

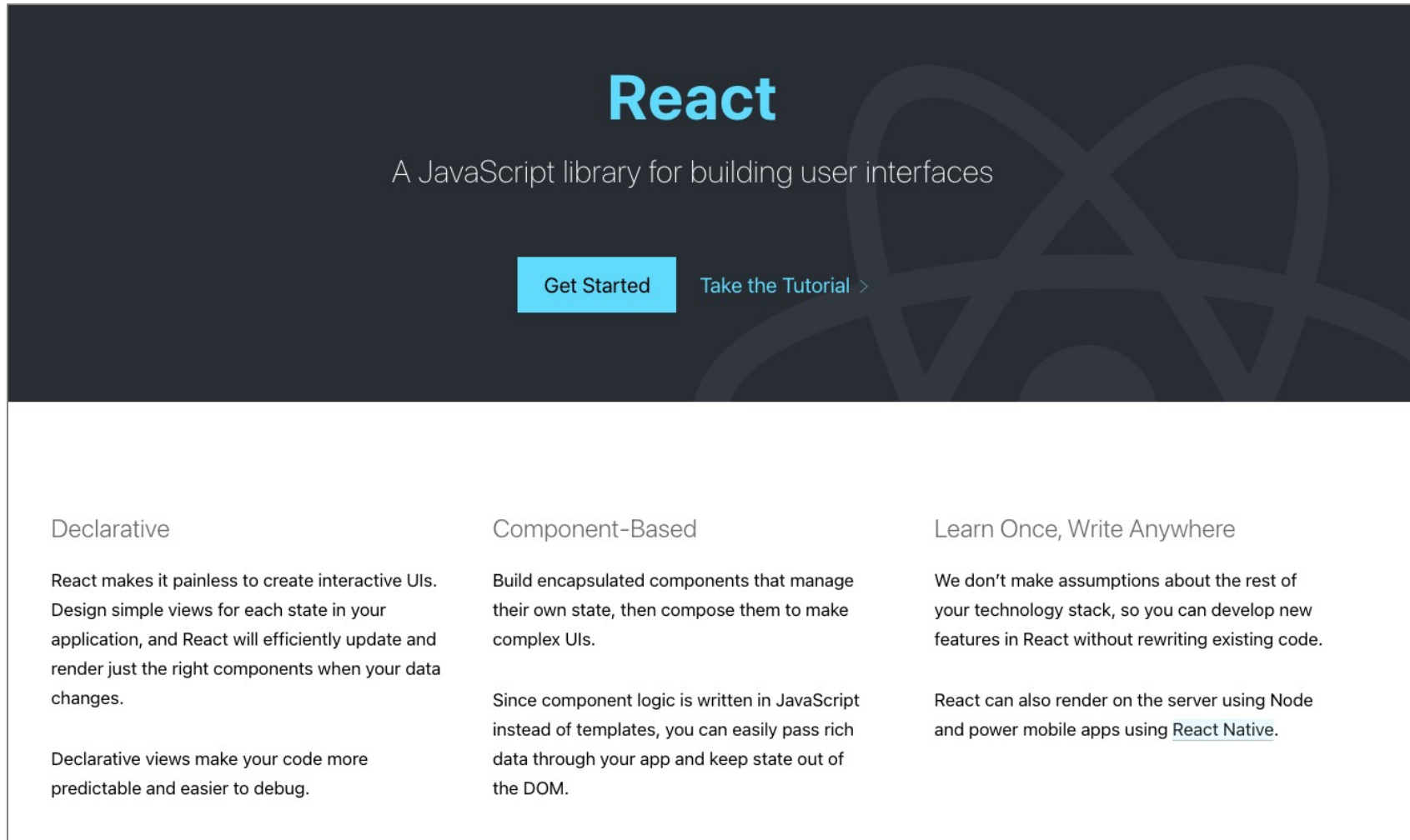
React 실행 플로우 분석

0.목차

- React 소개
- 용어 정리
- 플로우 분석

React 소개

1. 리액트 소개

A screenshot of the React.js landing page. The top section has a dark background with the word "React" in large blue letters, followed by the tagline "A JavaScript library for building user interfaces". Below this are two buttons: "Get Started" (highlighted in blue) and "Take the Tutorial >". The bottom section is white and contains three columns of text describing React's features: Declarative, Component-Based, and Learn Once, Write Anywhere.

React

A JavaScript library for building user interfaces

[Get Started](#) [Take the Tutorial >](#)

Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

Learn Once, Write Anywhere

We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using [React Native](#).

1. 리액트 소개

👍 좋아요

👍 좋아요

명령적

```
if( user.likes() ) {  
  if( hasBlue() ) {  
    removeBlue();  
    addGrey();  
  } else {  
    removeGrey();  
    addBlue();  
  }  
}
```

명시적

Declarative

React makes it painless to create interactive UIs.

```
if( this.state.liked ) {  
  return <blueLike />;  
} else {  
  return <greyLike />;  
}
```

predictable and easier to debug.

React

A JavaScript library for building user interfaces

Get Started

Take the Tutorial >

Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

Learn Once, Write Anywhere

We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using [React Native](#).

1. 리액트 소개

React

A JavaScript library for building user interfaces

[Get Started](#) [Take the Tutorial >](#)

Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

Component-Based

Build encapsulated components that manage their own state, then compose them to form complex UIs.

컴포넌트 단위 개발

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

Learn Once, Write Anywhere

We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using [React Native](#).

1. 리액트 소개

React

A JavaScript library for building user interfaces

[Get Started](#) [Take the Tutorial >](#)

Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

X 프레임워크 O 라이브러리

Learn [React with AWS Amplify](#)

We don't make any assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and powered by our open-source [React Native](#) library.

reconciler

renderer

1. 리액트 소개

renderer의 분리

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

```
33
34 ✓ function App() {
35   const [state, setState] = useState({ liked: 0 });
36   return (
37     <>
38       <button onClick={() => setState((state) => state.liked + 1)}>
39         <span>`liked ${state} time(s)`</span>
40         <SomeComponent />
41       </button>
42     </>
43   );
44 }
45
```

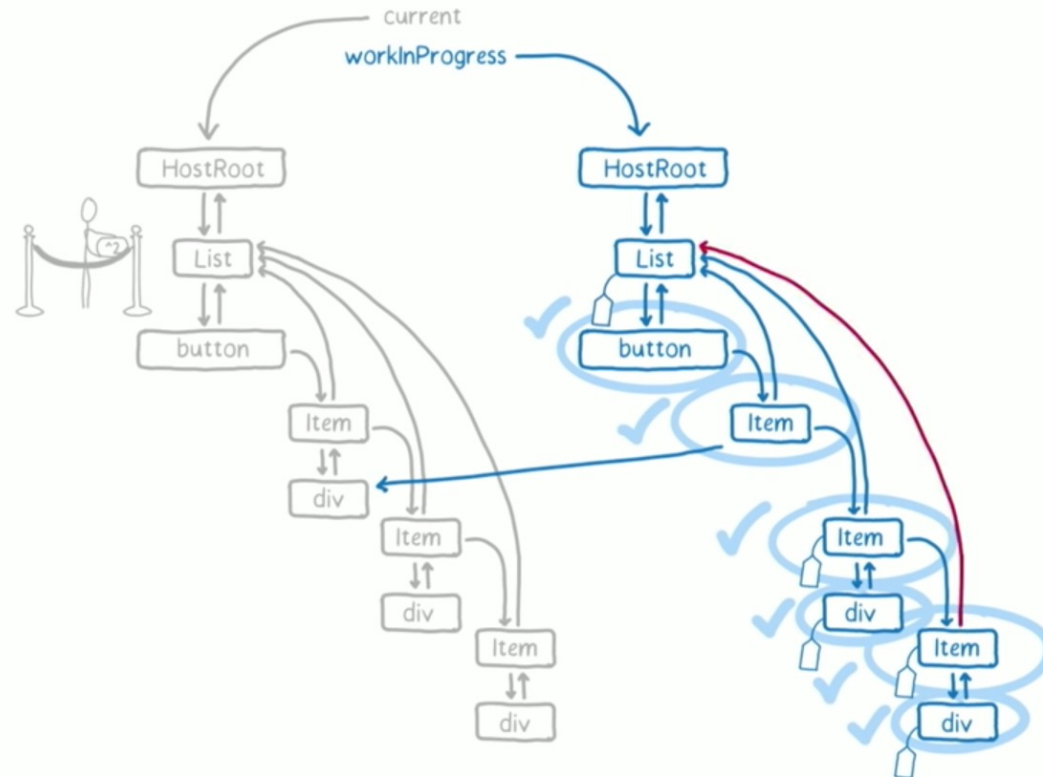
컴포넌트 단위 개발

2. 용어 정리

Reconciliation (재조정)

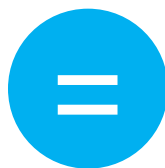
React는 메모리에 가상의 DOM 트리(**Fiber Tree**)를 관리하고, 이 가상의 DOM 트리에 변경사항을 선 적용한 뒤, 실제 DOM 트리와의 달라진 점만을 diff 해 변경이 필요한 (일부) 부분만을 리렌더링한다.

