useTransition

`useTransition`은 UI의 일부를 백그라운드에서 렌더링 할 수 있도록 해주는 React Hook입니다.

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

레퍼런스

useTransition()

컴포넌트의 최상위 수준에서 `useTransition` 을 호출하여 일부 state 업데이트를 Transition 으로 표시합니다.

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

function TabContainer() {
  const [isPending, startTransition] = useTransition();
  // ...
}
```

아래에서 더 많은 예시를 확인하세요.

매개변수

`useTransition` 은 어떤 매개변수도 받지 않습니다.

반환값

`useTransition` 은 정확히 두 개의 항목이 있는 배열을 반환합니다.

1. `isPending` 플래그는 대기 중인 Transition 이 있는지 알려줍니다.
2. `startTransition` 함수는 상태 업데이트를 Transition 으로 표시할 수 있게 해주는 함수입니다.

---

startTransition(action)

`useTransition` 이 반환하는 `startTransition` 함수를 사용하면 업데이트를 Transition으로 표시할 수 있습니다.

```
function TabContainer() {
  const [isPending, startTransition] = useTransition();
  const [tab, setTab] = useState('about');

  function selectTab(nextTab) {
```

```
    startTransition(() => {
      setTab(nextTab);
    });
  }
  // ...
}
```

중요합니다!

startTransition 내에서 호출되는 함수를 “Actions”이라고 합니다.

`startTransition` 에 전달된 함수를 “Action”이라고 합니다. 따라서 시작 트랜지션 내에서 호출되는 모든 콜백(예: 콜백 프로퍼티)의 이름은 `action` 이거나 “Action” 접미사를 포함해야 합니다.

```
function SubmitButton({ submitAction }) {
  const [isPending, startTransition] = useTransition();

  return (
    <button
      disabled={isPending}
      onClick={() => {
        startTransition(async () => {
          await submitAction();
        });
      }}
    >
      Submit
    </button>
  );
}
```

매개변수

- `action` : 하나 이상의 `set` 함수를 호출하여 일부 상태를 업데이트하는 함수입니다. React는 매개변수 없이 즉시 `action` 을 호출하고 `action` 함수 호출 중에 동기적으로 예약된 모든 상태 업데이트를 Transitions으로 표시합니다. `action` 에서 await된 비동기 호출은 Transition에 포함되지만, 현재로서는 `await` 이후의 `set` 함수 호출을 추가적인 `startTransition` 으로 감싸야 합니다(문제 해결 참조). Transitions으로 표시된 상태 업데이트는 non-blocking 방식으로 처리되며, 불필요한 로딩 표시가 나타나지 않습니다.

반환값

`startTransition` 은 아무것도 반환하지 않습니다.

주의 사항

- `useTransition` 은 Hook이므로 컴포넌트나 커스텀 Hook 내부에서만 호출할 수 있습니다. 다른 곳(예시: 데이터 라이브러리)에서 Transition 을 시작해야 하는 경우, 독립형 `startTransition` 을 호출하세요.
- 해당 state의 `set` 함수에 액세스할 수 있는 경우에만 업데이트를 Transition 으로 래핑할 수 있습니다. 일부 prop이나 커스텀 Hook 값에 대한 응답으로 Transition 을 시작하려면 `useDeferredValue` 를 사용해 보세요.
- `startTransition` 에 전달하는 함수는 동기식이어야 합니다. React는 이 함수를 즉시 실행하여 실행하는 동안 발생하는 모든 state 업데이트를 Transition 으로 표시합니다. 나중에 더 많은 state 업데이트를 수행하려고 하면(예시: timeout), Transition 으로 표시되지 않습니다.
- `startTransition` 에 전달하는 함수는 즉시 호출되며, 실행 중 발생하는 모든 상태 업데이트를 Transition으로 표시합니다. 예를 들어 `setTimeout` 내에서 상태를 업데이트하려고 하면, 해당 업데이트는 Transition으로 표시되지 않습니다.
- 비동기 요청 이후의 상태 업데이트를 전환으로 표시하려면, 반드시 또 다른 `startTransition` 으로 감싸야 합니다. 이는 알려진 제한 사항으로 향후 수정될 예정입니다(문제 해결 참조).
- `startTransition` 함수는 안정된 식별성(stable identity)을 가지므로 Effect 의존성에서 생략되는 경우가 많습니다. 하지만 포함해도 Effect가 실행되지는 않습니다. linter가 의존성을 생략해도 오류를 발생시키지 않는다면 생략해도 안전합니다. 자세한 내용은 Effect 의존성 제거에 대한 문서를 참고하세요.
- Transition으로 표시된 state 업데이트는 다른 state 업데이트에 의해 중단됩니다. 예를 들어 Transition 내에서 차트 컴포넌트를 업데이트한 다음 차트가 다시 렌더링 되는 도중에 입력을

시작하면 React는 입력 업데이트를 처리한 후 차트 컴포넌트에서 렌더링 작업을 다시 시작합니다.

- Transition 업데이트는 텍스트 입력을 제어하는 데 사용할 수 없습니다.
- 진행 중인 Transition 이 여러 개 있는 경우, React는 현재 Transition 을 함께 일괄 처리합니다. 이는 향후 릴리즈에서 제거될 가능성이 높은 제한 사항입니다.

사용법

Actions으로 non-blocking 업데이트 수행

컴포넌트 상단에서 `useTransition` 을 호출하여 Actions을 생성하고, 대기 상태에 접근하세요.

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

function CheckoutForm() {
  const [isPending, startTransition] = useTransition();
  // ...
}
```

