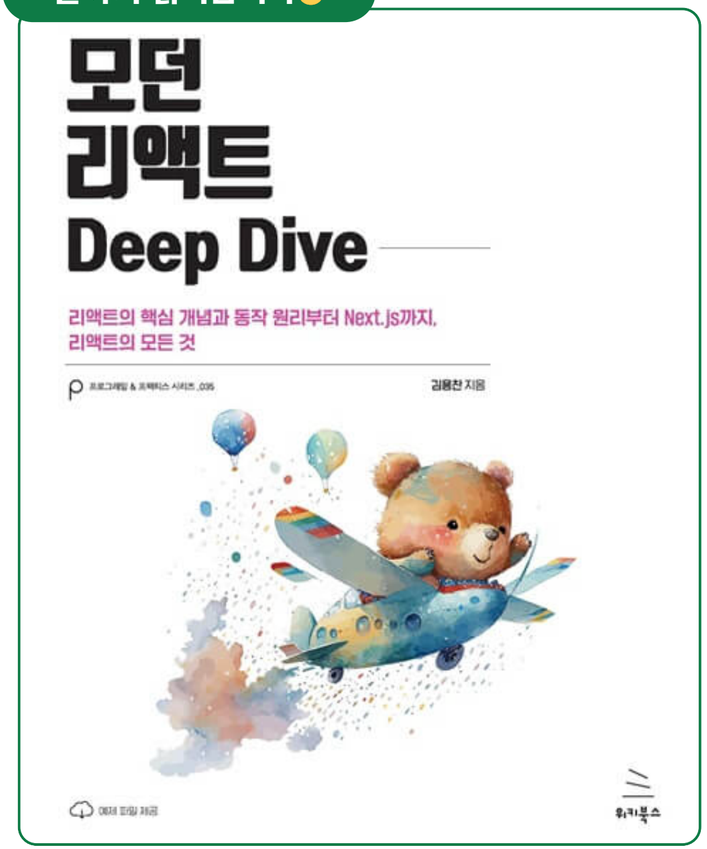
리액트
그게 뭔데

리액트 핵심 요소 깊게 살펴보기
– 모던 리액트 Deep Dive

2024.03.10
1주차 발표 장표
두선아

끝까지 읽어봅시다🔥



시작하기 전에

- 알고 쓰자 리액트
- DIL의 중요성: 시간을 정하고, 학습하자
- 파이팅의 중요성👍

2장. 리액트 핵심 요소 깊게 살펴보기

코어하다 코어해

JSX
116~128P

- 맨날 쓰는 그거
- JSX의 정의
- createElement

가상 DOM과 React Fiber
128~143P

- 브라우저 랜더링과 가상 DOM
- React Fiber의 개념
- Fiber Tree

Class Component Function Component
143~172P

- 올 게 왔다 Class Component
- Function Component와 생명주기, 랜더링
- 리액트의 랜더링 프로세스

Memoization
182~188P

- 최적화를 위한 각고의 노력
- 예제를 만들어 봄

“

JSX

Javascript XML

“JSX란?

JavaScript 코드 내에서 HTML과 유사한 구문을 사용하여
UI를 구조화하고 컴포넌트를 더욱 직관적으로 작성한다

오늘도 작성한 JSX

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import ErrorBoundary from "./ErrorBoundary";

const rootElement = document.getElementById("root")!;
const root = ReactDOM.createRoot(rootElement);

root.render(
  <React.StrictMode>
    <ErrorBoundary>
      <App />
    </ErrorBoundary>
  </React.StrictMode>
);
```

너를 좀 더 알아보자

JSX Element

- JSX에서는 HTML 태그와 유사한 형태로 표현됩니다.

```
<JSXOpeningElement JSXAttributes(optional)></JSXClosingElement>

<JSXSelfClosingElement JSXAttributes(optional) />

<></>
```

Open & Closing Tag

Self Closing Tag

Fragment Tag

JSX Attribute

- JSX Element에는 HTML과 유사하게 속성을 지정할 수 있습니다.
- 이러한 속성은 JSX Attribute를 통해 지정되며, 키-값 쌍으로 표현됩니다.

JSX의 장점

직관적인 UI 작성

- HTML과 유사한 구문을 사용하므로 UI를 작성하는 데 있어서 직관적이고 편리합니다.

컴포넌트 재사용성

- JSX를 사용하여 컴포넌트를 정의하면, 이를 다른 곳에서 쉽게 재사용할 수 있습니다.

JavaScript와의 통합

- JSX는 JavaScript 코드 내에서 자연스럽게 통합되므로
  별도의 템플릿 언어를 배우거나 사용할 필요가 없습니다.

JSX 변환

JSX

