



리액트와 함께 장고 시작하기 / 리액트

리듀서와 useReducer 훅

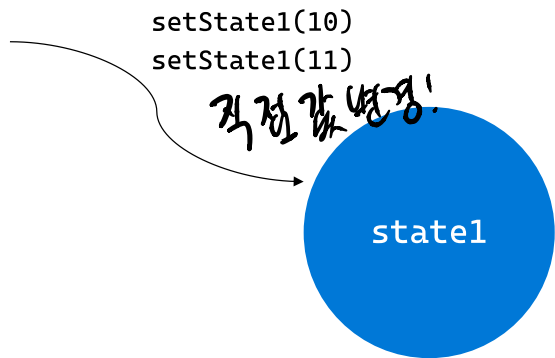
여러분의 파이썬/장고 페이스메이커가 되겠습니다.

상태값 자체가 정수처럼 단순한 경우가 아니라면
⇒ 객체 상태값을 변경하는 setter에도 루직이 존재할 수 있다.

리액트에서 상태값을 변경하는 2가지 방법

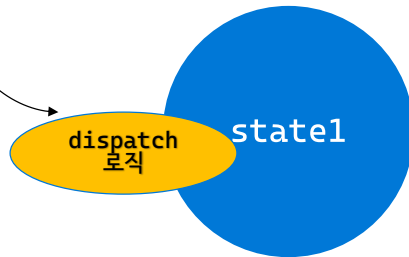
1. setter 함수를 직접 제공 → setter 로직이 다수 혼재
 ⇒ 개별 setter를 사용하는 것들에서 로직들이 중복될 수 있다
2. setter 함수를 제공하지 않고, **dispatch** 함수를 제공 → Redux에서의 방식

setter 로직을 dispatch를 통해서만 처리



→ 객체를 넘겨준다

dispatch({ type: 'INCREMENT', payload: 1 })
dispatch({ type: 'INCREMENT', payload: 1 })



useReducer **혹** 우리가 직접으로 관리하는 게 아니라

컴포넌트의 상태를 리덕스의 리듀서처럼 관리 가능
dispatch를 하게 되면 내부적으로 reducer 로컬!

∴ 상태값 세팅!
(object를 반환)

```
const INITIAL_STATE = { name: 'Tom', age: 10 };
```

```
const reducer = (state, action) => {  
  switch (action.type) {  
    case 'setName': return { ...state, name: action.name };  
    case 'setAge': return { ...state, age: action.age };  
    default: return state;  
  }  
};
```

Context API를 통해,
깊은 트리의 자식 컴포넌트에
dispatch 전달을 할 수 있겠다.

```
const Person = () => {  
  const [state, dispatch] = React.useReducer(reducer, INITIAL_STATE);  
  return (  
    <div>  
      <p>name : {state.name}</p>  
      <p>age: {state.age}</p>  
      <input type="text" value={state.name} onChange={e => dispatch({ type: 'setName', name: e.currentTarget.value })} />  
      <input type="text" value={state.age} onChange={e => dispatch({ type: 'setAge', age: e.currentTarget.value })} />  
    </div>  
  );  
};
```

useReducer가 관리하는
상태값 object

dispatch로부터 **action 객체**를 받아서
상태값에 대한 setter 로직을 구현 가능하!
= 구조화

참고) Redux에서의 dispatch 함수 생성

```
import { handleActions } from "redux-actions";
```

```
const SET_NAME = 'setName';
```

```
const SET_AGE = 'setAge';
```

```
const actions = handleActions({
```

```
  [SET_NAME]: (state, { payload: name }) =>
```

```
    produce(state, (draft) => {
```

```
      draft.name = name;
```

```
    }),
```

```
  [SET_AGE]: (state, { payload: age }) => {
```

```
    produce(state, (draft) => {
```

```
      draft.age = age;
```

```
    });
```

```
  },
```

```
});
```

dispatch 로컬과, reducer 함수내에서도 접근하게 때문에

문자열로 주어진다면 오타의 확률이 낮아서 상수로 논리를 선언

Life is short.
You need Python and Django.

I will be your pacemaker.



Ask Company



리액트와 함께 장고 시작하기 / 리액트

Context API

여러분의 파이썬/장고 페이스메이커가 되겠습니다.