A 트리에서 A' 트리로, 변경사항을 반영하는 이 작업을 **reconciliation**이라고 한다.



1. 리액트 소개

React가 한마디로 해주는 일



(UI를 만들 때)

minimizing & batching dom updates

fiber reconciler engine

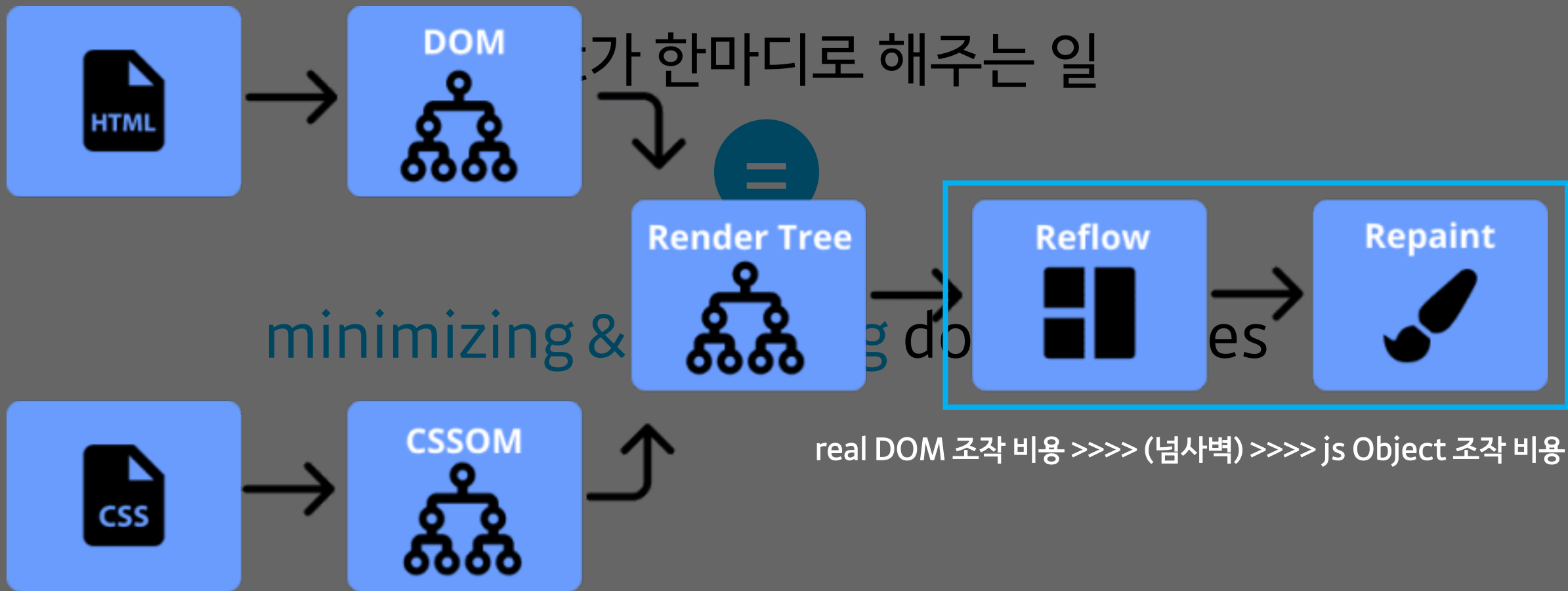
double buffering

2 phase

1. 리액트 소개

reflow: 화면 구조가 변경되었을 때, 뷰포트 내 렌더 트리 노드의 위치와 크기를 계산하는 과정

repaint: 화면에 변화가 있을 때 화면을 다시 그리는 과정



용어 정리

2. 용어 정리

Component, Element, fiber

```
33
34 function App() {
35   const [state, setState] = useState({ liked: 0 });
36   return (
37     <>
38       <button onClick={() => setState((state) => state.liked + 1)}>
39         <span>`liked ${state} time(s)`</span>
40         <SomeComponent />
41       </button>
42     </>
43   );
44 }
45
```

React에서 개발자가 UI를 선언하는 단위. class로도, function으로도 작성이 가능하다.

2. 용어 정리

Component, Element, fiber

(jsx 문법)

```
function App() {  
  const [state, setState] = useState({ liked: 0 });  
  return (  
    <>  
      <button onClick={() => setState((state) => state.liked + 1)}>  
        <span>`liked ${state} time(s)`</span>  
        <SomeComponent />  
      </button>  
    </>  
  );  
}
```

```
return /*#__PURE__*/ React.createElement(React.Fragment,  
null, /*#__PURE__*/ React.createElement("button", {  
  onClick: function onClick() {  
    return setState(function (state) {  
      return state.liked + 1;  
    });  
  }  
}, /*#__PURE__*/ React.createElement("span", null, "`liked  
$, state, " time(s)`"))));
```

DOM 노드나, 컴포넌트를 설명하는 plain, immutable 자바스크립트 객체

2. 용어 정리

Component, Element, fiber

```
var ReactDOM = function (type, key, ref, self, source, owner, props) {  
  You, 2 weeks ago | 1 author (You)  
  var element = {  
    // This tag allows us to uniquely identify this as a React Element  
    $$typeof: REACT_ELEMENT_TYPE,  
    // Built-in properties that belong on the element  
    type: type,  
    key: key,  
    ref: ref,  
    props: props,  
    // Record the component responsible for creating this element.  
    _owner: owner,  
  };  
};
```

```
▼ {$$typeof: Symbol(react.element), key: null, ref: null, props: {...}, type: f, ...} ⓘ  
  $$typeof: Symbol(react.element)  
  key: null  
  ▶ props: {}  
  ref: null  
  ▶ type: class LikeButton  
    _owner: null  
  ▶ _store: {validated: false}  
    _self: null  
    _source: null  
  ▶ __proto__: Object
```

```
▼ {$$typeof: Symbol(react.element), type: "button", key: null, ref: null, props: {...}, ...} ⓘ  
  $$typeof: Symbol(react.element)  
  key: null  
  ▶ props: {children: {...}, onClick: f}  
    ref: null  
    type: "button"  
  ▶ _owner: FiberNode {tag: 1, key: null, stateNode: LikeButton, elementType: f, type: f, ...}  
  ▶ _store: {validated: false}  
    _self: null  
    _source: null  
  ▶ __proto__: Object
```

DOM 노드나, 컴포넌트를 설명하는 plain, **immutable** 자바스크립트 객체

2. 용어 정리

Component, Element, fiber

```
function FiberNode(tag, pendingProps, key){
  // Instance
  this.tag = tag; // fiber의 종류를 나타냄
  this.key = key;
  this.type = null; // 추후에 React element의 type을 저장
  this.stateNode = null; // 호스트 컴포넌트에 대응되는 HTML element를 저장

  // Fiber
  this.return = null; // 부모 fiber
  this.child = null; // 자식 fiber
  this.sibling = null; // 형제 fiber
  this.index = 0; // 형제들 사이에서의 자신의 위치

  this.pendingProps = pendingProps; // workInProgress는 아직 작업이 끝난 상태가 아니므로 props를 pending으로
  this.memoizedProps = null; // Render phase가 끝나면 pendingProps는 memoizedProps로 관리
  this.updateQueue = null; // 컴포넌트 종류에 따라 element의 변경점 또는 라이프사이클을 저장
  this.memoizedState = null; // 함수형 컴포넌트는 훅을 통해 상태를 관리하므로 hook 리스트가 저장된다.

  // Effects
  this.effectTag = NoEffect; // fiber가 가지고 있는 side effect를 기록
  this.nextEffect = null; // side effect list
  this.firstEffect = null; // side effect list
  this.lastEffect = null; // side effect list

  this.expirationTime = NoWork; // 컴포넌트 업데이트 발생 시간을 기록
  this.childExpirationTime = NoWork; // 서브 트리에서 업데이트가 발생할 경우 기록

  this.alternate = null; // 반대편 fiber를 참조
}
```

Element에 1:1로 대응되는 Element의 확장팩.

내부 가상 DOM의 노드 역할을 하는 객체로,
컴포넌트에 필요한 모든 정보를 담고 있다.

