useCallback

`useCallback`은 리렌더링 간에 함수 정의를 캐싱해 주는 **React Hook**입니다.

```
const cachedFn = useCallback(fn, dependencies)
```

중요합니다!

React 컴파일러는 값과 함수를 자동으로 메모이제이션하므로 `useCallback`을 수동으로 사용할 일이 줄어듭니다. 컴파일러를 사용해 메모이제이션을 자동으로 처리할 수 있습니다.

레퍼런스

useCallback(fn, dependencies)

리렌더링 간에 함수 정의를 캐싱하려면 컴포넌트의 최상단에서 `useCallback` 을 호출하세요.

```
import { useCallback } from 'react';

export default function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
```

아래에서 더 많은 예시를 확인해보세요.

매개변수

- `fn` : 캐싱할 함숫값입니다. 이 함수는 어떤 인자나 반환값도 가질 수 있습니다. React는 첫 렌더링에서 이 함수를 반환합니다. (호출하는 것이 아닙니다!) 다음 렌더링에서 `dependencies` 값이 이전과 같다면 React는 같은 함수를 다시 반환합니다. 반대로 `dependencies` 값이 변경되었다면 이번 렌더링에서 전달한 함수를 반환하고 나중에 재사용할 수 있도록 이를 저장합니다. React는 함수를 호출하지 않습니다. 이 함수는 호출 여부와 호출 시점을 개발자가 결정할 수 있도록 반환됩니다.

- `dependencies` : `fn` 내에서 참조되는 모든 반응형 값의 목록입니다. 반응형 값은 props와 state, 그리고 컴포넌트 안에서 직접 선언된 모든 변수와 함수를 포함합니다. 린터가 React를 위한 설정으로 구성되어 있다면 모든 반응형 값이 의존성으로 올바르게 명시되어 있는지 검증합니다. 의존성 목록은 항목 수가 일정해야 하며 `[dep1, dep2, dep3]` 처럼 인라인으로 작성해야 합니다. React는 `Object.is` 비교 알고리즘을 이용해 각 의존성을 이전 값과 비교합니다.

반환값

최초 렌더링에서는 `useCallback` 은 전달한 `fn` 함수를 그대로 반환합니다.

후속 렌더링에서는 이전 렌더링에서 이미 저장해 두었던 `fn` 함수를 반환하거나 (의존성이 변하지 않았을 때), 현재 렌더링 중에 전달한 `fn` 함수를 그대로 반환합니다.

주의 사항

- `useCallback`은 Hook이므로, **컴포넌트의 최상위 레벨** 또는 커스텀 Hook에서만 호출할 수 있습니다. 반복문이나 조건문 내에서 호출할 수 없습니다. 이 작업이 필요하다면 새로운 컴포넌트로 분리해서 state를 새 컴포넌트로 옮기세요.
- React는 **특별한 이유가 없는 한 캐시 된 함수를 삭제하지 않습니다.** 예를 들어 개발 환경에서는 컴포넌트 파일을 편집할 때 React가 캐시를 삭제합니다. 개발 환경과 프로덕션 환경 모두에서, 초기 마운트 중에 컴포넌트가 일시 중단되면 React는 캐시를 삭제합니다. 앞으로 React는 캐시 삭제를 활용하는 더 많은 기능을 추가할 수 있습니다. 예를 들어, React에 가상화된 목록에 대한 빌트인 지원이 추가한다면, 가상화된 테이블 뷰포트에서 스크롤 밖의 항목에 대해 캐시를 삭제하는것이 적절할 것 입니다. 이는 `useCallback`을 성능 최적화 방법으로 의존하는 경우에 개발자의 예상과 일치해야 합니다. 그렇지 않다면 state 변수 나 ref가 더 적절할 수 있습니다.

용법

컴포넌트의 리렌더링 건너뛰기

렌더링 성능을 최적화할 때 자식 컴포넌트에 넘기는 함수를 캐싱할 필요가 있습니다. 먼저 이 작업을 수행하는 방법에 대한 구문을 살펴본 다음 어떤 경우에 유용한지 알아보겠습니다.

컴포넌트의 리렌더링 간에 함수를 캐싱하려면 함수 정의를 `useCallback` Hook으로 감싸세요.

```
import { useCallback } from 'react';

function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
  // ...
```

`useCallback`에게 두 가지를 전달해야 합니다

1. 리렌더링 간에 캐싱할 함수 정의

2. 함수에서 사용되는 컴포넌트 내부의 모든 값을 포함하고 있는 의존성 목록

최초 렌더링에서 `useCallback`으로부터 반환되는 함수는 호출시에 전달할 함수입니다.