Context API의 필요성

여러 단계에 걸쳐, 하위 컴포넌트로 속성값을 전달할 때에는, 각 단계별로 속성값을 전달하는 코드를 기계적/반복적으로 작성해야하는 번거로움

```
const App = () => <Level1 message="Hello Context API" />;
```

```
const Level1 = ({ message }) => (
```

객체, Array, state 등을 전달

```
<div>
```

```
  Level1
```

```
  <Level2 message={message} />
```

```
</div>
```

```
);
```

기계적인 속성값 전달

```
const Level2 = ({ message }) => (
```

```
<div>
```

```
  Level2
```

```
  <Level3 message={message} />
```

```
</div>
```

```
);
```

```
const Level3 = ({ message }) => <div>Level3 :  
{message}</div>;
```

* 계속 상위 단계의 컴포넌드로
속성값을 전달해야 해서 귀찮음!

* 공간에 끼맞는 컴포넌드가
해당 속성값을 필요로 하지 않는다면 비효율적!

∴ 이런 단점들을 해결할 수 있는 게 Context API

Context API 활용 예시

Level1/Level2 컴포넌트가 중간에 개입하지 않고서도, 값을 전달 가능

// Provider/Consumer 속성을 가집니다.

```
const MessageContext = React.createContext('default message');
```

```
const App = () => (  
  <MessageContext.Provider value="Hello Context API">  
    <Level1 />  
  </MessageContext.Provider>  
);
```

해당 value가 변경될 경우,
하위 모든 컴포넌트는 다시 렌더링
(하위 컴포넌트가 shouldComponentUpdate가 false를 반환하더라도)

Consumer에서 Provider 찾기에
실패했을 경우에 사용될 기본값

→ 문자열, 숫자, object...
전달하려고 하는 value

상위로 올라가며, 가까운 Provider를 찾습니다.
관련 Provider가 없을 경우, 기본값을 사용

```
const Level1 = () => (  
  <div>  
    Level1  
    <Level2 />  
  </div>  
)
```

```
const Level2 = () => (  
  <div>  
    Level2  
    <Level3 />  
  </div>  
)
```

```
const Level3 = () => (  
  <div>  
    <MessageContext.Consumer>  
      {message => `Level 3: ${message}`}  
    </MessageContext.Consumer>  
  </div>  
)
```

Consumer내의 children은
필히 함수로만 적용해야만, 값을 받아올 수 있습니다.

provider가 전달하려고
하는 값을 함수의 인자로 전달!
⇒ 반환값으로 element
인자를 하나 받는 함수

Context 객체를 중첩하기

아래 코드는 render에서만 사용 가능 → 다른 생명주기 메서드에서는 사용 불가

```
const AlertContext = React.createContext(null);  
const MessageContext = React.createContext('default message'); // Provider/Consumer를 반환
```

```
const Level3 = () => (  
  <div>  
    <AlertContext.Consumer>  
      {alert => (  
        <MessageContext.Consumer>  
          {message => `Level 3: ${alert}, ${message}`}  
        </MessageContext.Consumer>  
      )}  
    </AlertContext.Consumer>  
  </div>  
);  
  
// Level1, Level2 생략
```

“사용이 너무 번거롭다 지기”

2개의 context 객체 중점!

```
const App = () => (  
  <AlertContext.Provider value="Alert Message">  
    <MessageContext.Provider value="Hello Context API">  
      <Level1 />  
    </MessageContext.Provider>  
  </AlertContext.Provider>  
);
```

다른 컴포넌트 메서드에서 다수의 Context 접근하기 (1/2)

고차 컴포넌트를 활용하여, 속성값(props)으로 전달토록 구성 가능

```
const Level1 = () => (  
  <div>  
    Level1  
    <Level2Wrapper />  
  </div>  
);
```

```
const Level2Wrapper = (props) => (  
  <AlertContext.Consumer>  
    {alert => (  
      <MessageContext.Consumer>  
        {message => (  
          <Level2  
            {...props}  
            alert={alert}  
            message={message} />  
          )}  
        </MessageContext.Consumer>  
      )}  
    </AlertContext.Consumer>  
  );
```

```
const App = () => (  
  <AlertContext.Provider value="Alert Message">  
    <MessageContext.Provider value="Hello Context API">  
      <Level1 />  
    </MessageContext.Provider>  
  </AlertContext.Provider>  
);
```

```
class Level2 extends React.Component  
{  
  render() {  
    const { alert, message } =  
    this.props;  
    return (  
      <div>  
        Level 2: {alert}, {message}  
      </div>  
    );  
  }  
}
```