2. 용어 정리

Component, Element, fiber

```
function FiberNode(tag, pendingProps, key){
  // Instance
  this.tag = tag; // fiber의 종류를 나타냄
  this.key = key;
  this.type = null; // 추후에 React element의 type을 저장
  this.stateNode = null; // 호스트 컴포넌트에 대응되는 HTML element를 저장

  // Fiber
  this.return = null; // 부모 fiber
  this.child = null; // 자식 fiber
  this.sibling = null; // 형제 fiber
  this.index = 0; // 형제들 사이에서의 자신의 위치

  this.pendingProps = pendingProps; // workInProgress는 아직 작업이 끝난 상태가 아니므로 props를 pending으로
  this.memoizedProps = null; // Render phase가 끝나면 pendingProps는 memoizedProps로 관리
  this.updateQueue = null; // 컴포넌트 종류에 따라 element의 변경점 또는 라이프사이클을 저장
  this.memoizedState = null; // 함수형 컴포넌트는 훅을 통해 상태를 관리하므로 hook 리스트가 저장된다.

  // Effects
  this.effectTag = NoEffect; // fiber가 가지고 있는 side effect를 기록
  this.nextEffect = null; // side effect list
  this.firstEffect = null; // side effect list
  this.lastEffect = null; // side effect list

  this.expirationTime = NoWork; // 컴포넌트 업데이트 발생 시간을 기록
  this.childExpirationTime = NoWork; // 서브 트리에서 업데이트가 발생할 경우 기록

  this.alternate = null; // 반대편 fiber를 참조
}
```

*"Fiber is **reimplementation of the stack**, specialized for React components. You can think of a single fiber as a **virtual stack frame**."*

The advantage of reimplementing the stack is that you can keep stack frames in memory and execute them however (and whenever) you want. This is crucial for accomplishing the goals we have for scheduling.

*Fiber is the **new reconciliation engine** in React 16. Its main goal is to enable incremental rendering of the virtual DOM.*

- reimplementation of the stack
- virtual stack frame
- unit of concurrency
- ...

2. 용어 정리

Component, Element, fiber

내부 자료구조 객체
reconciliation 엔진
동시성의 단위
virtual stack frame...

```
function FiberNode(tag, pendingProps, key){
  // Instance
  this.tag = tag; // fiber의 종류를 나타냄
  this.key = key;
  this.type = null; // 추후에 React element의 type을 저장
  this.stateNode = null; // 호스트 컴포넌트에 대응되는 HTML element를 저장

  // Fiber
  this.return = null; // 부모 fiber
  this.child = null; // 자식 fiber
  this.sibling = null; // 형제 fiber
  this.index = 0; // 형제들 사이에서의 자신의 위치

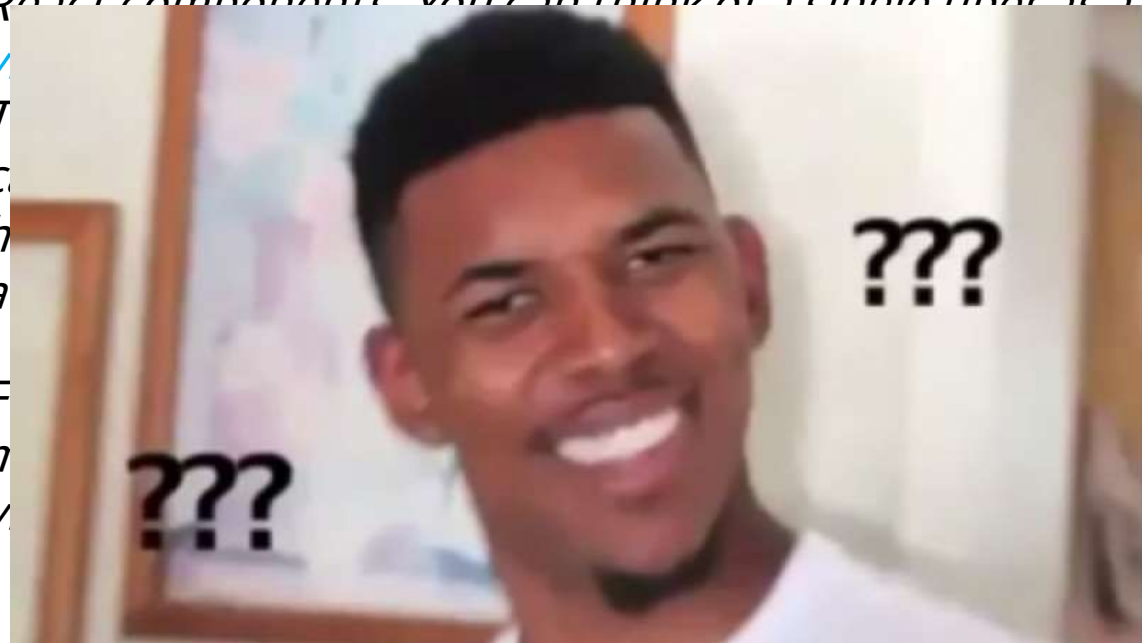
  this.pendingProps = pendingProps; // workInProgress는 아직 작업이 끝난 상태가 아니므로 props를 pending으로
  this.memoizedProps = null; // Render phase가 끝나면 pendingProps는 memoizedProps로 관리
  this.updateQueue = null; // 컴포넌트 종류에 따라 element의 변경점 또는 라이프사이클을 저장
  this.memoizedState = null; // 함수형 컴포넌트는 훅을 통해 상태를 관리하므로 hook 리스트가 저장된다.

  // Effects
  this.effectTag = NoEffect; // fiber가 가지고 있는 side effect를 기록
  this.nextEffect = null; // side effect list
  this.firstEffect = null; // side effect list
  this.lastEffect = null; // side effect list

  this.expirationTime = NoWork; // 컴포넌트 업데이트 발생 시간을 기록
  this.childExpirationTime = NoWork; // 서브 트리에서 업데이트가 발생할 경우 기록

  this.alternate = null; // 반대편 fiber를 참조
}
```

*"Fiber is **reimplementation of the stack**, specialized for React components. You can think of a single fiber as a*



- virtual stack frame
- unit of concurrency
- ...

2. 용어 정리

리액트 컴포넌트 = functions of data

리액트 컴포넌트 트리 = nested function call

리액트 컴포넌트 렌더링 = calling function

```
function FiberNode(tag, pendingProps, key){
  // Instance
  this.tag = tag; // fiber의 종류를 나타냄
  this.key = key;
  this.type = null; // 추후에 React element의 type을 저장
  this.stateNode = null; // 호스트 컴포넌트에 대응되는 HTML element

  // Fiber
  this.return = null; // 부모 fiber
  this.child = null; // 자식 fiber
  this.sibling = null; // 형제 fiber
  this.index = 0; // 형제들 사이에서의 자신의 위치

  this.pendingProps = pendingProps; // workInProgress는 아직
  this.memoizedProps = null; // Render phase가 끝나면 pending
  this.updateQueue = null; // 컴포넌트 종류에 따라 element의 변경값
  this.memoizedState = null; // 함수형 컴포넌트는 훅을 통해 상태를

  // Effects
  this.effectTag = NoEffect; // fiber가 가지고 있는 side effect
  this.nextEffect = null; // side effect list
  this.firstEffect = null; // side effect list
  this.lastEffect = null; // side effect list

  this.expirationTime = NoWork; // 컴포넌트 업데이트 발생 시간을 가
  this.childExpirationTime = NoWork; // 서브 트리에서 업데이트가

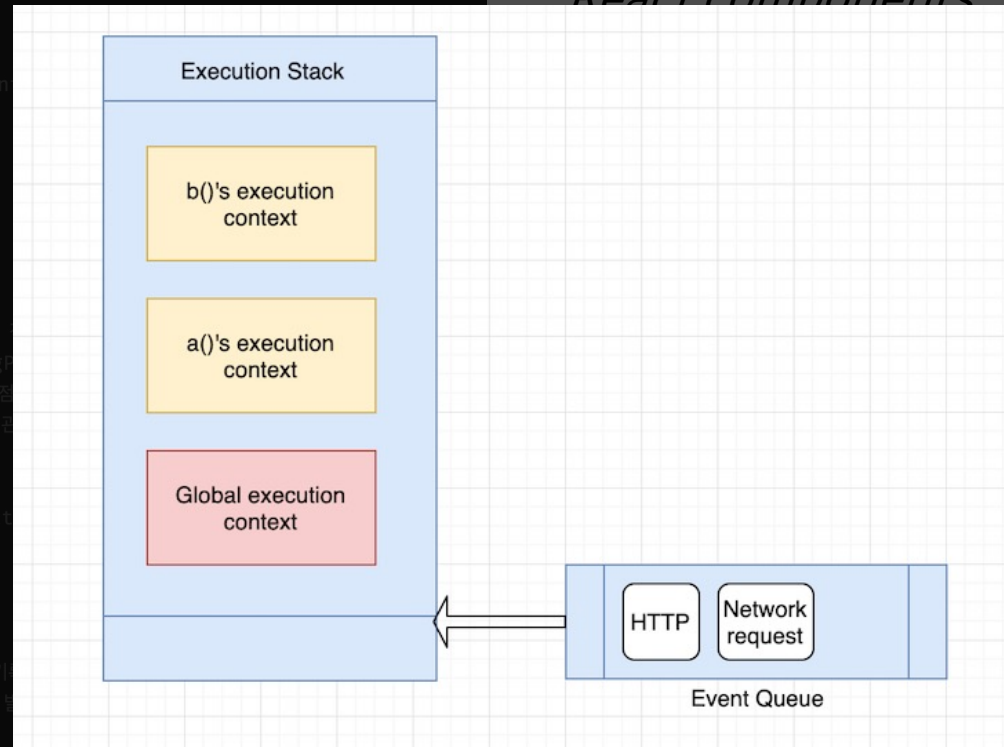
  this.alternate = null; // 반대편 fiber를 참조
}
```