이어지는 렌더링에서 React는 의존성을 이전 렌더링에서 전달한 의존성과 비교합니다. 의존성 중 하나라도 변한 값이 없다면(`Object.is`로 비교), `useCallback`은 전과 똑같은 함수를 반환합니다. 그렇지 않으면 `useCallback`은 *이번* 렌더링에서 전달한 함수를 반환합니다.

다시 말하면, `useCallback`은 의존성이 변하기 전까지 리렌더링 간에 함수를 캐싱합니다.

이 기능이 언제 유용한지 예시를 통해 살펴보겠습니다.

`handleSubmit` 함수를 `ProductPage`에서 `ShippingForm` 컴포넌트로 전달한다고 가정해 봅시다.

```
function ProductPage({ productId, referrer, theme }) {
  // ...
  return (
    <div className={theme}>
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
```

`theme` prop을 토글 하면 앱이 잠시 멈춘다는 것을 알게 되었는데, JSX에서 `<ShippingForm />`을 제거하면 앱이 빨라진 것처럼 느껴집니다. 이는 `<ShippingForm />` 컴포넌트의 최적화를 시도해 볼 가치가 있다는 것을 나타냅니다.

기본적으로, 컴포넌트가 리렌더링할 때 React는 해당 컴포넌트의 모든 자식을 재귀적으로 리렌더링합니다. 이는 `ProductPage`가 다른 `theme` 값으로 리렌더링 할 때, `ShippingForm` 컴포넌트 **또한** 리렌더링 하는 이유입니다. 이것은 리렌더링에 많은 계산을 요구하지 않는 컴포넌트에서는 괜찮습니다. 하지만 리렌더링이 느린 것을 확인한 경우, `ShippingForm`을 `memo`로 감싸면 마지막 렌더링과 동일한 props일 때 리렌더링을 건너뛰도록 할 수 있습니다.

```
import { memo } from 'react';

const ShippingForm = memo(function ShippingForm({ onSubmit }) {
  // ...
```

```
});
```

이렇게 변경한 `ShippingForm` 은 모든 props가 마지막 렌더링과 *같다면* 리렌더링을 건너뜁니다. 여기가 함수 캐싱이 중요해지는 순간입니다! `useCallback` 없이 `handleSubmit` 을 정의했다고 가정해 봅시다.

```
function ProductPage({ productId, referrer, theme }) {
  // theme이 바뀔때마다 다른 함수가 될 것입니다...
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }

  return (
    <div className={theme}>
      {/* ... 그래서 ShippingForm의 props는 같은 값이 아니므로 매번 리렌더링 할 것입니다.*/}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

자바스크립트에서 `function () {}` 나 `() => {}` 은 항상 *다른* 함수를 생성합니다. 이는 `{}` 객체 리터럴이 항상 새로운 객체를 생성하는 방식과 유사합니다. 보통의 경우에는 문제가 되지 않지만, 여기서는 `ShippingForm` props는 절대 같아질 수 없고 `memo` 최적화는 동작하지 않을 것이라는 걸 의미합니다. 여기서 `useCallback` 이 유용하게 사용됩니다.

```
function ProductPage({ productId, referrer, theme }) {
  // React에게 리렌더링 간에 함수를 캐싱하도록 요청합니다...
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ...이 의존성이 변경되지 않는 한...