* Level2 컴포넌트를
프래그멘테이션 컴포넌트의 형태로 구현

다른 컴포넌트 메서드에서 다수의 Context 접근하기 (2/2)

함수형 컴포넌트에서는 useContext 훅을 통해, Consumer를 보다 간결하게 처리

```
const Level1 = () => (  
  <div>  
    Level1  
    <Level2 />  
  </div>  
);
```

```
const Level2 = () => {  
  const alert = React.useContext(AlertContext);  
  const message = React.useContext(MessageContext);  
  return (  
    <div>  
      Level 2: {alert}, {message}  
    </div>  
  );  
};
```

* useContext(AlertContext)

이렇게 Context가 전달하려는
속성값에 바로 접근한다. → getter

* consumer를 사용하지 않아도 된다.
(provider는 연결 수 있다)

```
const App = () => (  
  <AlertContext.Provider value="Alert Message">  
    <MessageContext.Provider value="Hello Context API">  
      <Level1 />  
    </MessageContext.Provider>  
  </AlertContext.Provider>  
);
```

하위 컴포넌트에서 Context 데이터를 수정하기

Provider측의 state를 수정하는 함수를
Context로 전달하여 호출토록 구현

Redux도 유사한 방식

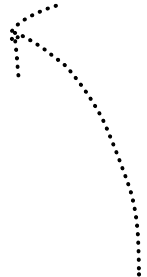
```
const CounterContext = React.createContext(null);

const Level3 = () => {
  const counter = React.useContext(CounterContext);
  return (
    <div onClick={counter.onIncrement}>
      Level 3: {counter.value}
    </div>
  );
};
```

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 0,
      onIncrement: this.onIncrement
    };
  }

  onIncrement = () => {
    const { value } = this.state;
    this.setState({
      value: value + 1
    });
  };

  render() {
    return (
      <CounterContext.Provider value={this.state}>
        <Level1 />
      </CounterContext.Provider>
    );
  }
}
```



Life is short.
You need Python and Django.

I will be your pacemaker.





리액트와 함께 장고 시작하기 / 리액트

Context API와 Reducer 패턴

여러분의 파이썬/장고 페이스메이커가 되겠습니다.

useContext 훅

Context API의 Consumer 컴포넌트와 유사한 활용

```
const PostContext = React.createContext();
```

```
const PostDetail = () => {  
  const post = React.useContext(PostContext);  
  return (  
    <div>{JSON.stringify(post)}</div>  
  );  
};
```

```
const App = () => {  
  const post = { title: "제목", content: "내용" };  
  return (  
    <PostContext.Provider value={post}>  
      <PostDetail />  
    </PostContext.Provider>  
  );  
};
```

* provider에서 값을 넘기면
- consumer 나 useContext를 통해서 전달받을 수 있다!

∴ 특정 컴포넌트 내에서만 사용할 상태값이라면
굳이 context API나 Redux 등에 태우지 않아도 된다.
→ 여러 컴포넌트 간에 서로 공유해야 할 값에 사용!!
(따라서 setter 함수는 dispatch 함수를 활용)

useReducer 훅

컴포넌트의 상태값을 리덕스의 리듀서처럼 관리 가능

```
const INITIAL_STATE = { name: 'Tom', age: 10 };
```

```
const reducer = (state, action) => {  
  switch ( action.type ) {  
    case 'setName': return { ...state, name: action.name };  
    case 'setAge': return { ...state, age: action.age };  
    default: return state;  
  }  
};
```

```
const Person = () => {  
  const [state, dispatch] = React.useReducer(reducer, INITIAL_STATE);  
  return (  
    <div>  
      <p>name : {state.name}</p>  
      <p>age: {state.age}</p>  
      <input type="text" value={state.name} onChange={e => dispatch({ type: 'setName', name: e.currentTarget.value })} />  
      <input type="text" value={state.age} onChange={e => dispatch({ type: 'setAge', age: e.currentTarget.value })} />  
    </div>  
  );  
};
```

Context API를 통해,
깊은 트리의 자식 컴포넌트에
dispatch 전달을 할 수 있겠다.

Life is short.
You need Python and Django.

I will be your pacemaker.