*"Fiber is **reimplementation of the stack**, specialized for React components. You can think of a single fiber as a*

*implementing the stack is that you
s in memory and execute them
(over) you want. This is crucial for
als we have for scheduling.*

*conciliation engine in React 16. Its
e incremental rendering of the*

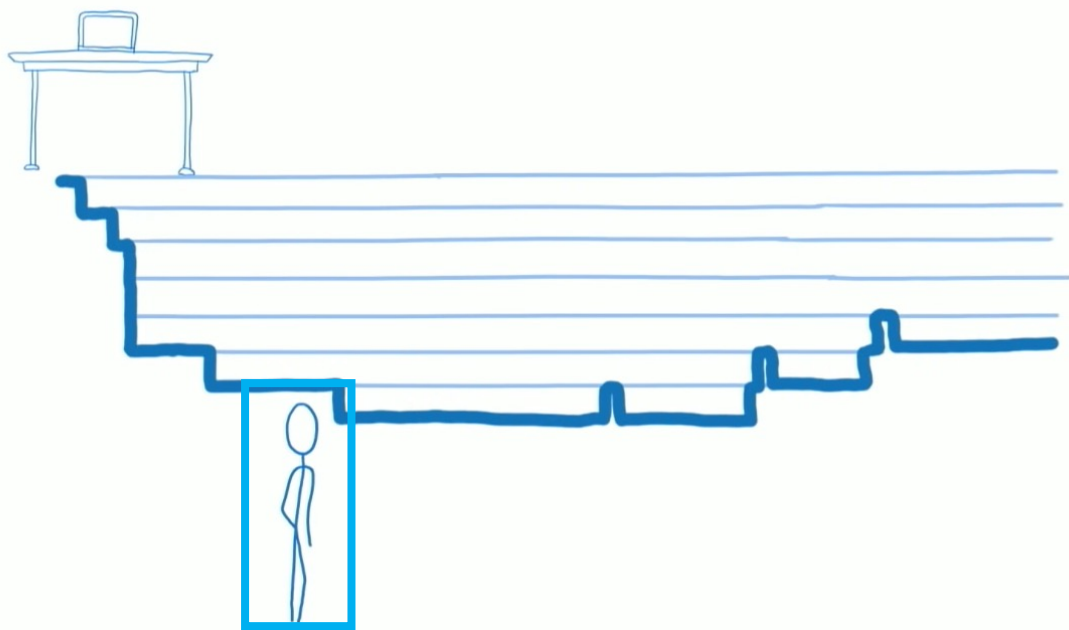
ementation of the stack
stack frame
concurrency



2. 용어 정리

Fiber이 해결하려고 했던 문제

(이전) stack reconciler



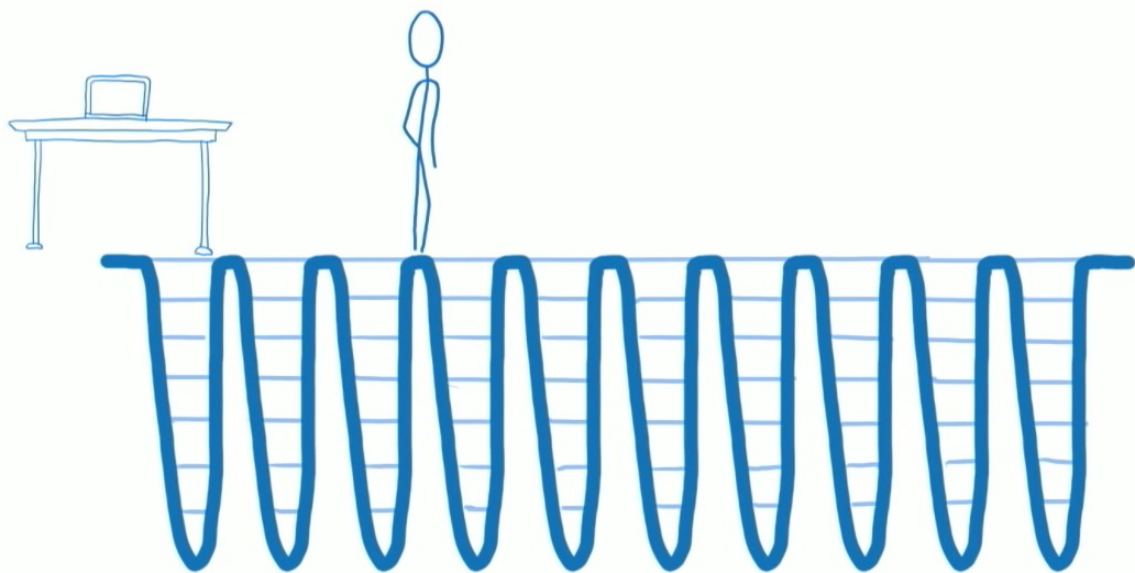
main thread

- ▶ 트리의 Leaf node에 도달할 때까지 update, mount를 recursive하게 실시한다.
- ▶ 즉, 거대한 한번의 렌더링이 call stack을 막아 다른 중요도 높은 작업 (애니메이션) 등의 실행을 막는다.
- ▶ 작업의 임시 중단, 취소, 우선순위 설정, 스케줄링 불가능

2. 용어 정리

Fiber이 해결하려고 했던 문제

fiber reconciler



- ▶ 작업을 더 작은 단위로 쪼갤 수 있게 한다.
- ▶ 중간에 현재의 rendering call stack을 중단하고, stash한 뒤 우선순위가 더 높은 작업을 실시할 수 있다.
- ▶ 중단된 상태를 기억해 작업을 재개할 수 있다.

```
requestIdleCallback((deadline) => {  
  // while we have time, perform work for a part of the components tree  
  while ((deadline.timeRemaining() > 0 || deadline.didTimeout) && nextComponent) {  
    nextComponent = performWork(nextComponent);  
  }  
});
```