```
const ComponentA = <A required={true}>Hello</A>;
```

React.createElement

```
const ComponentA = React.createElement(A, { required: true }, "Hello");
```

- Babel 플러그인을 사용: @babel/plugin-transform-react-jsx

자동 런타임

```
const ComponentA = (0, _jsxRuntime.jsx)(A, { required: true, children: "Hello" });
```

- React 17 및 Babel 7.9.0 이후 버전에서는 자동 런타임 변환을 지원합니다.
- 이 경우, JSX가 _jsxRuntime.jsx 함수 호출로 변환됩니다.

빠지면 아쉬운

활용 예제🤔

```
function TextOrHeading({ isHeading, children }) {
  return createElement(
    isHeading ? "h1" : "p", // JSXElement
    { className: "text" }, // JSXAttribute (optional)
    children // JSXChildren (optional)
  );
}
```

이 함수에서는 JSXElement를 첫 번째 인수로 선언하여 요소를 정의하고
JSXAttribute와 JSXChildren을 이후 인수로 전달합니다.

이렇게 JSX를 React에서 활용해서
중복 코드를 줄이는 컴포넌트를 만들 수 있습니다.

“

JSX는 React에서 매우 중요한 개념이자
React 애플리케이션을 개발하는 데 있어서
반드시 익숙해져야 하는 도구이다.

JSX의 개념과 문법을 이해하면
더 나은 React 애플리케이션을 개발하는 데
도움이 되겠다😀

가상 DOM과 React Fiber

“ React의 특징

- 실제 DOM이 아닌 가상 DOM을 사용해 렌더링을 최적화한다.

Document Object Model

- DOM은? 웹 페이지의 구조화된 표현을 제공하는 인터페이스

만들어볼까요

1. 브라우저가 사용자가 요청한 주소를 방문해 HTML을 받아온다
2. 브라우저 랜더링 엔진이 HTML을 파싱해 DOM 트리를 생성한다 (DOM 노드로 구성됨)
3. 2번 과정에서 CSS 파일을 받아온다
4. CSS를 파싱해 CSSOM 트리를 생성한다
5. 2번에서 만든 DOM 노드를 순회한다. 사용자의 눈에 보이는 노드를 방문한다.
6. 5번의 노드를 대상으로, 해당 노드의 CSSOM 노드를 찾아 적용한다.

레이아웃 Layout/Reflow

- 노드의 크기와 위치를 계산

페인팅 Painting

- 노드를 화면에 그림

왜? 가상 DOM?

레이아웃과 페인팅은? 비용이 크다

- 사용자의 상호작용에 따라 레이아웃과 페인팅이 반복되어야 한다
- 레이아웃이 발생하면 페인팅도 발생한다
- 상위 노드의 레이아웃이 변경되면 하위 노드의 레이아웃도 변경된다

여기서 잠깐

가상 DOM이
일반 DOM을 관리하는 거보다
무조건 빠른 것이 아님
애플리케이션을 개발할 수 있을 만큼의
합리적이고 빠른 방법을 제공한다

– 댄 애브라모프 (리액트 개발자)

SPA의 경우는?

- 하나의 페이지에서 계속해서 레이아웃과 페인팅을 반복한다
- 라우팅이 발생하면, 헤더, 푸터 등을 제외한 대부분의 노드가 변경된다
- DOM의 변경 사항을 추적하고, 변경된 노드를 찾아 레이아웃과 페인팅을 반복한다

가상 DOM은?

- react-dom이 관리하는 가상 DOM은 실제 DOM과 동일한 구조를 가진다
- 웹 페이지가 표시해야할 DOM을 메모리에 저장하고, 변경 사항을 추적한다

기본 개념

React Fiber는?

- 자바스크립트 객체

Fiber Reconciler는?

- 가상 DOM을 관리한다.

Reconciliation은?

- 가상 DOM과 실제 DOM의 변경 사항을 비교하고, 변경 사항을 반영한다

리액트 파이버 🖐

동작 방식

1. 작업을 작은 단위로 분할하고, 우선순위를 정한다
2. 렌더링을 중단하고, 다시 시작할 수 있다
3. 이전 작업을 다시 재사용하거나, 폐기할 수 있다

단계 Phase

1. 랜더 단계 Render Phase : 사용자에게 노출되지 않는 비동기 작업을 수행 (예: 데이터 패칭, 레이아웃 계산)
2. 커밋 단계 Commit Phase : 실제 DOM에 반영한다 (동기식으로 일어나고, 중단될 수 없다)

- (동기) 과거 리액트의 조정 알고리즘은 stack을 사용했다 (싱글 스레드이므로 블로킹이 발생한다)
- (비동기) 리액트 파이버는 linked list를 사용해 작업을 분할하고, 우선순위를 정한다

파이버 객체

Fiber Object

컴포넌트가 최초로 마운트 되는 시점에 생성되어
가급적 재사용된다

Fiber Object의 특징

state가 변경되거나, 생명주기 메서드가 호출되면
파이버 객체가 업데이트된다.

리액트가 파이버를 처리할 때,
파이버 객체를 참조해 작업을 수행한다

우선 순위에 따라 유연하게 작업을 처리한다

