useDeferredValue

`useDeferredValue`는 일부 UI 업데이트를 지연시킬 수 있는 React Hook입니다.

```
const deferredValue = useDeferredValue(value)
```

레퍼런스

`useDeferredValue(value, initialValue?)`

컴포넌트의 최상위 레벨에서 `useDeferredValue`를 호출하여 지연된 버전의 값을 가져옵니다.

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

function SearchPage() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  // ...
}
```

아래에서 더 많은 예시를 확인하세요.

매개변수

- `value` : 지연시키려는 값입니다. 모든 타입을 가질 수 있습니다.
- **선택사항** `initialValue` : 컴포넌트 초기 렌더링 시 사용할 값입니다. 이 옵션을 생략하면 초기 렌더링 동안 `useDeferredValue` 는 값을 지연시키지 않습니다. 이는 대신 렌더링할 `value` 의 이전 버전이 없기 때문입니다.

반환값

- `currentValue` : 초기 렌더링 중 반환된 '지연된 값'은 사용자가 제공한 값과 같습니다. 업데이트가 발생하면 React는 먼저 이전 값으로 리렌더링을 시도(반환값이 이전 값과 일치하도록)하고, 그 다음 백그라운드에서 다시 새 값으로 리렌더링을 시도(반환값이 업데이트된 새 값과 일치하도록)합니다.

주의 사항

- Transition 내에서 업데이트할 때 `useDeferredValue` 는 항상 새로운 `value` 를 반환하며 지연된 렌더링을 생성하지 않습니다. 이미 업데이트가 지연되었기 때문입니다.

- `useDeferredValue` 에 전달하는 값은 문자열 및 숫자와 같은 원시값이거나, 컴포넌트의 외부에서 생성된 객체여야 합니다. 렌더링 중에 새 객체를 생성하고 즉시 `useDeferredValue` 에 전달하면 렌더링할 때마다 값이 달라져 불필요한 백그라운드 리렌더링이 발생할 수 있습니다.

- `useDeferredValue` 가 현재 렌더링(여전히 이전 값을 사용하는 경우) 외에 다른 값 (`Object.is` 로 비교)을 받으면 백그라운드에서 새 값으로 리렌더링하도록 예약합니다. `value` 에 대한 또 다른 업데이트가 있으면 백그라운드 리렌더링은 중단될 수 있습니다. React는 백그라운드 리렌더링을 처음부터 다시 시작할 것입니다. 예를 들어 차트가 리렌더링 가능한 지연된 값을 받는 속도보다 사용자가 Input에 값을 입력하는 속도가 더 빠른 경우, 차트는 사용자가 입력을 멈춘 후에만 리렌더링됩니다.

- `useDeferredValue` 는 `<Suspense>` 와 통합됩니다. 새로운 값으로 인한 백그라운드 업데이트로 인해 UI가 일시 중단되면 사용자는 Fallback을 볼 수 없습니다. 데이터가 로딩될 때까지 이전 지연된 값이 표시됩니다.

- `useDeferredValue` 는 그 자체로 추가 네트워크 요청을 방지하지 않습니다.

- `useDeferredValue` 자체로 인한 고정된 지연은 없습니다. React는 원래의 리렌더링을 완료하자마자 즉시 새로운 지연된 값으로 백그라운드 리렌더링 작업을 시작합니다. 그러나 이벤트로 인한 업데이트(예: 타이핑)는 백그라운드 리렌더링을 중단하고 우선순위를 갖습니다.

- `useDeferredValue` 로 인한 백그라운드 리렌더링은 화면에 커밋될 때까지 Effect를 실행하지 않습니다. 백그라운드 리렌더링이 일시 중단되면 데이터가 로딩되고 UI가 업데이트된 후에 해당 Effect가 실행됩니다.

사용법

새 콘텐츠가 로딩되는 동안 오래된 콘텐츠 표시하기

컴포넌트의 최상위 레벨에서 `useDeferredValue` 를 호출하여 UI 일부 업데이트를 지연할 수 있습니다.

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

function SearchPage() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  // ...
}
```

초기 렌더링 중에 `지연된 값` 은 사용자가 제공한 `값` 과 일치합니다.