`useTransition` 은 정확히 두 개의 항목이 있는 배열을 반환합니다.

1. `isPending 플래그` 는 대기 중인 Transition 이 있는지 알려줍니다.
2. `startTransition 함수` 는 상태 업데이트를 Transition으로 표시할 수 있게 해주는 함수입니다.

Transition을 시작하려면 다음과 같이 `startTransition` 에 함수를 전달합니다.

```
import {useState, useTransition} from 'react';
import {updateQuantity} from './api';

function CheckoutForm() {
  const [isPending, startTransition] = useTransition();
  const [quantity, setQuantity] = useState(1);

  function onSubmit(newQuantity) {
    startTransition(async function () {
```

```
      const savedQuantity = await updateQuantity(newQuantity);
      startTransition(() => {
        setQuantity(savedQuantity);
      });
    });
  }
  // ...
}
```

The function passed to `startTransition` is called the “Action”. You can update state and (optionally) perform side effects within an Action, and the work will be done in the background without blocking user interactions on the page. A Transition can include multiple Actions, and while a Transition is in progress, your UI stays responsive. For example, if the user clicks a tab but then changes their mind and clicks another tab, the second click will be immediately handled without waiting for the first update to finish.

진행 중인 Transition에 대해 사용자에게 피드백을 제공하기 위해 `isPending` 상태는 `startTransition` 을 처음 호출할 때 `true` 로 전환되며, 모든 Action이 완료되어 최종 상태가 사용자에게 표시될 때까지 `true` 상태를 유지합니다. Transition은 Action 내의 사이드 이펙트가 완료되도록 보장하여 원치 않는 로딩 표시기가 표시되지 않도록 합니다. 또한, Transition이 진행 중일 때 `useOptimistic` 을 사용하여 즉각적인 피드백을 제공할 수 있습니다.