```
function FiberNode(tag, pendingProps, key, mode) {
  // Instance
  this.tag = tag; // HostComponent(div같은 요소), FunctionComponent, ClassComponent
  this.key = key;
  this.elementType = null;
  this.type = null;
  this.stateNode = null; // 파이버 자체에 대한 reference 정보

  // Fiber (트리 구조)
  this.return = null;
  this.child = null; // 첫 번째 자식
  this.sibling = null; // 다음 형제
  this.index = 0; // 현재 자식의 인덱스
  this.ref = null;
  this.refCleanUp = null;

  this.pendingProps = pendingProps; // workInProgress에 대한 props
  this.memoizedProps = null; // pendingProps를 기반으로 렌더링된 이후의 props를 저장
  this.updateQueue = null; // 상태 업데이트, 콜백 함수, 이펙트 등을 저장
  this.memoizedState = null;
  this.dependencies = null;

  this.mode = mode;

  // Effects
  this.flags = NoFlags;
  this.subtreeFlags = NoFlags;
  this.deletions = null;

  this.lanes = NoLanes;
  this.childLanes = NoLanes;

  this.alternate = null;

  // 이하 프로파일러, __DEV__
}
```

React Fiber Tree🌲

- 파이버 트리는 리액트 내부에 2개의 트리를 가지고 있다
- 현재 트리(Current)와 대체 트리(WorkInProgress)로 구성된다: 더블 버퍼링

더블 버퍼링

- 현재 트리를 사용자에게 노출하고, 대체 트리에 변경 사항을 적용한다
- 변경 사항이 적용되면, 현재 트리와 대체 트리를 교체한다
- 불완전한 트리를 보여주지 않기 위해, 커밋 단계에서 더블 버퍼링을 사용한다

동작

1. current을 기준으로 작업 시작
2. 업데이트가 발생하면, 리액트에서 받은 데이터로 workInProgress 트리 빌드
3. workInProgress 트리 빌드가 끝나면, 다음 랜더링에 사용
4. current가 workInProgress로 교체된다

React Fiber Tree🌲

작업 순서: DFS

1. beginWork(): 파이버 작업을 시작한다, 자식이 없는 파이버를 만들 때까지 순회한다
2. 만약 형제가 없다면, completeWork(): 파이버 작업을 완료한다.
3. 만약 형제가 있다면, 형제로 이동한다
4. 2번, 3번이 끝나면 return을 찾아 올라가, 작업을 완료한다.

- props를 받아 파이버 내부에서 작업을 수행한다
- 재귀 순회 트리 업데이트 과정: 과거에는 동기, 현재는 비동기
- 우선 순위에 따라 유연하게 동작한다
  - 애니메이션 > 사용자 입력 > 데이터 가져오기 > 레이아웃 계산

“

실제 DOM에 반영하는 것은
동기적으로 일어나야 하고, 처리량이 많기 때문에
화면에 불완전하게 표시될 수 있는 가능성이 있다

그래서 작업을 가상/메모리상에서 수행하고
최종 결과물만 실제 브라우저에 적용하는 것이 가상 DOM이다

즉, 빠르게 랜더링하는 것이 아니라
불완전한 화면을 사용자에게 보여주지 않는 것
바로 값으로 UI를 표현하는 것(Value UI)
이런 흐름을 효율적으로 관리하기 위한 매커니즘이
가상 DOM과 리액트의 핵심이다

“

Class Component
Function Component

Class Component

class 컴포넌트의 constructor

- state을 초기화 한다
- props를 super에 전달한다
- Component, PureComponent

생명주기 메서드 Lifecycle Methods

- 1. mount
- 2. update
- 3. unmount

^^;;

