USING EFFECT AND CONTEXT

030523700 : Selected Topics in Computer

Dr. Phollakrit Wongsantisuk

Reference : Application Development Book with React and React Native

Effect Response

In React, there are certain characteristics that are considered to have an effect on the component, such as first impression, re-render, change of value in State, traffic to the server, etc. In some cases, we may want to react or define certain actions when an effect occurs, but we use different methods between Function Component and Class Component, as detailed below.

- ## Effect with Function Component

In the case of Function Component, there is already a function in the React Hook group for executing when an effect occurs for us to use directly, namely ***useEffect()***.

- We may import the useEffect() function directly, or call it through the React component.
- We assign a callback to the useEffect() function in the following format:

```
useEffect(callback [, dependencies])
```

Effect Response

```
import  React, { useEffect }  from  'react'
...
function  MyComponent() {
        useEffect(function() {
                //What to do when an effect occurs
        })

        /* Or other methods such as:
        React.useEffect(() => {  //use Arrow Function
                //What to do when an effect occurs
        })
        */

        return (...)
}
```

Effect Response

- Next, we need to consider what affects the occurrence of effects or dependencies, in other words, when do we trigger the callback assigned to useEffect()?
  - If you want to call a callback when an effect occurs in all cases. (e.g. component display, re-render, Contact with the server, etc.) It does not require specifying dependencies such as:

```
React.useEffect(() => {
        //...
})
```

  - To call a callback only for the first time that the component is displayed only once. Define dependencies as empty array, such as:

```
React.useEffect(() => {
        //...
}, [] )         //Define an empty array
```

Effect Response

- To call a callback when a specific State variable value changes, specify the names of those variables in the array (more than one can be defined), such as:

```
let [a, setA] = React.useState()
let [b, setB] = React.useState()

React.useEffect(() => {
        //...
}, [a, b]) //Call the callback when A or B changes.
```

- The callback function is triggered after the component is rendered, including re-render.

Effect Response

- ## Effect with Class Component

In the case of Class Component, there is no method for directly manipulating the effect, but it is still equivalent in nature, that is, the componentDidMount() and/or componentDidUpdate() methods, as the following principles.

- If we're going to do that, ***just once***. After the component is rendered, override or define the action in the componentDidMount() method.

```
class  MyComponent  extends  React.Component {
        ...
        render() {
                return ...
        }
        componentDidMount() {
                //Define actions after the first component renders
        }
}
```

Effect Response

- If we are only going to do that when the component has already re-rendered (usually due to a change in the value in the state), then override the componentDidUpdate() method.

```
class MyComponent extends React.Component {
        ...
        render() {
                return ...
        }

        componentDidUpdate() {
                //Define actions after component re-render
        }
}
```

