컴포넌트를 만드는 방법

1. 클래스를 통해 만들기

class App extends Component {

render(){

return(

//JSX 코드

)

}

}

1. 함수를 통해 만들기

조건부 렌더링 방법 (if 문 사용 불가)

1. 삼항 연산자 사용 ( T/F 값 구분 가능 )

<div>

{

1 + 1 === 2

? (<div>TRUE VALUE</div>)

: (<div> FALSE VALUE </div>)

}

</div>

1. AND 연산자 사용 ( T 값만 표시 가능)

<div>

{

1 + 1 === 2 && (<div>TRUE VALUE!</div>)

}

</div>

1. IIFE 사용

<div>

{

(function() {

if(value === 1) return (<div>VALUE 1 </div>);

if(value === 2) return (<div>VALUE 2 </div>);

if(value === 3) return (<div>VALUE 3 </div>);

}) ()

}

</div>

1. 화살표 함수 사용

<div>

(() => {

if(value === 1) return (<div>VALUE 1 </div>);

if(value === 2) return (<div>VALUE 2 </div>);

if(value === 3) return (<div>VALUE 3 </div>);

})()

</div>

컴포넌트 내부 데이터

1. props

* 부모가 자식에게 주는 값.
* 자식에서 값 수정 불가.

1. state

* 컴포넌트 내부에서 선언
* 내부에서 값 수정 가능..

함수형 컴포넌트

import React from 'react';

const MyName = ({ name }) => {

return (

<div>

</div>

);

};

export default MyName;

클래스형 컴포넌트

import React, { Component } from 'react';

class MyName extends Component {

render() {

return (

<div>

</div>

);

}

}

MyName.defaultProps = {

name: '기본이름'

};

export default MyName;

동적 데이터 사용 : state

class Counter extends Component {

state = {

number: 0

}

handleIncrease = () => {

this.setState({ //this.setState() : 함수 호출 시 객체로 전달 되는 값만 리렌더링

number:state.number + 1

});

}

handleDecrease = () => {

this.setState(

({number}) => ({

Number: number – 1

})

);

}

render() {

return (

<div>

<h1>카운터</h1>

<div>값: {this.state.number}</div>

<button onClick={this.handleIncrease}>+</button>

// 이벤트에 전달해 주는 값은 ‘함수’ 이다.

// ‘함수 호출’ 전달 시 무한 루프 발생.

<button onClick={this.handleDecrease}>-</button>

</div>

);

}

}

React LifeCYcle API

1. 컴포넌트 초기 생성

* Constructor

: 컴포넌트가 새로 만들어질 때 마다

* componentWillMount

: 화면 표시 직전 호출 -> But, UNSAFE\_componentWillMount()로 변경

* componentDidMount

: 컴포넌트가 화면에 나타나게 되었을 때. (외부 라이브러리 연동, ajax 요청, DOM 속성 값 제어)

1. 컴포넌트 업데이트

* componentWillReceiveProps

: 새로운 props를 받았을 때 호출. -> UNSAFE\_componentWIllReceiveProps()로 변경

* getDerivedStateFromProps

: props로 받아온 값을 state로 동기화하는 작업에 사용

* shouldComponentUpdate

: 컴포넌트 최적화에 사용

: false 리턴 시 해당 조건에서 render 함수 호출 안함.

* componentWillUpdate

: shouldComponentUpdate에서 true 반환 시 호출

: animation, event listener 초기화 작업 수행, 이 함수 후에 render() 호출.

* getSnapshotBeforeUpdate

: render 이후 DOM 변화 일어나기 전의 상태 저장

: componentDidUpdate 에서 3번 째 param으로 사용 가능.

* componentDidUpdate

: render() 호출 후 발생.

: props와 state 값 변경 및 변경 이전 값(prevProps, prevState) 조회 가능.

1. 컴포넌트 제거

* componentWillUnmount

: event, setTimeout, 외부 라이브러리 인스턴스 제거

1. 컴포넌트 에러 발생 시

* componentDidCatch

: state.error 를 true로 설정하게 하여, render() 함수에서 에러 처리

React 폴더구조

* action
  + App 명령어 (action type) 및 액션 메소드 (action creator)
  + 모든 명령어와 액션 메소드를 모아 두거나, 도메인 별로 구분해두기도 함.
  + 리듀서 (reducer)로 데이터 생성 요청.
  + redux-saga를 이용하 비동기 처리

// action type(명령어)

**export** **const** COMPLETE\_TODO = 'COMPLETE\_TODO'

// action creators(액션 메서드)

**export** **function** **complete**({complete, id}) {

**return** { type: COMPLETE\_TODO, complete, id};

}

‘

* component
  + React 컴포넌트로 구성되어 있으며 보통 도메인 별로 구분되어 있음.
  + 컨테이너 컴포넌트와 프레젠테이션 컴포넌트를 구분해서 개발.
    - 컨테이너 컴포넌트
      * 데이터, 공통 관리가 필요한 객체, 컴포넌트 간 인터랙션 관리
      * 다수의 프레젠테이션 컴포넌트로 구성
    - 프레젠테이션 컴포넌트
      * 일반적인 UI 컴포넌트

**class** **TODOList** **extends** **Component** {

render() {

**const** {todos, onClick} = **this**.props;

**return** (

<**ul**>

{todos.map(todo =>

<**Todo**

key={todo.id}

onClick={onClick}

{...todo}

/>

)}

</**ul**>

);

}

}

// 컨테이너 컴포넌트에서 프레젠테이션 컴포넌트로 전달하는 state

const todolistStateToProps = (state) => {

return {

todos: state.todos

}

}

// 컨테이너 컴포넌트에서 프레젠테이션 컴포넌트로 액션을 보내는 함수

const todolistDispatchToProps = (dispatch) => {

return {

onClick(data){

dispatch(complete(data)) // 액션 메서드

}

}

}

// 연결

export default connect(todolistStateToProps,todolistDispatchToProps)(TODOList);

**class** **TODO** **extends** **Component** {

render() {

**const** {id, todo, complete, onClick} = **this**.props; // 컨테이너 컴포넌트에서 받은 prop

**return** (

<**li** id={id}

onClick={() => onClick({

id : id,

complete : !complete

})}

className={!!complete ? 'completed' : ''}

>{todo}</**li**>

);

}

}

* reducer
  + 리듀서 (reducer)로 구성.
    - 리듀서 (reducer)
      * 액션 메서드에서 변경한 상태를 받아 기존 상태를 최신화.
      * Input state와 output state가 항상 같은 순수 함수로 구현.
  + 하나로 구성하기도 하지만 도메인 별로 구분하기도 함.
  + Action 파일 + reducer 파일 = ducks 기법

**import** todoAction **from** '../action/index';

**const** {ADD\_TODO} = todoAction.todo;

**const** todo = (state, action) => {

**switch** (action.type) {

**case** ADD\_TODO:

**return** {

text: action.text,

completed: false

};

**default**:

**return** state;

}

}

**const** todos = (state = [], action) => {

**switch** (action.type) {

**case** ADD\_TODO:

**return** [

...state, todo(undefined, action)

];

**default**:

**return** state;

}

}

**export** **default** combineReducers({

todos

});

* store
  + index.js 파일 하나만 존재.
  + 미들웨어 설정

**import** { createStore, compose, applyMiddleware } **from** "redux";

**import** thunk **from** "redux-thunk";

**export** **default** **function** **configureStore**(reducer, initialState = {}) {

**const** storeEnhancers = compose(

applyMiddleware(thunk)

);

**return** createStore(reducer, initialState, storeEnhancers);

}