```
class MyComponent extends React.Component: <MyProps, MyState> {
  constructor(props: MyProps) {
    super(props)
    this.state = {
      count: 0
    }
  }
  ...
}

public render() {
  const {
    props: { required, text },
    state: { count, isLimited },
  } = this;

  return (
    <div>{count}</div>
  )
}
```

예제 만들기

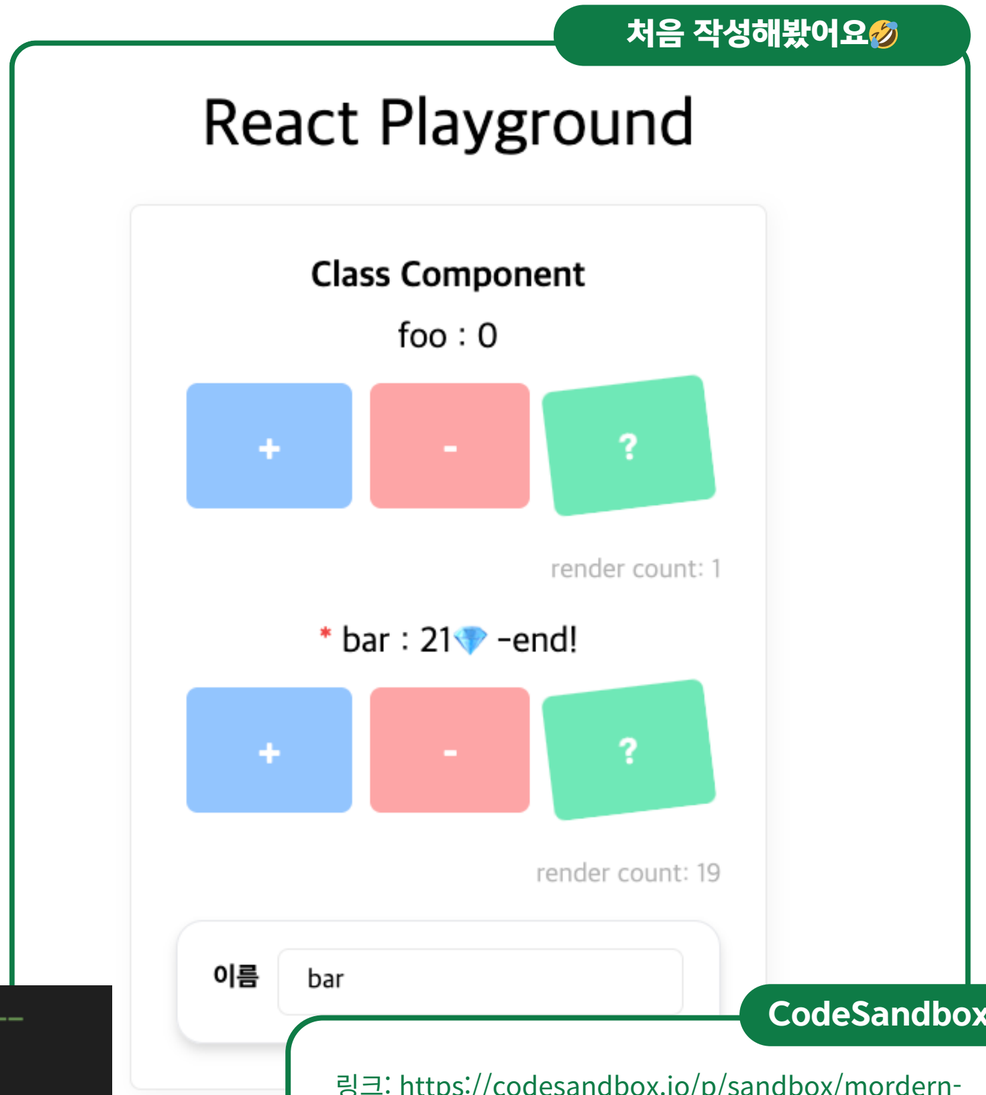
B. 생명주기 메소드 써보기

```
// ----------------------------------------
/** B. life-cycle method
 *
 * [Mount]
 * constructor()
 * -> static getDerivedStateFromProps()
 * -> render()
 * -> componentDidMount()
 *
 * [Update]
 * static getDerivedStateFromProps()
 * -> shouldComponentUpdate()
 * -> render()
 * -> getSnapshotBeforeUpdate()
 * -> componentDidUpdate()
 *
 * [Unmount]
 * componentWillUnmount()
 * */
```

A. Class Component의 메소드와 This