업데이트가 발생하면 `지연된 값` 은 최신 `값` 보다 "뒤쳐지게" 됩니다. React는 먼저 지연된 값을 업데이트하지 *않은 채로* 렌더링한 다음, 백그라운드에서 새로 받은 값으로 리렌더링을 시도합니다.

이것이 언제 유용한지 예시를 통해 살펴보겠습니다.

중요합니다!

이 예시에서는 Suspense 지원 데이터 소스 중 하나를 사용한다고 가정합니다.

- Relay와 Next.js 같이 Suspense를 지원하는 프레임워크로 데이터 가져오기.
- `lazy` 를 활용한 지연 로딩 컴포넌트.
- `use` 를 사용해서 Promise 값 읽기.

Suspense와 그 한계에 대해 자세히 알아보기.

아래 예시에서는 검색 결과를 불러오는 동안 `SearchResults` 컴포넌트가 일시 중지Suspend됩니다. `"a"` 를 입력하고 결과를 기다린 다음 `"ab"` 로 수정해 보세요. `"a"` 에 대한 결과가 로딩 폴백으로 대체될 것입니다.

App.js SearchResults.js 새로고침 Clear 포크

```
import { Suspense, useState } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
  const [query, setQuery] = useState('');
  return (
    <>
      <label>
        Search albums:
        <input value={query} onChange={e => setQuery(e.target.value)} />
      </label>
      <Suspense fallback={<h2>Loading...</h2>}>
```

자세히 보기

흔히 사용되는 또 다른 UI 패턴은 결과 목록의 업데이트를 지연(*defer*) 하고, 새로운 결과가 준비될 때까지 이전 결과를 계속 표시하는 것입니다. `useDeferredValue` 를 호출하여 쿼리의 지연된 버전을 전달하세요.

```
export default function App() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  return (
    <>
      <label>
        Search albums:
        <input value={query} onChange={e => setQuery(e.target.value)} />
      </label>
      <Suspense fallback={<h2>Loading...</h2>}>
        <SearchResults query={deferredQuery} />
      </Suspense>
    </>
  );
}
```

`query` 는 즉시 업데이트되므로 Input에 새 값이 표시됩니다. 그러나 `deferredQuery` 는 데이터가 로딩될 때까지 이전 값을 유지하므로 `SearchResults` 는 잠시 동안 오래된 결과를 표시합니다.

아래 예시에서 `"a"` 를 입력하고 결과가 로딩될 때까지 기다린 다음, 입력값을 `"ab"` 로 수정해보세요. 이제 새 결과가 로딩될 때까지 Suspense 폴백 대신 오래된 결과 목록이 표시되는 것을 확인할 수 있습니다.

App.js SearchResults.js 새로고침 Clear 포크

```
import { Suspense, useState, useDeferredValue } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  return (
    <>
      <label>
        Search albums:
```

```
          <input value={query} onChange={e => setQuery(e.target.value)} />
        </label>
```

자세히 보기

자세히 살펴보기

값을 지연시키는 것은 내부적으로 어떻게 작동하나요?

자세히 보기

콘텐츠가 오래되었음을 표시하기

위 예시에서는 최신 쿼리에 대한 결과 목록이 아직 로딩 중이라는 표시가 없습니다. 새 결과를 로딩하는 데 시간이 오래 걸리는 경우 사용자에게 혼란을 줄 수 있습니다. 결과 목록이 최신 쿼리와 일치하지 않는다는 것을 사용자에게 더 명확하게 알리기 위해, 오래된 결과 목록이 표시될 때 시각적 표시를 추가할 수 있습니다.

```
<div style={{
  opacity: query !== deferredQuery ? 0.5 : 1,
}}>
  <SearchResults query={deferredQuery} />
</div>
```

이렇게 변경하면 입력을 시작하자마자 새 결과 목록이 로딩될 때까지 오래된 결과 목록이 약간 어두워집니다. 아래 예시에서와 같이 점진적으로 어두워진다고 느껴지도록 CSS Transition을 추가하여 흐리게 표시되는 것을 지연시킬 수도 있습니다.