Action과 일반 이벤트 처리의 차이점

1. Action에서 수량 업데이트
2. Action 없이 수량 업데이트

예시 1 of 2: Action에서 수량 업데이트

이 예시에서 `updateQuantity` 함수는 카트에 있는 품목의 수량을 업데이트하기 위해 서버에 요청하는 시뮬레이션을 수행합니다. 이 함수는 요청을 완료하는 데 최소 1초가 소요되도록 *인위적으로 속도가 늦춰져 있습니다.*

수량을 빠르게 여러 번 업데이트하면, 요청이 진행 중인 동안에는 “Total” 상태가 대기 중으로 표시됩니다. 그리고 최종 요청이 완료된 후에만 “Total”이 업데이트됩니다. 업데이트가 Action 내에서 발생하기 때문에, 요청이 진행 중인 동안에도 “quantity”은 계속해서 업데이트될 수 있습니다.

App.js Item.js Total.js api.js 새로고침 Clear 포크

```
import { useState, useTransition } from "react";
import { updateQuantity } from "./api";
import Item from "./Item";
import Total from "./Total";

export default function App({}) {
  const [quantity, setQuantity] = useState(1);
  const [isPending, startTransition] = useTransition();

  const updateQuantityAction = async newQuantity => {
    // transition의 보류 중인 상태에 액세스하려면,
    // startTransition을 다시 호출하세요.
```

자세히 보기

이 예시는 Actions의 작동 방식을 보여주기 위한 기본 예시이지만, 순서대로 완료되는 요청은 처리하지 않습니다. 수량을 여러 번 업데이트하는 경우 이전 요청이 완료된 후 나중에 요청이 완료되어 수량이 순서대로 업데이트되지 않을 수 있습니다. 이는 알려진 제한 사항으로 향후 수정될 예정입니다(문제 해결 참조).

일반적인 사용 사례를 위해 React는 다음과 같은 내장 추상화 기능을 제공합니다.

- useActionState

- `<form>` actions
- Server Functions

이러한 솔루션은 요청 순서를 자동으로 처리합니다. Transitions를 사용하여 custom Hook 또는 라이브러리를 구축하여 비동기 상태 전환을 관리하는 경우, 요청 순서를 더욱 세밀하게 제어할 수 있지만, 직접 처리해야 합니다.

다음 예시

컴포넌트에서 Action 프로퍼티를 노출하기

컴포넌트에서 `action` 프로퍼티를 노출시켜 부모 컴포넌트에서 Action을 호출할 수 있습니다.

예를 들어, 이 `TabButton` 컴포넌트는 `onClick` 에서 실행될 로직이 `action` prop으로 감싸져있습니다.

```
export default function TabButton({ action, children, isActive }) {
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  return (
    <button onClick={() => {
      startTransition(async () => {
        // await the action that's passed in.
        // This allows it to be either sync or async.
        await action();
      });
    }}>
      {children}
    </button>
  );
}
```

부모 컴포넌트가 `action` 내부에서 상태를 업데이트하기 때문에, 해당 상태 업데이트는 Transition으로 표시됩니다. 그렇기 때문에 “Posts”을 클릭한 후 즉시 “Contact”를 클릭해도 사용자 상호작용이 차단되지 않습니다.

App.js TabButton.js AboutTab.js PostsTab.js Cc 새로고침 Clear 포크

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

export default function TabButton({ action, children, isActive }) {
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  if (isPending) {
    return <b className="pending">{children}</b>;
  }
  return (
    <button onClick={async () => {
```

자세히 보기

중요합니다!

When exposing an `action` prop from a component, you should `await` it inside the transition.

이렇게 하면 `action` 콜백이 동기적이든 비동기적이든 상관없이 작동할 수 있으며, `action` 내부의 `await` 을 추가적인 `startTransition` 으로 감쌀 필요가 없습니다.

대기 상태를 시각적으로 표현하기

`useTransition` 이 반환하는 `isPending` boolean 값을 사용하여 transition이 진행 중임을 사용자에게 표시할 수 있습니다. 예를 들어 탭 버튼은 특별한 “pending” 시각적 상태를 가질 수 있습니다.

