있습니다. 첫 번째 요소가 `temp1` 이고 두 번째 요소가 `temp2` 라고 가정하면 브라우저 콘솔을 사용하여 양쪽 배열의 각 의존성이 동일한지 확인할 수 있습니다.

```
Object.is(temp1[0], temp2[0]); // 첫 번째 의존성이 배열 간에 동일한가요?
Object.is(temp1[1], temp2[1]); // 두 번째 의존성이 배열 간에 동일한가요?
```

```
Object.is(temp1[2], temp2[2]); // ... 나머지 모든 의존성도 확인합니다  ...
```

렌더링마다 다른 의존성을 찾아냈다면 일반적으로 다음 중 하나의 방법으로 수정할 수 있습니다.

- Effect에서 이전 state를 기반으로 state 업데이트하기
- 불필요한 객체 의존성 제거하기
- 불필요한 함수 의존성 제거하기
- Effect에서 최신 props와 state를 읽기

최후의 수단으로 (이러한 방법들이 도움이 되지 않은 경우), `useMemo` 나 `useCallback` (함수의 경우)을 이용할 수 있습니다.

Effect가 무한 반복됩니다.

Effect가 무한 반복되려면 다음 두 가지 조건이 충족되어야 합니다.

- Effect에서 state를 업데이트함.
- 변경된 state가 리렌더링을 유발하며, 이에 따라 Effect의 종속성이 변경됨.

문제를 해결하기 전에 Effect가 외부 시스템(DOM, 네트워크, 서드파티 위젯 등)에 연결되어 있는지 스스로 자문해보세요. Effect에서 왜 state를 변경했나요? 변경된 state가 외부 시스템과 동기화됐나요? 또는 Effect를 통해 애플리케이션의 데이터 흐름을 관리하려고 하는 건가요?

외부 시스템이 없다면 Effect를 제거해서 로직을 단순화할 수 있는지 고려해보세요.

만약 실제로 어떤 외부 시스템과 동기화 중이라면 Effect가 state를 언제 어떤 조건에서 업데이트해야 하는지에 대해 고려해 보세요. 컴포넌트의 시각적 출력에 영향을 주는 state가 변했나요? 렌더링에 사용되지 않는 데이터를 추적해야 한다면 리렌더링을 야기하지 않는 ref가 더 적합할 수 있습니다. Effect가 필요 이상으로 state를 업데이트하는지(리렌더링을 야기하지 않도록) 확인해보세요.

마지막으로 Effect가 제대로 된 시점에 state를 업데이트했지만 여전히 무한 반복되는 경우, 해당 state의 업데이트가 Effect의 종속성의 변경을 야기했을 수 있습니다. 종속성 변경을 디버깅하는 방법을 읽어보세요.

컴포넌트가 마운트 해제되지 않았음에도 정리 함수가 실행됩니다.

정리 함수는 마운트 해제 시 뿐만 아니라 변경된 종속성으로 인한 모든 리렌더링 전에 실행됩니다. 또한 개발 환경에서는 React가 컴포넌트가 마운트된 직후에 한 번 더 설정과 정리를 실행합니다.

설정 코드와 상응하는 정리 코드가 없다면 보통은 코드에 문제가 있을 가능성이 높습니다.

```
useEffect(() => {
  // 🔴 피하세요: 상응하는 설정 로직이 없는 정리 로직
  return () => {
    doSomething();
  };
}, []);
```

정리 로직은 설정 로직과 '대칭'이어야 하며 설정이 수행한 것을 중지하거나 되돌릴 수 있어야 합니다.

```
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
```

Effect의 생명주기와 컴포넌트의 생명주기가 어떻게 다른지 확인해 보세요.

Effect가 시각적인 작업을 수행하며, 실행되기 전에 깜빡임이 보입니다.

Effect가 `브라우저가 화면을 그리는 것` 을 차단해야 하는 경우 `useEffect` 를 `useLayoutEffect` 로 대체하세요. **이것은 대부분의 Effect에는 필요하지 않습니다.** 브라우저 페인팅 이전에 Effect를 실행하는 것이 중요한 경우에만 필요합니다. 예를 들어 사용자가 보기 전에 툴팁의 위치를 측정하고 지정해야 하는 경우가 있습니다.

이전
useDeferredValue

다음
useEffectEvent