브라우저의 메인 스레드가 비어 있으면 지정한 콜백 함수를 실행하도록 지시할 수 있는 함수

2. 용어 정리

Fiber이 해결하려고 했던 문제

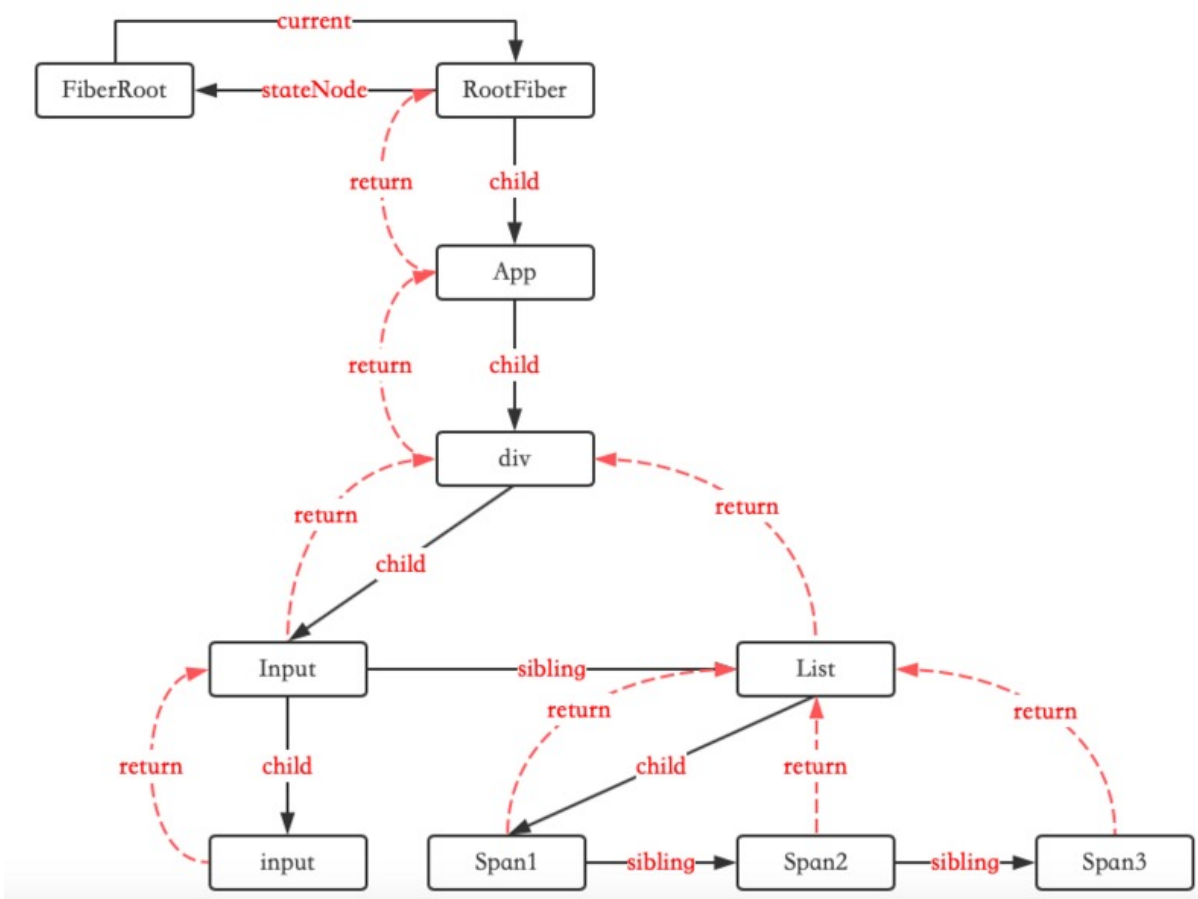


2. 용어 정리

(컴포넌트를 관리하기 위한, e.g. state, type, props...)

코드레벨에서의 **Fiber 객체** = 모든 정보를 가지고 있는 가상 DOM의 노드

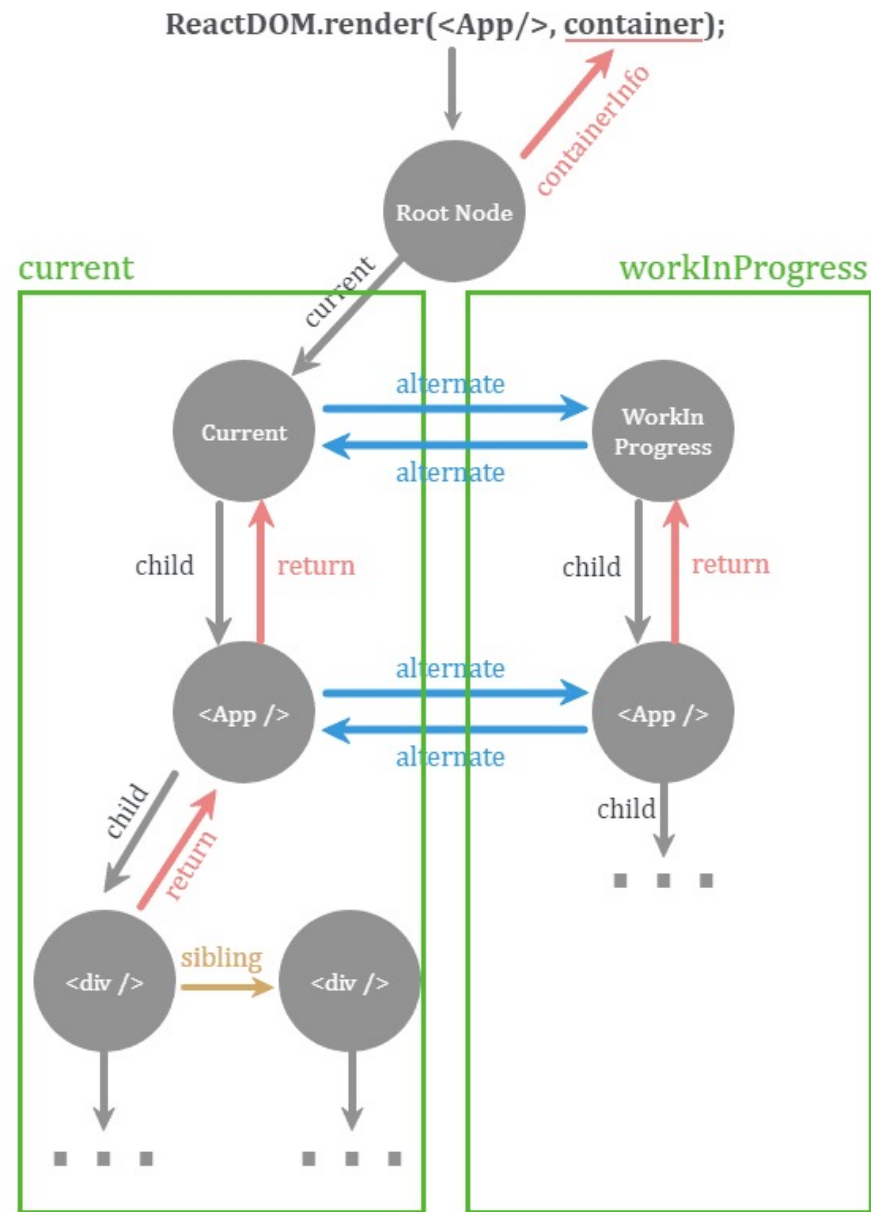
```
▼ FiberNode {tag: 2, key: null, elementType: null, type: null, stateNode: null, ...} ⓘ  
  actualDuration: 0  
  actualStartTime: -1  
  alternate: null  
  child: null  
  childLanes: 0  
  dependencies: null  
  elementType: "button"  
  firstEffect: null  
  flags: 0  
  index: 0  
  key: null  
  lanes: 1  
  lastEffect: null  
  memoizedProps: null  
  memoizedState: null  
  mode: 0  
  nextEffect: null  
  ▶ pendingProps: {children: {...}, onClick: f}  
  ref: null  
  ▶ return: FiberNode {tag: 1, key: null, stateNode: LikeButton, elementType: f, type: f, ...}  
  selfBaseDuration: 0  
  sibling: null  
  stateNode: null  
  tag: 5  
  treeBaseDuration: 0  
  type: "button"  
  updateQueue: null
```



2. 용어 정리

실제로는 2개의 Fiber Tree가 존재합니다.
(Double Buffering)

Virtual DOM



플로우 분석

3. 플로우 분석

1. 개발자가 작성한 컴포넌트가 어떻게 실제 브라우저의 DOM에 마운트(반영)되는가?
2. setState 등의 hook으로 일어난 상태변화가 어떻게 반영되는가?

3. 플로우 분석

React 앱 실행 플로우는 2단계의 phase를 가집니다.

2 phase

phases

Phase 1

render / reconciliation

interruptible

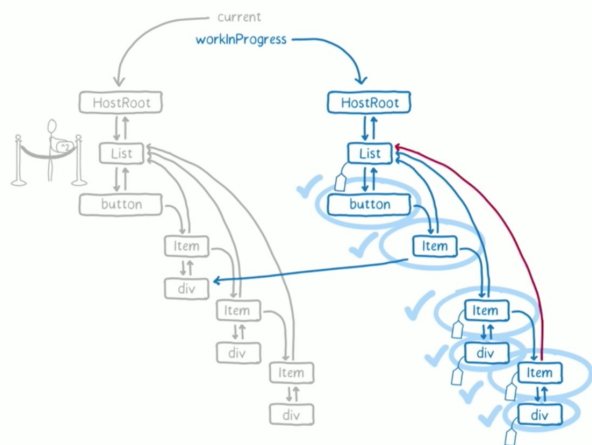
Phase 2

commit

not interruptible

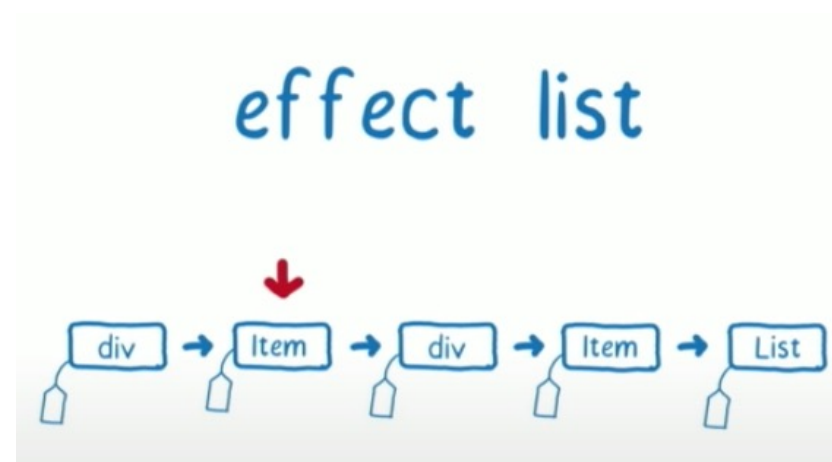
3. 플로우 분석

Phase 1 - Render Phase



Update가 반영된 fiber tree (**workInProgressTree**) 와 **effect list**를 만드는 과정

Phase 2 - Commit Phase



Effect list가 모두 소비되고, workInProgressTree가 currentTree가 된다.