```
function TabButton({ action, children, isActive }) {
  const [isPending, startTransition] = useTransition();
  // ...
  if (isPending) {
    return <b className="pending">{children}</b>;
  }
  // ...
```

이제 탭 버튼 자체가 바로 업데이트되므로 “Posts”을 클릭하는 반응이 더 빨라진 것을 확인할 수 있습니다.

App.js | TabButton.js | AboutTab.js | PostsTab.js | Cc | 새로고침 | Clear | 포크

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

export default function TabButton({ action, children, isActive }) {
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  if (isPending) {
    return <b className="pending">{children}</b>;
```

```
  }
  return (
```

자세히 보기

원치 않는 로딩 표시기 방지

이 예시에서 `PostsTab` 컴포넌트는 use를 사용하여 데이터를 가져옵니다. “Posts” 탭을 클릭하면 `PostsTab` 컴포넌트가 *suspend* 되어 가장 가까운 로딩 Fallback이 나타납니다.

App.js TabButton.js 새로고침 Clear 포크

```
import { Suspense, useState } from 'react';
import TabButton from './TabButton.js';
import AboutTab from './AboutTab.js';
import PostsTab from './PostsTab.js';
import ContactTab from './ContactTab.js';

export default function TabContainer() {
  const [tab, setTab] = useState('about');
  return (
    <Suspense fallback={<h1>🌀 Loading...</h1>}>
```

```
      <TabButton
```

자세히 보기

로딩 표시기를 표시하기 위해 전체 탭 컨테이너를 숨기면 사용자 경험이 어색해집니다. `useTransition`을 `TabButton`에 추가하면 탭 버튼 내부에 대기 중인 상태를 표시할 수 있습니다.

”Posts”을 클릭하면 더 이상 전체 탭 컨테이너가 스피너로 바뀌지 않습니다.

App.js TabButton.js 새로고침 Clear 포크

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

export default function TabButton({ action, children, isActive }) {
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  if (isPending) {
    return <b className="pending">{children}</b>;
  }
  return (
    <button onClick={() => {
```

자세히 보기

Suspense에서 Transition 을 사용하는 방법에 대해 자세히 알아보세요.

중요합니다!

Transition은 *이미 표시된* 콘텐츠(예시: 탭 컨테이너)를 숨기지 않을 만큼만 “대기”합니다. 만약 Posts 탭에 중첩된 `<Suspense>` 경계가 있는 경우 Transition 은 이를 “대기”하지 않습니다.

Suspense-enabled 라우터 구축

React 프레임워크나 라우터를 구축하는 경우 페이지 탐색을 Transition 으로 표시하는 것이 좋습니다.

```
function Router() {
```

```
const [page, setPage] = useState('/');
const [isPending, startTransition] = useTransition();

function navigate(url) {
  startTransition(() => {
    setPage(url);
  });
}
// ...
```

세 가지 이유로 이 방법을 권장합니다.

- Transition은 중단할 수 있으므로 사용자는 리렌더링이 완료될 때까지 기다릴 필요 없이 바로 클릭할 수 있습니다.
- Transition은 원치 않는 로딩 표시기를 방지하므로 사용자가 탐색 시 갑작스러운 이동을 방지할 수 있습니다.
- Transition은 모든 보류 중인 작업을 대기하므로 사용자는 사이드 이펙트가 완료된 후에 새로운 페이지를 볼 수 있습니다.

다음은 navigation에 Transitions를 사용하는 간단한 라우터 예시입니다.

App.js 새로고침 Clear 포크

```
import { Suspense, useState, useTransition } from 'react';
import IndexPage from './IndexPage.js';
import ArtistPage from './ArtistPage.js';
import Layout from './Layout.js';

export default function App() {
  return (
    <Suspense fallback={<BigSpinner />}>
      <Router />
    </Suspense>
  );
}
```

자세히 보기

중요합니다!

Suspense-enabled 라우터는 기본적으로 탐색 업데이트를 Transition 으로 래핑할 것으로 예상됩니다.

Error boundary로 사용자에게 오류 표시하기

startTransition에 전달된 함수에서 오류가 발생하면 error boundary를 사용하여 사용자에게 오류를 표시할 수 있습니다. error boundary를 사용하려면 useTransition을 호출하는 컴포넌트를 error boundary로 감싸면 됩니다. startTransition에 전달된 함수에서 오류가 발생하면 error boundary의 Fallback이 표시됩니다.

AddCommentConta

새로고침 Clear 포크