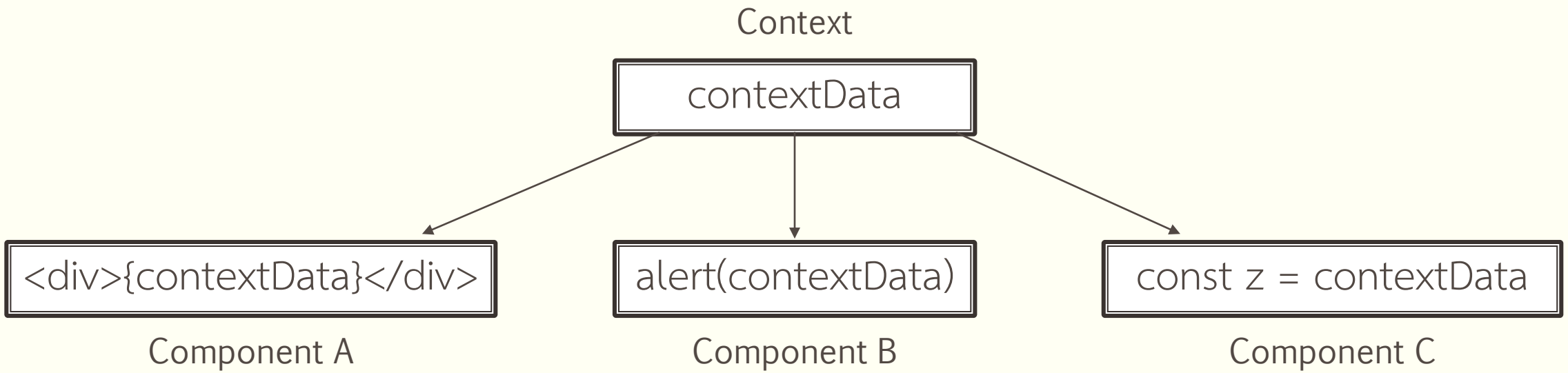
Effect Response

- If we are going to do that, both when the component is first rendered and if the component is re-rendered, then the override is shared by both the componentDidMount() and componentDidUpdate() methods.

```
class  MyComponent  extends  React.Component {
        ...
        render() {
                return ...
        }

        componentDidMount() {
                //Define actions after the first component renders
        }
        componentDidUpdate() {
                //Define actions after component re-render
        }
}
```

Centralized storage with Context

Creating a react web page brings together a bunch of individual components. For that reason, there may be cases where certain data needs to be shared between components. However, we cannot directly refer to intercomponent address information. Instead, it requires a centralized data creation method called Context so that different components can be interoperable. As in principle in the next image.



Centralized storage with Context

- ## Create and define a Provider for a Context

Since context data must be shared between components, we should create it as a separate file as a module and then import it into the component you want to use, and the configuration for that context must be done through the provider as follows:

- Create an add-in file, such as context.js. The variable that stores the context data must be configured with the createContext() function, which can be imported directly or called through the React component.

react/app1/src/context.js

```
import  React, { createContext }  from  'react'


export  const  userContext = createContext()  //or React.createContext()
/* Because it's a variable, it can't be exported to default. */
```

Centralized storage with Context

- The created context variables or components have a provider property to use to supply or configure the context, and most of the time, we place the provider in the App.js file by importing the context from the created file. Then encapsulate all the components that will apply the value from the context to the provider. For example, let's say we have two components in an app, Header and Content, and we encapsulate the Provider and configure it through the value property as follows:

react/app1/src/App.js

```
import React from 'react'
import { userContext } from './context'      //Import Context
import Header from './context-header'
import Content from './context-content'
```

```
export default function App(){
    return (
        <userContext.Provider value={'Tom Jerry'}>
            <Header/>
            <Content/>
        </userContext.Provider>
    )
}
```

From the code, Tom Jerry is the value assigned to the context, which is then applied to the components.

Centralized storage with Context

- ## Using Context Data in Class Component

Components that will apply values from Context Must be encapsulated by the Provider (As in the App.js). If it is a Class Component, accessing the value in Context is as follows:

- Import a variable or component of a context that has already been created.
- Context variables must be assigned to a static contextType property that already belongs to the React Component, such as:

```
import { userContext } from ‘./context’
export default class Header extends React.Component {
      static contextType = userContext
      …
}
```

- To refer to a value from a context, read it from a property named context, such as this.context, so the overall appearance of the Class Component in the case of using a value from Context is as follows.

Centralized storage with Context

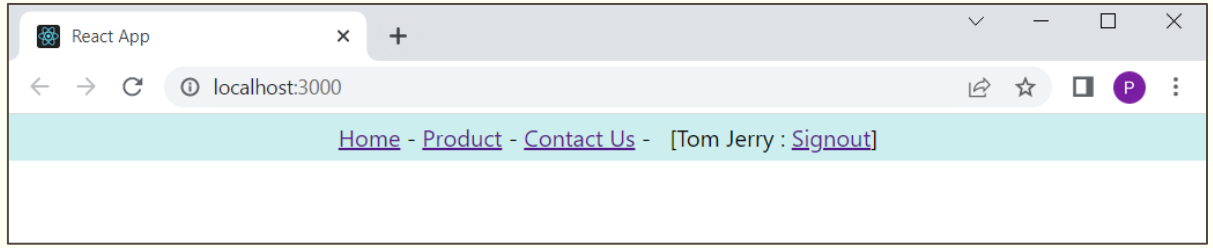
react/app1/src/context-header.js

```
import React from 'react'
import { userContext } from './context'

export default class Header extends React.Component {
    static contextType = userContext

    render() {
        const user = this.context

        const headerStyle = {
            backgroundColor:'#cee',
            textAlign:'center',
            padding: 5
        }
        return (
            <div style={headerStyle}>
                <a href=" ">Home</a> - 
                <a href=" ">Product</a> - 
                <a href=" ">Contact Us</a> -  
                [{user} : <a href=" ">Signout</a>]
            </div>
        )
    }
}
```