실행 플로우 #1

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
```

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

entry point

App.js

```
33
34 function App() {
35   const [state, setState] = useState({ liked: 0 });
36   return (
37     <>
38       <button onClick={() => setState((state) => state.liked + 1)}>
39         <span>`liked ${state} time(s)`</span>
40         <SomeComponent />
41       </button>
42     </>
43   );
44 }
45
```

3. 플로우 분석

```
분석1. ReactDOM.render
분석2. legacyRenderSubtreeIntoContainer
분석2-1. legacyCreateRootFromDOMContainer
분석2-2. createLegacyRoot
분석2-3. ReactDOMBlockingRoot
분석2-4. createRootImpl
분석2-5. createContainer
분석2-6. createFiberRoot
분석2-7. FiberRootNode
분석2-8. createHostRootFiber
분석2-9. createFiber > FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: null, ...}
분석2-11. initializeUpdateQueue
분석3. unbatchedUpdates
분석4. updateContainer
분석4-1. createUpdate
분석4-2. enqueueUpdate
> {baseState: null, firstBaseUpdate: null, lastBaseUpdate: null, shared: {...}, effects: null}
분석4-3. scheduleUpdateOnFiber
분석4-4. performSyncWorkOnRoot
분석4-5. renderRootSync
분석4-14. createWorkInProgress
> FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, ...} null
분석2-9. createFiber > FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: null, ...}
분석4-6. workLoopSync
분석4-7. performUnitOfWork
분석2-9. createFiber > FiberNode {tag: 2, key: null, elementType: null, type: null, stateNode: null, ...}
분석4-9. beginWork > FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, ...}
null
current > FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, ...}
분석4-10. updateHostRoot
분석4-9-2. processUpdateQueue
분석4-11. reconcileChildren
분석4-14. createFiberFromElement
분석4-15. createFiberFromTypeAndProps
분석2-9. createFiber > FiberNode {tag: 1, key: null, elementType: null, type: null, stateNode: null, ...}
분석4-7. performUnitOfWork
분석2-9. createFiber > FiberNode {tag: 2, key: null, elementType: null, type: null, stateNode: null, ...}
분석4-9. beginWork null class LikeButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = { liked: 0 };
  }

  render() {
    console.log("분석. RENDER!")
    return e(
      -
    )
  }
}
current null
분석4-9-0. updateClassComponent
분석2-11. initializeUpdateQueue
분석4-9-2. processUpdateQueue
분석4-9-3. finishClassComponent
분석. RENDER!
분석4-11. reconcileChildren
분석4-14. createFiberFromElement
분석4-15. createFiberFromTypeAndProps
분석2-9. createFiber > FiberNode {tag: 5, key: null, elementType: null, type: null, stateNode: null, ...}
분석4-7. performUnitOfWork
분석2-9. createFiber > FiberNode {tag: 2, key: null, elementType: null, type: null, stateNode: null, ...}
분석4-9. beginWork null button
current null
분석4-11. reconcileChildren
분석4-14. createFiberFromElement
분석4-15. createFiberFromTypeAndProps
분석2-9. createFiber > FiberNode {tag: 5, key: null, elementType: null, type: null, stateNode: null, ...}
분석4-7. performUnitOfWork
분석2-9. createFiber > FiberNode {tag: 2, key: null, elementType: null, type: null, stateNode: null, ...}
분석4-9. beginWork null span
current null
분석4-11. reconcileChildren
분석4-16. completeUnitOfWork
분석4-17. completeWork > FiberNode {tag: 5, key: null, elementType: "span", type: "span", stateNode: null, ...}
분석5. createInstance
<span>liked 0 time(s)</span>
분석6. appendAllChildren
분석4-17. completeWork > FiberNode {tag: 5, key: null, elementType: "button", type: "button", stateNode: null, ...}
분석5. createInstance
> <button>...</button>
분석6. appendAllChildren
분석4-17. completeWork > FiberNode {tag: 1, key: null, stateNode: LikeButton, elementType: f, type: f, ...}
분석4-17. completeWork
> FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, ...}
분석5. commitRoot
```

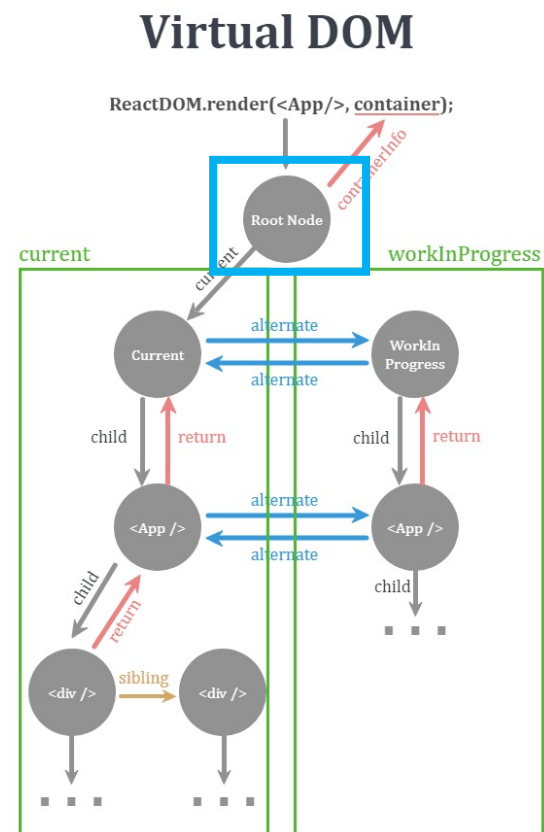


3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

1. 개발자가 작성한 Root 컴포넌트에 대응하는 element 생성 (ReactDOM.render, React.createElement)
2. Container element로부터 FiberRoot 생성 (#root)
3. workInProgress 트리 생성 시작

```
▼ FiberRootNode {tag: 0, containerInfo: div#root, pendingCallbackNode: null, current: FiberNode, pingCache: null, ...}
  callbackNode: null
  callbackPriority: 0
  ▶ containerInfo: div#root
  ▶ context: {}
  ▶ current: FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, ...}
    entangledLanes: 0
  ▶ entanglements: (31) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ▶ eventTimes: (31) [-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ▶ expirationTimes: (31) [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
    expiredLanes: 0
    finishedLanes: 0
    finishedWork: null
    hydrate: false
    interactionThreadID: 1
    memoizedInteractions: Set(0) {}
    mutableReadLanes: 0
    mutableSourceEagerHydrationData: null
    pendingChildren: null
    pendingContext: null
  ▶ pendingInteractionMap: Map(0) {}
    pendingLanes: 0
    pingCache: null
    pingedLanes: 0
    suspendedLanes: 0
    tag: 0
    timeoutHandle: -1
    _debugRootType: "createLegacyRoot()"
  __proto__: Object
```

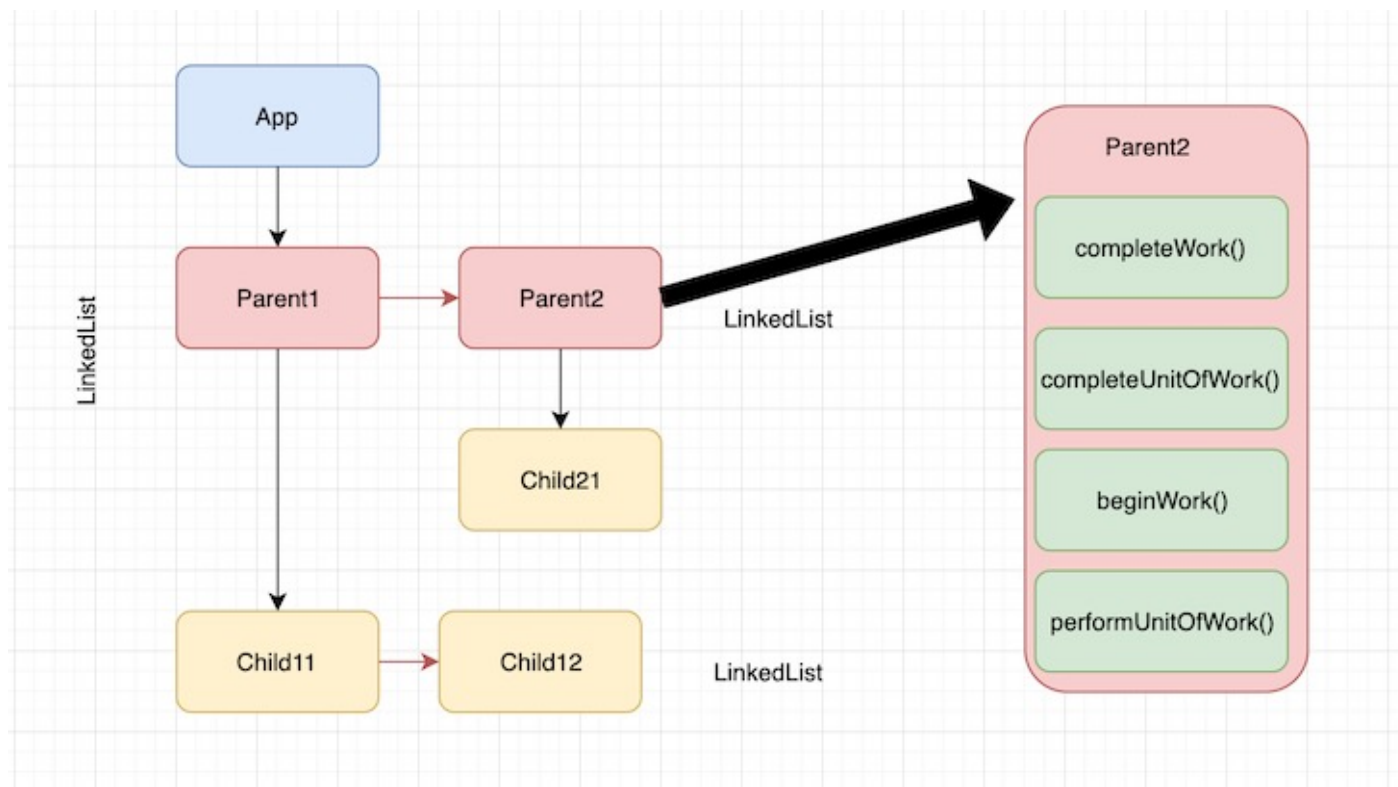


3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

4. 모든 fiberNode는 **workLoopSync**을 통해 작업이 처리됨