```
import { useTransition } from "react";
import { ErrorBoundary } from "react-error-boundary";

export function AddCommentContainer() {
  return (
    <ErrorBoundary fallback={<p>⚠️Something went wrong</p>}>
      <AddCommentButton />
    </ErrorBoundary>
```

```
    );
  }
```

자세히 보기

Troubleshooting

Transition에서 입력 업데이트가 작동하지 않습니다

입력을 제어하는 state 변수에는 Transition 을 사용할 수 없습니다.

```
const [text, setText] = useState('');
// ...
function handleChange(e) {
  // ❌ 제어된 입력 state에 Transition 을 사용할 수 없습니다.
  startTransition(() => {
    setText(e.target.value);
  });
}
// ...
```

이는 Transition 이 non-blocking이지만, 변경 이벤트에 대한 응답으로 입력을 업데이트하는 것은 동기적으로 이루어져야 하기 때문입니다. 입력에 대한 응답으로 Transition 을 실행하려면 두 가지 옵션이 있습니다.

1. 두 개의 개별 state 변수를 선언할 수 있습니다. 하나는 입력 state(항상 동기적으로 업데이트됨) 용이고 다른 하나는 Transition 시 업데이트할 state입니다. 이를 통해 동기 state를 사용하여 입력을 제어하고 (입력보다 "지연"되는) Transition state 변수를 나머지 렌더링 로직에 전달할 수 있습니다.
2. 또는 state 변수가 하나 있고 실제 값보다 "지연"되는 `useDeferredValue` 를 추가할 수 있습니다. 그러면 non-blocking 리렌더링이 새로운 값을 자동으로 "따라잡기" 위해 트리거됩니다.

React가 state 업데이트를 transition으로 처리하지 않습니다

state 업데이트를 transition으로 래핑할 때는 `startTransition` 호출 *도중*에 발생해야 합니다.

```
startTransition(() => {
  // ✅ startTransition 호출 *도중* state 설정
  setPage('/about');
});
```

`startTransition` 에 전달하는 함수는 동기식이어야 합니다. You can't mark an update as a Transition like this:

```
startTransition(() => {
  // ❌ startTransition 호출 *후에* state 설정
  setTimeout(() => {
    setPage('/about');
  }, 1000);
});
```

대신 다음과 같이 할 수 있습니다.

```
setTimeout(() => {
  startTransition(() => {
    // ✅ startTransition 호출 *도중* state 설정
    setPage('/about');
  });
}, 1000);
```

React는 `await` 이후의 상태 업데이트를 Transition으로 처리하지 않습니다.

`startTransition` 함수 내부에서 `await`를 사용할 경우, `await` 이후에 발생하는 상태 업데이트는 Transition으로 처리되지 않습니다. 각 `await` 이후에 발생하는 상태 업데이트를 별도의 `startTransition` 호출로 감싸야 합니다.

```
startTransition(async () => {
  await someAsyncFunction();
  // ❌ await 이후에 startTransition을 사용하지 않음
  setPage('/about');
});
```

하지만 이 방법이 대신 동작합니다.

```
startTransition(async () => {
  await someAsyncFunction();
  // ✅ await *이후에* startTransition을 사용
  startTransition(() => {
    setPage('/about');
  });
});
```

이는 JavaScript의 한계로 인해 React가 AsyncContext의 범위를 잃기 때문입니다. 향후 AsyncContext가 지원되면 이러한 제한 사항은 해결될 것입니다.

컴포넌트 외부에서 `useTransition`을 호출하고 싶습니다

Hook이기 때문에 컴포넌트 외부에서 `useTransition`을 호출할 수 없습니다. 이 경우 대신 독립형 `startTransition` 메서드를 사용하세요. 동일한 방식으로 작동하지만 `isPending` 표시기를 제공하지 않습니다.

`startTransition`에 전달한 함수는 즉시 실행됩니다

이 코드를 실행하면 1, 2, 3이 출력됩니다.