Centralized storage with Context

react/app1/src/context.js

```
import React, {createContext} from 'react'


export const userContext = createContext()
```



react/app1/src/App.js

```
import React from 'react'
import { userContext } from './context'
import Header from './context-header'


export default function App() {
  return (
    <userContext.Provider value={'Tom Jerry'}>
      <Header/>
    </userContext.Provider>
  )
}
```

Centralized storage with Context

- ## Using Context Data in Function Component

Using values from context in a function component can be done simply because there is already a function in the React Hook group that is directly available, namely *useContext()*, as follows:

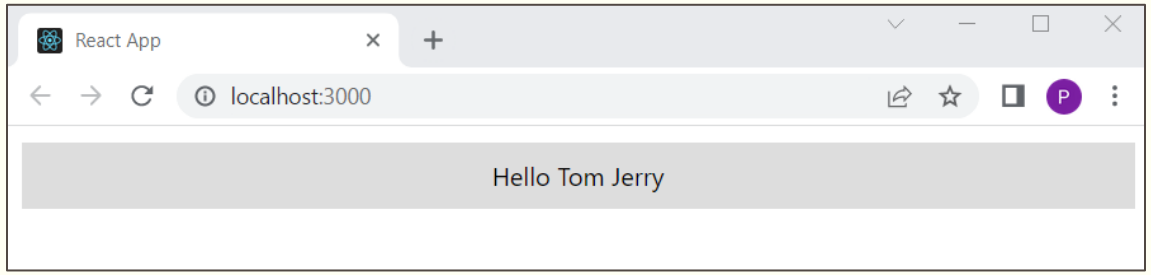
react/app1/src/context-content.js

```
import React from 'react'
import { ..... } from './context'

export default function Content() {
    let user = React.useContext(userContext)

    const ..... = {
        .....:'#ddd',
        textAlign:'.....',
        margin: 10,
        padding: 10
    }

    return (
        <div style={contentStyle}>
            ..... {user}
        </div>
    )
}
```

Centralized storage with Context

react/app1/src/context.js

```
import React, {createContext} from 'react'


export const userContext = .....................................
```