  return (
```

```
      <div className={theme}>
        {/* ...ShippingForm은 같은 props를 받게 되고 리렌더링을 건너뛸 수 있습니다.*/}
        <ShippingForm onSubmit={handleSubmit} />
      </div>
    );
}
```

`handleSubmit`을 `useCallback`으로 감쌈으로써 리렌더링 간에 이것이 (의존성이 변경되기 전까지는) 같은 함수라는 것을 보장합니다. 특별한 이유가 없다면 함수를 꼭 `useCallback`으로 감쌀 필요는 없습니다. 이 예시에서의 이유는 'memo'로 감싼 컴포넌트에 전달하기 때문에 해당 함수가 리렌더링을 건너뛸 수 있기 때문입니다. `useCallback`이 필요한 다른 이유는 이 페이지의 뒷부분에서 설명하겠습니다.

중요합니다!

`useCallback`은 성능 최적화를 위한 용도로만 사용해야 합니다. 만약 코드가 `useCallback` 없이 작동하지 않는다면 먼저 근본적인 문제를 찾아 해결해야 합니다. 그 다음에 `useCallback`을 다시 추가할 수 있습니다.

자세히 살펴보기

`useCallback`과 `useMemo`는 어떤 연관이 있나요?

자세히 보기

자세히 살펴보기

항상 `useCallback`을 사용해야 할까요?

자세히 보기

useCallback과 함수를 직접 선언하는 것의 차이점

1. `useCallback` 과 `memo` 로 리렌더링 건너뛰기 2. 컴포넌트를 항상 리렌더링하기

예시 1 of 2: `useCallback` 과 `memo` 로 리렌더링 건너뛰기

이 예시에서 `ShippingForm` 컴포넌트는 **인위적으로 느리게 만들었기 때문에** 렌더링하는 React 컴포넌트가 실제로 느릴 때 어떤 일이 일어나는 지 볼 수 있습니다. 카운터를 증가시키고 테마를 토글 해보세요.

카운터를 증가시키면 느려진 `ShippingForm` 이 리렌더링하기 때문에 느리다고 느껴집니다. 이는 예상된 동작입니다. 카운터가 변경되었으므로 사용자의 새로운 선택을 화면에 반영해야 하기 때문입니다.

다음으로 테마를 토글 해보세요. **`useCallback` 을 `memo` 와 함께 사용한 덕분에, 인위적인 지연에도 불구하고 빠릅니다!** `ShippingForm` 은 `handleSubmit` 함수가 변하지 않았기 때문에 리렌더링을 건너뛰었습니다. `productId` 와 `referrer` (`useCallback` 의 의존성) 모두 마지막 렌더링으로부터 변하지 않았기 때문에 `handleSubmit` 함수도 변하지 않았습니다.

App.js ProductPage.js ShippingForm.js 새로고침 Clear 포크

```
import { useCallback } from 'react';
import ShippingForm from './ShippingForm.js';

export default function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);

  return (
```

자세히 보기

다음 예시

Memoized 콜백에서 상태 업데이트하기

때때로 memoized 콜백에서 이전 상태를 기반으로 상태를 업데이트해야 할 때가 있습니다.

`handleAddTodo` 함수는 `todos` 로부터 다음 할 일을 계산하기 때문에 이를 의존성으로 명시했습니다.

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos([...todos, newTodo]);
  }, [todos]);
  // ...
```

보통은 memoized 함수가 가능한 한 적은 의존성을 갖는 것이 좋습니다. 다음 상태를 계산하기 위해 어떤 상태를 읽는 경우, 업데이트 함수를 대신 넘겨줌으로써 의존성을 제거할 수 있습니다.

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos(todos => [...todos, newTodo]);
  }, []); // ✅ todos 의존성은 필요하지 않습니다.
  // ...
```

여기서 `todos`를 의존성으로 만들고 안에서 값을 읽는 대신, React에 *어떻게* 상태를 업데이트할지에 대한 지침을 넘겨줍니다. 업데이트 함수에 대해 더 알아보세요.

Effect가 너무 자주 실행되는 것을 방지하기

가끔 Effect 안에서 함수를 호출해야 할 수도 있습니다.

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

  function createOptions() {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    // ...
```

이것은 문제를 발생시킵니다. 모든 반응형 값은 Effect의 의존성으로 선언되어야 합니다. 하지만 `createOptions` 를 의존성으로 선언하면 Effect가 채팅방과 계속 재연결되는 문제가 발생합니다.

```
  useEffect(() => {
    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // 🔴 문제점: 이 의존성은 매 렌더링마다 변경됩니다.
  // ...
```

이를 해결하기 위해, Effect에서 호출하려는 함수를 `useCallback` 으로 감쌀 수 있습니다.

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

  const createOptions = useCallback(() => {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }, [roomId]); // ✅ roomId가 변경될 때만 변경됩니다.

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // ✅ createOptions가 변경될 때만 변경됩니다.
  // ...
```

이는 리렌더링 간에 `roomId` 가 같다면 `createOptions` 함수는 같다는 것을 보장합니다. **하지만, 함수 의존성을 제거하는 것이 더 좋습니다.** 함수를 Effect *안으로* 이동시키세요.