```
console.log(1);
startTransition(() => {
  console.log(2);
  setPage('/about');
});
console.log(3);
```

1, 2, 3을 출력할 것으로 예상됩니다. `startTransition`에 전달한 함수는 지연되지 않습니다. 브라우저 `setTimeout`과 달리 나중에 콜백을 실행하지 않습니다. React는 함수를 즉시 실행하지만, *함수가 실행되는 동안* 예약된 모든 상태 업데이트는 Transition으로 표시됩니다. 아래와 같이 작동한다고 상상하면 됩니다.

```
// React 작동 방식의 간소화된 버전

let isInsideTransition = false;

function startTransition(scope) {
  isInsideTransition = true;
  scope();
  isInsideTransition = false;
}

function setState() {
  if (isInsideTransition) {
    // ... Transition state 업데이트 예약 ...
  } else {
```

```
    // ... 긴급 state 업데이트 예약 ...
  }
}
```

Transitions에서 상태 업데이트가 순서대로 이루어지지 않아요

`startTransition` 내부에서 `await`를 사용하면 상태 업데이트가 순서대로 발생하지 않을 수 있습니다.

In this example, the `updateQuantity` function simulates a request to the server to update the item's quantity in the cart. This function *artificially returns every other request after the previous* to simulate race conditions for network requests.

수량을 한 번 업데이트한 후, 빠르게 여러 번 업데이트를 시도해 보세요. 그러면 잘못된 총합이 표시될 수 있습니다

App.js Item.js Total.js api.js 새로고침 Clear 포크

```
import { useState, useTransition } from "react";
import { updateQuantity } from "./api";
import Item from "./Item";
import Total from "./Total";

export default function App({}) {
  const [quantity, setQuantity] = useState(1);
  const [isPending, startTransition] = useTransition();
  // 실제 수량을 별도의 state에 저장하여 불일치를 표시합니다.
  const [clientQuantity, setClientQuantity] = useState(1);

  const updateQuantityAction = newQuantity => {
```

자세히 보기

여러 번 클릭하면 먼저 보낸 요청이 나중에 보낸 요청보다 늦게 처리될 수 있습니다. 이런 경우 React는 현재 의도한 순서를 알 수 있는 방법이 없습니다. 이는 업데이트가 비동기적으로 예약되고, React가 비동기 경계를 거쳐 순서에 대한 컨텍스트를 잃기 때문입니다.

이것은 예상된 동작입니다. Transition 내에서의 액션은 실행 순서를 보장하지 않기 때문입니다. 일반적인 사용 사례에서는 React가 `useActionState`나 `<form>` actions과 같은 더 높은 수준의 추상화를 제공하여 순서를 처리해 줍니다. 고급 사용 사례에서는 이 문제를 처리하기 위해 자체적인 큐잉(queuing) 및 취소 로직을 구현해야 합니다.

Example of `useActionState` handling execution order:

App.js Item.js Total.js api.js 새로고침 Clear 포크

```
import { useState, useActionState } from "react";
import { updateQuantity } from "./api";
import Item from "./Item";
import Total from "./Total";

export default function App({}) {
  // Store the actual quantity in separate state to show the mismatch.
  const [clientQuantity, setClientQuantity] = useState(1);
  const [quantity, updateQuantityAction, isPending] = useActionState(
    async (prevState, payload) => {
      setClientQuantity(payload);
      const savedQuantity = await updateQuantity(payload);
```

자세히 보기

이전
useSyncExternalStore

다음
컴포넌트

감사의 말

약관