```
function workLoopSync() {  
  // Already timed out, so perform work w  
  while (workInProgress !== null) {  
    performUnitOfWork(workInProgress);  
  }  
}
```



3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

4. 모든 fiberNode는 **workLoopSync**을 통해 작업이 처리됨

```
function workLoopSync() {  
  // Already timed out, so perform work w  
  while (workInProgress !== null) {  
    performUnitOfWork(workInProgress);  
  }  
}
```

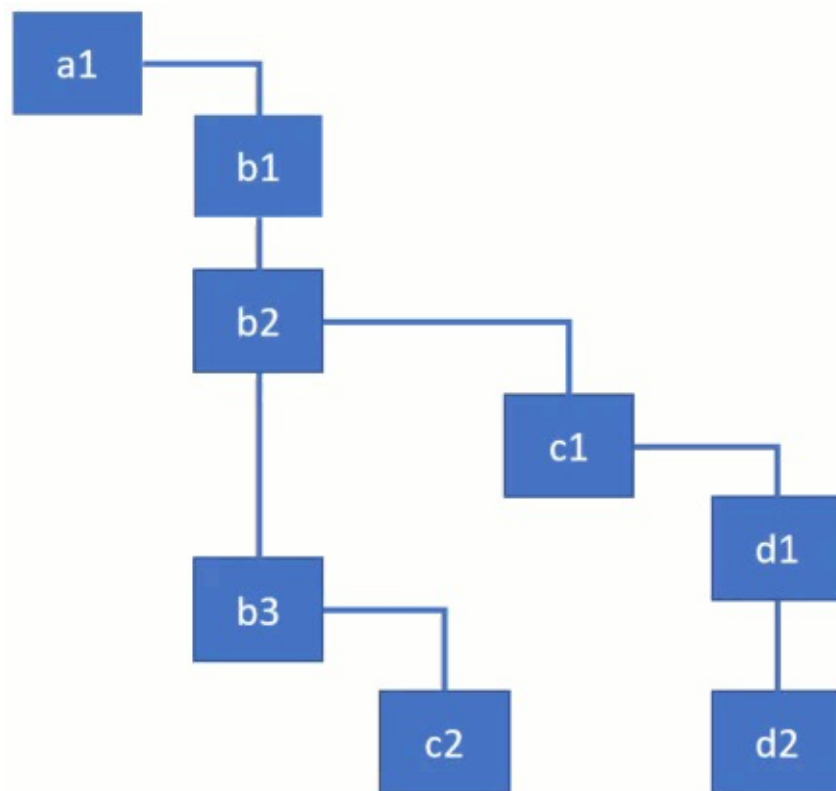
```
function performUnitOfWork(workInProgress) {  
  let next = beginWork(workInProgress);  
  if (next === null) {  
    next = completeUnitOfWork(workInProgress);  
  }  
  return next;  
}  
  
function beginWork(workInProgress) {  
  console.log('work performed for ' + workInProgress.name);  
  return workInProgress.child;  
}
```

```
function completeUnitOfWork(workInProgress) {  
  while (true) {  
    let returnFiber = workInProgress.return;  
    let siblingFiber = workInProgress.sibling;  
  
    nextUnitOfWork = completeWork(workInProgress);  
  
    if (siblingFiber !== null) {  
      // If there is a sibling, return it  
      // to perform work for this sibling  
      return siblingFiber;  
    } else if (returnFiber !== null) {  
      // If there's no more work in this returnFiber,  
      // continue the loop to complete the parent.  
      workInProgress = returnFiber;  
      continue;  
    } else {  
      // We've reached the root.  
      return null;  
    }  
  }  
}  
  
function completeWork(workInProgress) {  
  console.log('work completed for ' + workInProgress.name);  
  return null;  
}
```

3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

4. 모든 fiberNode는 `workLoopSync`을 통해 작업이 처리됨



`nextUnitOfWork`
`performUnitOfWork`
`beginWork`
`completeUnitOfWork`
`completeWork`

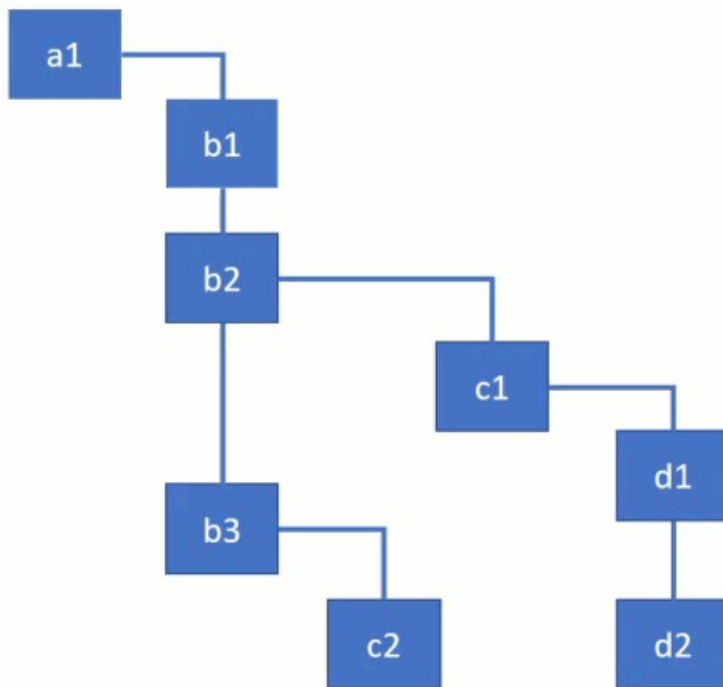
3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

4. 모든 fiberNode는 **workLoopSync**을 통해 작업이 처리됨

```
function workLoopSync() {  
  // Already timed out, so perform work w.  
  while (workInProgress !== null) {  
    performUnitOfWork(workInProgress);  
  }  
}
```

```
function performUnitOfWork(workInProgress) {  
  let next = beginWork(workInProgress);  
  if (next === null) {  
    next = completeUnitOfWork(workInProgress);  
  }  
  return next;  
}  
  
function beginWork(workInProgress) {  
  console.log('work performed for ' + workInProgress.name);  
  return workInProgress.child;  
}
```



nextUnitOfWork
performUnitOfWork
beginWork
completeUnitOfWork
completeWork

<https://vimeo.com/302222454>

3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

- 4. 모든 fiberNode는 **workLoopSync**을 통해 작업이 처리됨
- 5. 첫 마운트 시의 beginWork = fiberNode 생성
- 6. 첫 마운트 시의 completeWork = instance 생성

```
분석4-17. completeWork ▶ FiberNode {tag: 5, key: null, elementType: "span", type: "span", stateNode: null, ...}
분석5. createInstance
  <span>liked 0 time(s)</span>
분석6. appendAllChildren
분석4-17. completeWork ▶ FiberNode {tag: 5, key: null, elementType: "button", type: "button", stateNode: null, ...}
분석5. createInstance
  ▶ <button>...</button>
분석6. appendAllChildren
분석4-17. completeWork ▶ FiberNode {tag: 1, key: null, stateNode: App, elementType: f, type: f, ...}
분석4-17. completeWork
  ▶ FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, ...}
분석5. commitRoot
```

자식 노드부터 completeWork 호출
HostNode까지 complete 되면 commit phase 시작

3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

7. commitRoot에서는 effects list를 돌면서 side effects 처리, 브라우저에 paint 요청

Side effect

VDOM에 변경점을 만들거나(추가, 수정, 삭제..) 혹은 변경점을 만들어낼 수도 있는 작업(라이프 사이클)을 side effect라고 합니다.
아래는 리액트에서 사용되는 side effect tag입니다.