```
// ----------------------------------------
/**
 * A. method & this binding
 * - 1️⃣ this binding in function declaration
 * - 2️⃣ arrow function's this is upper scope
 * - 3️⃣ new method each time when render
 */
```

처음 작성해봤어요🤣



CodeSandbox

링크: https://codesandbox.io/p/sandbox/mordern-react-deep-dive-t9vlwm

docs/dusunax/dil/2024-03-08_sample.tsx하고 같은 코드입니다~

한계점

데이터의 흐름을 추적하기 어렵다
state의 업데이트 위치 어디? 생명 주기 메소드 코드의 흐름 파악해야함

재사용이 어렵다
HOC로 감싸거나, props를 전달해서 재사용 시, wrapper hell에 빠질 수 있다.
컴포넌트 상속으로 재사용 시, 클래스의 흐름을 쫓아야 함

컴포넌트가 커진다
로직, 내부 데이터 흐름이 복잡하면 생명주기 메서드를 많이 써서 컴포넌트가 뚱뚱해짐

상대적으로 어렵다
JS 개발자들이 class 문법에 익숙하지 않고, 혼돈의 JS this

코드 크기 최적화 어려움
번들 크기를 줄이기 어렵다. 메서드명이 minified되지 않고, 사용하지 않는 메서드가 트리쉐이킹 되지 않음

hot reloading에 불리

- class component: 최초 랜더링 시 instance를 생성 => 내부에서 state 값 관리.
  - instance 내부에 있는 render()를 수정할 시, instance를 새로 만듦 => state이 초기화됨
- function component: state 값을 클로저에 저장, 함수가 다시 실행되도 state을 잃지 않음

Function Component

16.8 버전 이전에는

- 무상태 컴포넌트를 구현하기 위한 수단

랜더링된 값으로 알아보는 차이점

구분	차이점	props	설명
class component	랜더링된 값을 고정하지 않음	this.props	따라서 생명주기 메서드가 변경된 nextProps 값을 읽을 수 있다 (변화하는 this)
function component	랜더링된 값을 고정함	props는 인수	랜더링이 일어날 때, 그 순간의 props, state을 기준으로 랜더링

class component에서 props를 고정하고 싶다면?

- 미리 함수를 실행해 인수에 전달해놓는다.
- render(){ 여기 } 내에 값을 넣는다. 이상해요

Function Component

16.8 버전 이전에는

- 무상태 컴포넌트를 구현하기 위한 수단

랜더링된 값으로 알아보는 차이점

구분	차이점	props	설명
class component	랜더링된 값을 고정하지 않음	this.props	따라서 생명주기 메서드가 변경된 nextProps 값을 읽을 수 있다 (변화하는 this)
function component	랜더링된 값을 고정함	props는 인수	랜더링이 일어날 때, 그 순간의 props, state을 기준으로 랜더링

class component에서 props를 고정하고 싶다면?

- 미리 함수를 실행해 인수에 전달해놓는다.
- render(){ 여기 } 내에 값을 넣는다. 이상해요

React Rendering

랜더링이란?

리액트 애플리케이션 트리 안에 있는 모든 컴포넌트들이
현재 자신들이 가지고 있는 props와 state의 값을 기반으로
어떻게 UI를 구성하고, 이를 바탕으로 어떤 DOM 결과를
브라우저에 제공할 것인지 계산하는 일련의 과정

브라우저 랜더링에 필요한
DOM 트리를 만드는 과정

React Rendering

최초 랜더링

- 사용자가 처음 애플리케이션에 진입
- 브라우저에 랜더링할 결과물(정보)를 제공하기 위해 최초 랜더링 수행

리랜더링

클래스 컴포넌트

- setState 실행
- forceUpdate 실행(shouldComponentUpdate를 건너뜀, render 내부에서 실행하면? 무한 루프)

함수 컴포넌트

- useState의 setter 실행
- useReducer의 dispatch 실행
- key props가 변경되는 경우
- 부모 컴포넌트가 랜더링 될 경우
- props가 변경되는 경우

React Rendering

랜더링 프로세스

- 랜더링 프로세스가 시작되면, 컴포넌트 루트에서부터 아래쪽으로 내려가며
  업데이트가 필요한 컴포넌트를 찾는다
  - 발견 시, 클래스 컴포넌트라면 render()
  - 발견 시, 함수 컴포넌트라면 FunctionComponent()를 호출 후, 결과 저장
  - JSX니까 createElement()를 호출하는 구문으로 변환된다
- Reconciliation 재조정
  - 컴포넌트의 랜더링 결과물을 수집, 가상 DOM과 비교해서
    실제 DOM에 반영하기 위한 변경 사항 수집하는 과정.
- 변경 사항을 동기 시퀀스로 DOM에 적용

