[React-deep-dive]-2장.리액트 핵심요소 깊게 살펴보기

리액트 핵심요소 깊게 살펴보기

결론: jsx는 반드시 트랜스파일러를 거쳐야한다.

jsx 내부에 트리 구조로 표현하고 싶은 다양한 것 들을 작성 → 트랜스파일 과정을 거쳐 자바스크립트가 일할수있는 코드로 변경하는 것이 목표라고 할수있다

JSX의 정의

JSXElement

jsx를 구성하는 가장 기본적인 요소, **html의 요소**와 비슷한 역할

- jsxElement가 되기위해서는, 아래의 형태중 하나

1. JSXOpeningElement
2. JSXClosingElement:
3. JSXSelfClosingElement :
4. Fragment : <></>

JSXElementName

결론: 요소의 이름으로 쓸수 있는 것은 아래와 같음

- **JSXIdentifier**
  - JSX내부에서 사용할수있는 식별자 → 자바스크립트 식별자 규칙과 동일

    → *숫자로 시작 불가 ex) <1></1> / 특수문자는 $, _ 만 가능*
- **JSX Namespaced Name (리액트에서는 사용x)**
  - :을 통해 서로 다른 식별자를 이어주는 것

    → :로는 하나만 묶을수잇음 `<foo:bar></foo:bar>`
- **JSXMemberExpression**
  - . 을 통해 서로 다른 식별자를 이어주는것 , 하나이상 가능 `<foo.bar.baz></foo.bar.baz>`

JSXAttributes

JSXElement 에 부여할 수 있는 속성 (필수값아님)

- JSXSpreadAttributes
  - 자바스크립트의 전개 연산자와 동일한 역할
  - {...AssignmentExpression} → 객체뿐아니라 자바스크립트에서 취급되는 모든 표현식이 가능
- JSXAttribute
  - 속성을 나타내는 키와 값으로 짝을 이루어서 표현
  - 키: AttributeName 값 : AttributeValue `<element name={value}/>`

JSXChildren

JSXElement의 자식값

- JSXChild
  - JSXChildren을 이루는 기본 단위
  - JSXText , JSXElement, JSXFragment , JSX
- JSXStrings
  - “” , ‘’ 로 구성된 문자열

위 내용을 바탕으로한 JSX예제 책 참고~ 122p

자바스크립트에서의 JSX 변환

- JSXElement를 첫번째 인수로 선언해 요소를 정의한다 → **활용**
- 옵셔널인 JSXChildren, JSXAttributes ,JSXStrings는 이후 인수로 넘겨주어 처리한다

위의 특성활용

이점을 활용하면 경우에따라 다른 요소를 렌더링 해야할때 굳이 요소전체를 감싸지 않더라도 처리할수 있다.

```
function Test ({isTest , children }) {
return isTest? (<div>children</div>) : (<button>children</button>)
}
//isTest 유무에 따라 삼항연산자 사용

function Test ({isTest , children }) {

return createEelement (
   isTest ? 'div' : 'button',
   {attribute},
```

```
    children,
})
```

결론: 트랜스 파일 과정에서 JSX 반환값은 결국 React.creaeteElement로 귀결 된다.

가상DOM 과 리액트 파이버

DOM과 브라우져 렌더링과정

(HTML파싱 → DOM) + (CSS파싱→ CSSOM) = Render Tree → 레이아웃, 페인팅 거쳐서 렌더링

가상 DOM

결론 : 가상돔은 웹페이지가표시해야할 DOM 을 일단 메모리에 저장하고 리액트가 실제변경에 대한 준비가 완료 됐을때 실제 브라우져의 DOM에 반영

리액트파이버

가상돔과 렌더링 과정 최적화를 가능하게 해주는 리액트가 관리하는 자바스크립트 객체

파이버

- 하나의 작업 단위로 구성
- 작업을 작은 단위로 분할하고 쪼갠 다음, 운선순위를 매긴다
- 이러한 작업을 일시중지하고 나중에 다시 저장 할 수 있다
- 이전에 했던 작업을 다시 재사용 하거나 필요하지 않은경우에는 폐기

- 위의 작업은 비동기로 일어남
- 컴포넌트가 최초로 마운트 되는 시점에 생성되어 가급적 재사용됨 (리액트 요소는 렌더링이 발생 할때마다 생성)
- state가 변경되거나, 생명주기 메서드가 실행되거나, DOM의 변경이 필요한 시점등에 실행됨 →유연하게 처리됨

파이버 재조정자

- 가상돔 - 실제돔 을 비교 (재조정) 해 변경사항을 수집
- 둘 사이에 차이가 있으면 변경에 관련된 정보를 가진 파이버를 기준으로 화면에 렌더링을 요청

리액트 파이버 트리

결론 : 파이버 트리는 리액트 내부에서 두개가 존재

Current Tree

workInProgress Tree

- **더블 버퍼링** : 리액트 파이버의 작업이 끝나면 리액트는 단순히 포인터만 변경해 workInProgress Tree를 current 트리로 바꿔버림.
- workInProgress 를 빌드 하는 작업이 끝나면 다음 렌더링에 이트리를 사용한다
- 이 트리가 UI에 최종적으로 렌더링 되어 반영이 완료 되면 current가 이 workInProgress 로 변경된다

파이버의 작업순서

일반적인 파이버 노드의 생성 흐름

1. 리액트는 beginwork( ) 함수를 실행해 파이버 작업을 수행하는데, 더이상 자식이 없는 파이버를 만날 때 까지 트리 형식으로 시작된다.
2. 그다음 completeWork( )함수를 실행해 파이버 작업을 완료한다

3. 형제가 있다면 형제로 넘어간다
4. 모두 끝나면 자신의 작업이 완료 되었음 알린다.

state 변경등 업데이트가 발생하면 이미 만든 current 트리가 존재하고, setState로 인한 업데이트 요청을 받아 workInProgress 트리를 다시 빌드 (위의 과정과 동일)

최초 렌더링 시에는 모든 파이버를 새롭게 만들어야 했지만 이제는 파이버가 이미 존재 하므로 되도록 새로 생성하지 않고 기존 파이버에서 업데이트된 props를 받아 파이버 내부에서 처리한다.

가급적 객체를 새롭게 만들기보다는 기존에있는 객체를 재활용하기 위해 내부속성값만 초기화하거나 바꾸는 형태로 트리를 업데이트 한다.

클래스 컴포넌트와 함수 컴포넌트

클래스 컴포넌트

클래스 컴포넌트를 만들려면 클래스를 선언 → 만들고 싶은 컴포넌트를 extends

```
import React, { Component } from "react";

class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increase = () => {
    this.setState({ count: this.state.count + 1 });
  };
```

```
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increase}>Increase</button>
      </div>
    );
  }
}

export default Counter;
```

클래스 컴포넌트의 생명주기 메서드

생명주기 메서드가 실행되는 시점

- **mount**: 컴포넌트가 마운팅 되는 시점
- **update** : 이미 생성된 컴포넌트의 내용이 변경되는 시점
- **unmount** : 컴포넌트가 더이상 존재하지 않는 시점