```
function ChatRoom({ roomId }) {
```

```
  const [message, setMessage] = useState('');

  useEffect(() => {
    function createOptions() { // ✅ useCallback이나 함수 의존성이 필요하지 않습니다.
      return {
        serverUrl: 'https://localhost:1234',
        roomId: roomId
      };
    }

    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✅ roomId가 변경될 때만 변경됩니다.
  // ...
```

이제 코드는 더 간단해졌고 `useCallback`은 필요하지 않습니다. Effect의 의존성 제거에 대해 더 알아보세요.

커스텀 Hook 최적화하기

커스텀 Hook을 작성하는 경우, 반환하는 모든 함수를 `useCallback`으로 감싸는 것이 좋습니다.

```
function useRouter() {
  const { dispatch } = useContext(RouterStateContext);

  const navigate = useCallback((url) => {
    dispatch({ type: 'navigate', url });
  }, [dispatch]);

  const goBack = useCallback(() => {
    dispatch({ type: 'back' });
  }, [dispatch]);

  return {
    navigate,
    goBack,
  };
```

이렇게 하면 Hook을 사용하는 컴포넌트가 필요할 때 가지고 있는 코드를 최적화할 수 있습니다.

문제 해결

컴포넌트가 렌더링 될 때마다 `useCallback`이 다른 함수를 반환합니다.

두 번째 인수로 의존성 배열을 지정했는지 확인하세요!

의존성 배열을 까먹으면 `useCallback`은 매번 새로운 함수를 반환합니다.

```
function ProductPage({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }); // 🔴 매번 새로운 함수를 반환합니다: 의존성 배열 없음
  // ...
```

다음은 두 번째 인수로 의존성 배열을 넘겨주도록 수정한 코드입니다.

```
function ProductPage({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ✅ 불필요하게 새로운 함수를 반환하지 않습니다.
  // ...
```

이것이 도움이 되지 않는다면 의존성 중 적어도 하나가 이전 렌더링과 다른 것이 문제입니다. 의존성을 콘솔에 직접 기록하여 이 문제를 디버깅할 수 있습니다.

```
  const handleSubmit = useCallback((orderDetails) => {
    // ..
  }, [productId, referrer]);

  console.log([productId, referrer]);
```

그런 다음 콘솔에서 서로 다른 렌더링의 배열을 마우스 오른쪽 클릭 후 “전역 변수로 저장”을 선택할 수 있습니다. 첫 번째 것이 `temp1`, 두 번째 것이 `temp2`로 저장됐다면, 브라우저 콘솔을 통해 각 의존성이 두 배열에서 같은지 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  ...
```

어떤 의존성이 memoization을 깨고 있는지 찾았다면 이를 제거하거나 memoization하는 방법을 찾으세요.

반복문에서 각 항목마다 `useCallback`을 호출하고 싶지만, 이는 허용되지 않습니다.

`Chart` 컴포넌트가 `memo`로 감싸져 있다고 생각해 봅시다. `ReportList` 컴포넌트가 렌더링 될 때마다, 모든 `Chart` 항목이 리렌더링 하는 것을 막고 싶습니다. 하지만 반복문에서 `useCallback`을 호출할 수 없습니다.

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item => {
        // 🔴 이렇게 반복문 안에서 useCallback을 호출할 수 없습니다.
        const handleClick = useCallback(() => {
          sendReport(item)
        }, [item]);

        return (
```

```
          <figure key={item.id}>
            <Chart onClick={handleClick} />
          </figure>
        );
      })}
    </article>
  );
}
```

대신 개별 항목을 컴포넌트로 분리하고, 거기에 `useCallback` 을 넣으세요.

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item =>
        <Report key={item.id} item={item} />
      )}
    </article>
  );
}

function Report({ item }) {
  // ✅ useCallback을 최상위 레벨에서 호출하세요
  const handleClick = useCallback(() => {
    sendReport(item)
  }, [item]);

  return (
    <figure>
      <Chart onClick={handleClick} />
    </figure>
  );
}
```

대안으로 마지막 스니펫에서 `useCallback` 을 제거하고 대신 `Report` 자체를 `memo` 로 감싸도 됩니다. `item` prop이 변경되지 않으면 `Report` 는 리렌더링하지 않기 때문에 `Chart` 도 리렌더링을 건너뜁니다.

```
function ReportList({ items }) {
```

```
  // ...
}

const Report = memo(function Report({ item }) {
  function handleClick() {
    sendReport(item);
  }

  return (
    <figure>
      <Chart onClick={handleClick} />
    </figure>
  );
});
```


이전
useActionState

다음
useContext