```
// 결과
{type: MyComponent, props: {count: 0, required: true, children: "안녕"}}
```

“

React Rendering

랜더 Render Phase

- 컴포넌트를 실행: render(), return
- 결과와 이전 가상 DOM을 비교
- type, props, key 중 변경이 있다면 변경이 필요한 컴포넌트

커밋 Commit Phase

- 랜더 단계의 변경 사항을 실제 DOM에 적용
- 업데이트 시, 만든 DOM 노드 및 인스턴스를 리액트 내부의 참조를 업데이트
  - componentDidMount, componentDidUpdate 메서드, useLayoutEffect 훅 호출
- 커밋 이후, 브라우저의 랜더링이 발생한다

리액트의 랜더링이 일어나도 랜더 단계에서 변경사항을 감지할 수 없다면?

- 커밋 단계가 생략되어 브라우저 DOM 업데이트가 일어나지 않을 수 있다

“

실제 DOM에 반영하는 것은
동기적으로 일어나야 하고, 처리량이 많기 때문에
화면에 불완전하게 표시될 수 있는 가능성이 있다

그래서 작업을 가상/메모리상에서 수행하고
최종 결과물만 실제 브라우저에 적용하는 것이 가상 DOM이다

즉, 빠르게 랜더링하는 것이 아니라
불완전한 화면을 사용자에게 보여주지 않는 것
바로 값으로 UI를 표현하는 것(Value UI)
이런 흐름을 효율적으로 관리하기 위한 매커니즘이
가상 DOM과 리액트의 핵심이다

“

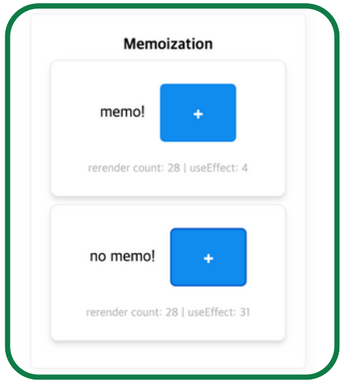
Memoization

메모하냐 마느냐

그 것이 문제로다

갑론을박

memoization 코드 예제
memo와 객체의 얕은 비교



CodeSandbox

링크: https://codesandbox.io/p/sandbox/mordern-react-deep-dive-t9vlwm

docs/dusunax/dil/2024-03-09_sample.tsx하고 같은 코드입니다~

반대파

memoization도 비용이다!

- 값을 비교하고 랜더링 혹은 재계산이 필요한지 확인하는 작업
- 결과물을 저장해 두었다가 다시 꺼내오는 작업

신중하게 해야한다.

- premature optimization, premature memoization
- 섣부른 최적화를 경계하자

silver bullet이 아니다

- 리액트가 모든 컴포넌트를 PureComponent로 만들거나, memo로 감싸는 작업을 하지 않은 이유
  - 메모이제이션은 트레이트 오프가 있는 기능이다.

찬성파

memoization은 비용이다?

- props에 대한 얕은 비교이다.
  리랜더링 리액트의 Reconciliation 알고리즘과 같다.
  어차피 결과물은 저장 중

잠재적 위험 비용을 아낀다.

- 랜더링 비용이 짱 비쌈
- 컴포넌트 내부의 복잡한 로직 재실행
  - 그리고 자식 컴포넌트에서 반복적으로 일어남
- 리액트가 이전 트리 and 신규 트리 비교하는 비용

성능 모니터링보다야 섣부른 최적화의 이점이 더 크다

개인적인 결론:
아직 리액트 성능 모니터링에 대한 개념 이해가 완벽하지 않으므로
성능 개선이 확실한 / 필요한 지점에서 memoization을 사용하자

감사합니다

레퍼런스/출처

모던 리액트 딥 다이브 – 김용찬, 위키북스

How browsers work: Behind the scenes of modern web browsers

How web browsers work – parsing the CSS

링크

코드 예제: https://codesandbox.io/p/sandbox/mordern-react-deep-dive-t9vlwm

(월간 CS-3월) 모던 리액트 딥 다이브: https://github.com/monthly-cs/2024-03-modern-react-deep-dive