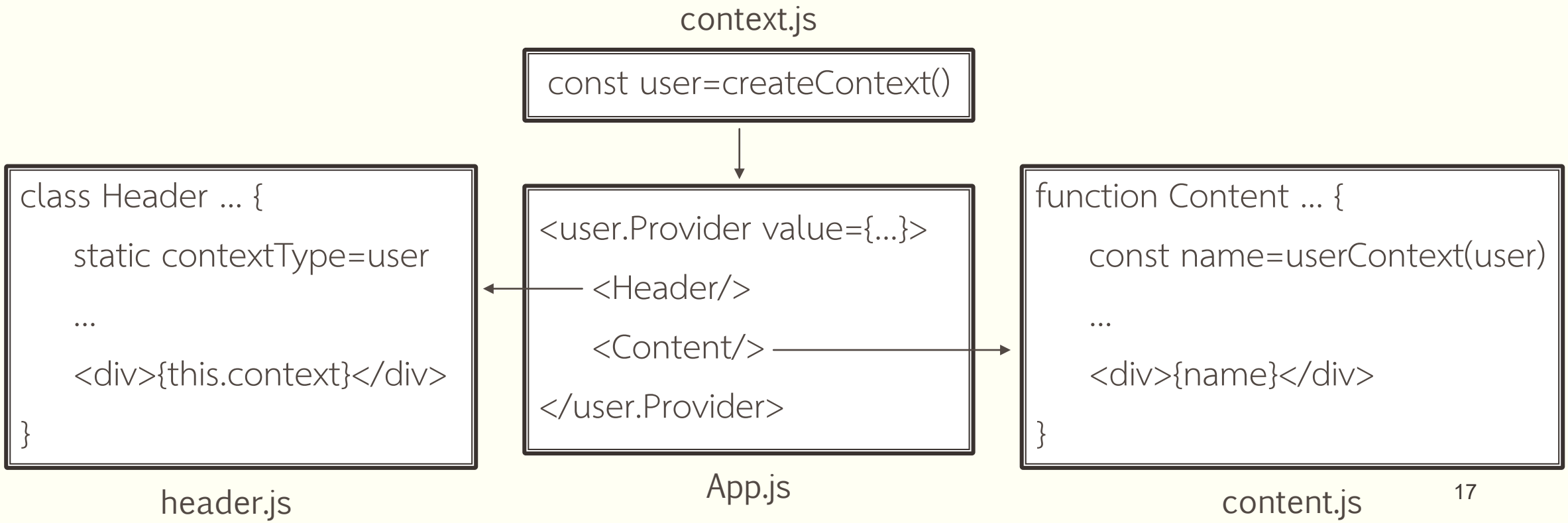


react/app1/src/App.js

```
import React from 'react'
import {.....................} from './context'
import ..................... from './context-content'


export default function App() {
  return (
    <..................................... value={'Tom Jerry'}>
      <Content/>
    </.....................................>
  )
}
```

Centralized storage with Context

From the creation of context variables, to provider wrapping, to both class and function component implementation, this can be summarized as shown in the diagram below. However, if the code is defined as in the past. When you put them all together and do a test, you can get the results as shown in the following picture.



Centralized storage with Context

react/app1/src/App.js

```
import React from 'react'
import { userContext } from './context'
import Header from './context-header'
import Content from './context-content'

export default function App() {
  return (
    <userContext.Provider value={'Tom Jerry'}>
      <Header/>
      <Content/>
    </userContext.Provider>
  )
}
```

Centralized storage with Context

- ## Storing State Data in a Context

Using data from contexts, we set a fixed value to the provider, so it is not possible to update that information later to be consistent across all components. We may resolve this issue by storing state data in context instead of the original format. Next, when we update the context information on any component, it will immediately affect the other components.

- The context creation file is defined as before, such as:

react/app1/src/context.js

```
import  React, { createContext }  from  'react'


export  const  userContext=createContext()  //หรือ React.creatContext()
```

Centralized storage with Context

- At the App.js file, which is usually in the form of a Function Component, we create a State according to the original principle, such as: *let [user, setUser]=React.useState()*
- Create a provider and assign a value from State to the value property in an array of both its values and the function for changing the value, such as:

react/app1/src/App.js

```
import React from 'react'
import { userContext } from './context'
import ..... from './context-header2'
import ..... from './context-content2'
```

```
export default function App() {
    let [user, setUser] = .....
    return (
        <..... value={[user, setUser]}>
            <Header2/>
            <Content2/>
        </.....>
    )
}
```

Centralized storage with Context

- The component to which we will apply the value from the context must read the value of the context in array destructuring to obtain both variables and functions to change the value of the state. We then take it from variable values to the components as usual. And if you want to update, do it through the State function as follows.

react/app1/src/context-header2.js

```
import React from 'react'
import {.....} from './context'

export default class ..... extends React.Component {
    static contextType = userContext

    render() {
        let [user, setUser] = .....
```

```
        const headerStyle = {
            backgroundColor:'#cee',
            textAlign:'center',
            padding: 5
        }

        const onClickSignout = (event) => {
            event.preventDefault()
            setUser('')
        }
```

Centralized storage with Context

react/app1/src/context-header2.js (ต่อ)

```
    const onClickSignin = (event) => {
        event.preventDefault()
        setUser('.....')
    }

    return (
        <div style={.....}>
            <a href=" ">Home</a> - 
            <a href=" ">Product</a> - 
            <a href=" ">Contact Us</a> -  
```

```
            {
                (user)
                ? <span>[{user} : <a href=" "
                        onClick={.....}>Signout</a>]</span>
                : <span>[<a href=" " onClick={.....}>
                        Signin</a>]</span>
            }
        </div>
    )
  }
}
```

Centralized storage with Context

react/app1/src/context-content2.js (ต่อ)

```
import ..... from 'react'
import { ..... } from './context'

export default function ..... {
    let [user, setUser] = .....

    const ..... = {
        backgroundColor:'#ddd',
        textAlign:'center',
        margin: 10,
        padding: 10
    }

    const onClickSignin = (.....) => {
        event.preventDefault()
        setUser('.....')
    }

    return (
        <div style={contentStyle}>
        {
            (user)
            ? <span>Hello {.....}</span>
            : <span>Please <a href=" " onClick={.....}>
                        Signin</a></span>
        }
        </div>
    )
}
```

Centralized storage with Context