App.js SearchResults.js 새로고침 Clear 포크

```
import { Suspense, useState, useDeferredValue } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  const isStale = query !== deferredQuery;
  return (
    <>
      <label>
        Search albums:
        <input value={query} onChange={e => setQuery(e.target.value)} />
```

자세히 보기

UI 일부에 대해 리렌더링 지연하기

`useDeferredValue`를 성능 최적화 용도로 적용할 수도 있습니다. UI 일부의 리렌더링 속도가 느리고, 이를 최적화할 쉬운 방법이 없으며, 나머지 UI를 차단하지 않도록 하려는 경우에 유용합니다.

키 입력 시마다 리렌더링되는 텍스트 필드와 컴포넌트(예: 차트 또는 긴 목록)가 있다고 상상해 보세요.

```
function App() {
  const [text, setText] = useState('');
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <SlowList text={text} />
    </>
  );
}
```

먼저, Props가 같은 경우 리렌더링을 건너뛰도록 `SlowList`를 최적화합니다. 이렇게 하려면, `memo`로 감싸주세요.

```
const SlowList = memo(function SlowList({ text }) {
  // ...
});
```

그러나 이는 `SlowList` Props가 이전 렌더링 때와 *동일한* 경우에만 도움이 됩니다. 지금 직면하고 있는 문제는 Props가 *다르고* 실제로 다른 시각적 출력을 보여줘야 할 때 속도가 느리다는 것입니다.

구체적으로, 주요 성능 문제는 Input에 타이핑할 때마다 `SlowList` 가 새로운 Props를 수신하고 전체 트리를 리렌더링하면 타이핑이 끊기는 느낌이 든다는 것입니다. 이 경우 `useDeferredValue` 를 사용하면 입력 업데이트(빨라야 하는)를 결과 목록 업데이트(느려도 되는)보다 높은 우선순위에 둘 수 있습니다.

```
function App() {
  const [text, setText] = useState('');
  const deferredText = useDeferredValue(text);
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <SlowList text={deferredText} />
    </>
  );
}
```

이렇게 한다고 해서 `SlowList` 의 리렌더링 속도가 빨라지지는 않습니다. 하지만 키 입력을 차단하지 않도록 목록 리렌더링의 우선순위를 낮출 수 있다는 것을 React에 알려줍니다. 목록은 입력보다 “지연”되었다가 “따라잡을” 것입니다. 이전과 마찬가지로 React는 가능한 한 빨리 목록을 업데이트하려고 시도하지만, 사용자가 입력하는 것을 차단하지는 않습니다.

useDeferredValue와 최적화되지 않은 리렌더링의 차이점

1. 목록 리렌더링 지연
2. 목록의 최적화되지 않은 리렌더링

예시 1 of 2: 목록 리렌더링 지연

이 예시에서는 `SlowList` 컴포넌트의 각 항목을 **인위적으로 느려지도록 하여** `useDeferredValue` 를 통해 input의 반응성을 유지하는 방법을 확인할 수 있습니다. input에 타이핑하면 입력은 빠르게 느껴지는 반면 목록은 “지연”되는 것을 확인할 수 있습니다.

App.js SlowList.js 새로고침 Clear 포크

```
import { useState, useDeferredValue } from 'react';
import SlowList from './SlowList.js';

export default function App() {
  const [text, setText] = useState('');
  const deferredText = useDeferredValue(text);
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <SlowList text={deferredText} />
    </>
  );
```

다음 예시

주의하세요!

이 최적화를 위해서는 `SlowList` 를 `memo` 로 감싸야 합니다. `text` 가 변경될 때마다 React는 부모 컴포넌트를 빠르게 리렌더링할 수 있어야 하기 때문입니다. 리렌더링하는 동안 `deferredText` 는 여전히 이전 값을 가지므로 `SlowList` 는 리렌더링을 건너뛸 수

있습니다. (Props는 변경되지 않았습니다.) `memo` 가 없으면 어쨌든 리렌더링해야 하므로 최적화의 취지가 무색해집니다.

자세히 살펴보기

값을 지연하는 것은 디바운싱 및 스로틀링과 어떤 점이 다른가요?

자세히 보기

이전
useDebugValue

다음
useEffect