```
shared > ReactSideEffectTags.js
export const NoEffect = /           / 0b00000000000000;
export const PerformedWork = /      / 0b00000000000001;
export const Placement = /         / 0b00000000000010;
export const Update = /           / 0b00000000000100;
export const PlacementAndUpdate = / / 0b00000000000110;
export const Deletion = /          / 0b00000000010000;
export const ContentReset = /      / 0b00000000100000;
export const Passive = /           / 0b00010000000000;
/* ... */
```

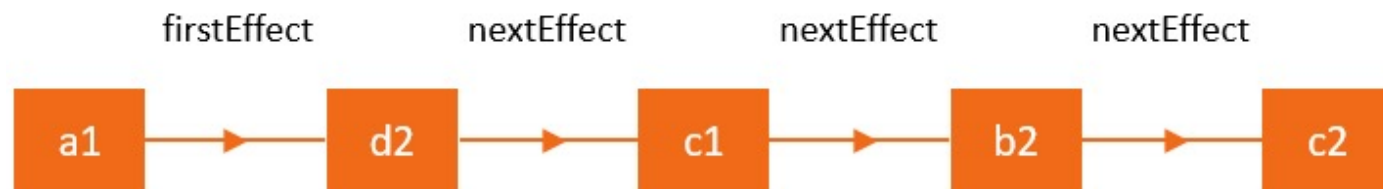
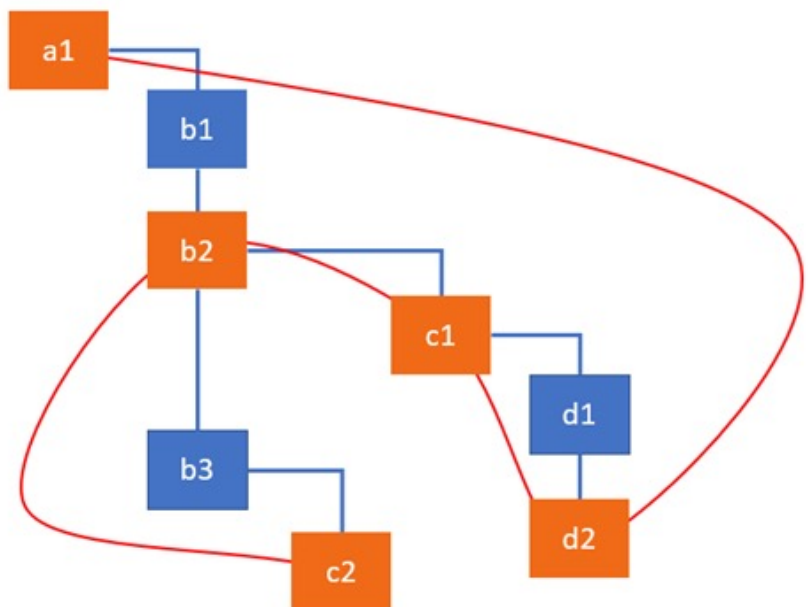
이 tag는 fiber의 `effectTag`에 저장됩니다. 해당 tag가 달린 fiber는 effect로서 취급되며 연결 리스트로 상위로 엮여 올라 갑니다. 최종적으로 최상위 fiber가 하위 모든 effect를 가지고 있게 되며 이는 Commit phase에서 소비됩니다.

```
{
  update
  {
    effectTag: 4,
    elementType: class ClickCounter,
    firstEffect: null,
    memoizedState: {count: 1},
    type: class ClickCounter,
    stateNode: {
      state: {count: 1}
    },
    updateQueue: {
      baseState: {count: 1},
      firstUpdate: null,
      ...
    }
  }
}
```


3. 플로우 분석

개발자가 작성한 컴포넌트가 어떻게
실제 브라우저의 DOM에 마운트(반영)되는가?

7. commitRoot에서는 effects list를 돌면서 side effects 처리, 브라우저에 paint 요청



실행 플로우 #2

setState 등의 hook으로 일어난
상태변화가 어떻게 반영되는가?

3. 플로우 분석

setState 등의 hook으로 일어난
상태변화가 어떻게 반영되는가?

1. 이벤트 핸들러 등으로 hook 호출
2. update 객체를 만들고, Fiber 노드의 updateQueue에 삽입

```
33
34 √ function App() {
35   const [state, setState] = useState({ liked: 0 });
36   return (
37     √
38     √ <button onClick={() => setState((state) => state.liked + 1)}>
39       <span>`liked ${state} time(s)`</span>
40       <SomeComponent />
41     </button>
42   </>
43 );
44 }
45
```

```
분석4-1. createUpdate
분석4-2. enqueueUpdate
▶ {baseState: {...}, firstBaseUpdate: null, lastBaseUpdate: null, shared: {...}, effects: null}

▼ FiberNode {tag: 1, key: null, stateNode: App, elementType: f, type: f, ...} 1
  actualDuration: 0
  actualStartTime: -1
  ▶ alternate: FiberNode {tag: 1, key: null, stateNode: App, elementType: f, type: f, ...}
  ▶ child: FiberNode {tag: 5, key: null, elementType: "button", type: "button", stateNode: button, ...}
  childLanes: 0
  dependencies: null
  ▶ elementType: class App
  firstEffect: null
  flags: 1
  index: 0
  key: null
  lanes: 1
  lastEffect: null
  ▶ memoizedProps: {}
  ▶ memoizedState: {liked: 0}
  mode: 0
  nextEffect: null
  ▶ pendingProps: {}
  ref: null
  ▶ return: FiberNode {tag: 3, key: null, elementType: null, type: null, stateNode: FiberRootNode, ...}
  selfBaseDuration: 0
  sibling: null
  ▶ stateNode: App {props: {...}, context: {...}, refs: {...}, updater: {...}, state: {...}, ...}
  tag: 1
  treeBaseDuration: 0
  type: class App

▼ updateQueue:
  ▶ baseState: {liked: 0}
  effects: null
  ▼ firstBaseUpdate:
    callback: null
    eventTime: 286.7000000178814
    lane: 1
    next: null
    ▶ payload: state => ({liked: this.state.liked + 1})
    tag: 0
    ▶ __proto__: Object
  ▶ lastBaseUpdate: {eventTime: 286.7000000178814, lane: 1, tag: 0, callback: null, payload: f, ...}
```

3. 플로우 분석

setState 등의 hook으로 일어난
상태변화가 어떻게 반영되는가?

3. workLoop 수행 과정에서 updateQueue의
update 들이 소비되고, fiber 객체가 업데이트 됨
4. 이후 1번 플로우와 동일

```
{
  effectTag: 0,
  elementType: class ClickCounter,
  firstEffect: null,
  memoizedState: {count: 0},
  type: class ClickCounter,
  stateNode: {
    state: {count: 0}
  },
  updateQueue: {
    baseState: {count: 0},
    firstUpdate: {
      next: {
        payload: (state, props) => {...}
      }
    },
    ...
  }
}
```

work 수행 전 FiberNode

```
{
  update
  effectTag: 4,
  elementType: class ClickCounter,
  firstEffect: null,
  memoizedState: {count: 1},
  type: class ClickCounter,
  stateNode: {
    state: {count: 1}
  },
  updateQueue: {
    baseState: {count: 1},
    firstUpdate: null,
    ...
  }
}
```

work 수행 후 FiberNode

E.O.D

감사합니다 🙏



Sebastian Markbåge

2016년 6월 17일 · 🌐



How React Fiber Works

Basically the idea of React Fiber is to not use the JavaScript stack but instead unroll what we would normally do on the stack and put it into heap objects.

This is inspired by KC's work on OCaml's concurrency.



Reason @reasonml · 2017년 8월 18일



Oh btw, React Fiber's inspiration came from OCaml



Sebastian Markbåge @sebmarkbage · 2017년 8월 18일

@sebmarkbage @jlongster 님, 다른 사람 2명에게 보내는 답글

And OCaml's strategy around algebraic effects with delimited continuations give you opt-out regions where you can regain safety.