REACT HOOKS CHEATSHEET

Name

useState

Short description

useState is a React Hook that lets you add a state variable to your component.

Syntax/Anatomy

Setter is a function to change state value

```
const [value, setValue] = useState("React");
```

Initial value for the state

A state is a value managed by React. Its value is Retained between renders, and it triggers a re-render when changed

Basic usage

```
const User = () => {
  const [name, setName] = useState<string>("");

  const changeHandler = (e: ChangeEvent<HTMLInputElement>) => {
    setName(e.target.value);
  };

  return (
    <div>
      <input type="text" onChange={changeHandler} />
      <p>Name: {name}</p>
    </div>
  );
};
```

@JuntaoQiu

Usage - lazy initalise

You can pass in a function (initialiser) into useState, it needs to be a pure and non-argument function, the return value will be used as the initalise state value - so it will only execute once.

```
const User = () => {
  const [name, setName] = useState<string>(() => {
    // expensive and one-time off computing
    return "React";
  });
};
```

Usage - updating object/array

In React, state is considered read-only, so you should replace it rather than mutate your existing objects (note the spread assignment below).

```
const UserList = () => {
  const [users, setUsers] = useState<UserType[]>([]);

  const changeHandler = (e: ChangeEvent<HTMLInputElement>) => {
    setUsers([...users, { name: e.target.value }]);
  };

  return (
    <div>
      <input type="text" onChange={changeHandler} />
      {users.map((user) => (
        <div>{user.name}</div>
      ))}
    </div>
  );
};
```

Caveats

- As a hook, you can't call useState inside loops or conditions
- Setter updates the state variable for the next render
- React only re-render if the state is changed, when setter is called with a value that is identical to current state, it does nothing.

Name

`useEffect`

Short description

useEffect lets you synchronize a component with an external system. It could be handling network requests, handling DOM events or some other third-party widgets.

Syntax/Anatomy

setup function, this is the callback function React invoke when one of the dependencies changed.

```
useEffect(() ⇒ {
  fetch("https://api.server/users").then(r ⇒ r.json()).then()
}, [id])
```

Optional dependencies list, the effect will be executed whenever a change detected in the list

Basic usage

```
const UserList = () ⇒ {
  const [users, setUsers] = useState<UserType[]>([]);

  useEffect(() ⇒ {
    const fetchUsers = async () ⇒ {
      await fetch("https://api.server/users")
        .then((r) ⇒ r.json())
        .then((data) ⇒ setUsers(data));
    };

    fetchUsers();
  }, []);

  return ();
};
```

Usage - cleanup function

In the setup function, you should consider to return a cleanup in the end. After every re-render with changed dependencies, React will first run the cleanup function, and then run your setup with the new values. The cleanup will be called when the component is unmounted from DOM.

```
useEffect(() => {
  const handler = (e: KeyboardEvent) => {};
  window.addEventListener("keydown", handler);
  return () => {
    window.removeEventListener("keydown", handler);
  };
}, [inputRef]);
```

Usage - the setup effect (one-off)

You might not want to execute the effect everytime, for example, you might only want to register a event handler or fetch a collection of data for the very first time - then you pass in a **empty array** as dependencies list.

```
const UserList = () => {
  const [users, setUsers] = useState<UserType[]>([]);

  useEffect(() => {
    const fetchUsers = async () => {
      await fetch("https://api.server/users")
        .then((r) => r.json())
        .then((data) => setUsers(data));
    };
    fetchUsers();
  }, []);
  fetchUsers();
  }, []);

};
```

Caveats

- As a hook, you can't call useEffect inside loops or conditions
- You probably don't need an Effect if you're not trying to synchronize with some external system (network, third-party widgets, etc.)
- Effects only run on the client, so they don't work with SSR

Pitfall #1 - unnecessary state

The useEffect in the example below is not necessary. In React world, When something can be calculated from the existing props or state, don't put it in state. Instead, calculate it during rendering.

```
const User = () => {
  const [firstName, setFirstName] = useState('John');
  const [lastName, setLastName] = useState('Smith');

  const [fullName, setFullName] = useState('');

  useEffect(() => {
    setFullName(firstName + ' ' + lastName);
  }, [firstName, lastName]);
};
```

Fix for pitfall #1 - unnecessary state

Simply replace the useEffect with a humble variable will work fine in this case. The code runs faster and easier to understand.

```
const User = () => {
  const [firstName, setFirstName] = useState('John');
  const [lastName, setLastName] = useState('Smith');

  const fullName = firstName + ' ' + lastName;

  return (
    <div>
      <p>Name: {fullName}</p>
    </div>
  );
};
```

There is a similar case that we will cover with another example in **useMemo** section.

Pitfall #2 - event-specific logic

Use useEffect only for code that should run because the component **was displayed** to the user. The sequence seemingly is click -> save -> user updated -> notification shows, actually the notification shows after page reload too..

```
useEffect(() => {
  if(user.updated) {
    showNotification('User profile is updated');
  }
}, [user])

const saveProfile = () => {
  saveUser({ ... user, name: fullName });
}

return (
  <div>
    <p>Name: {fullName}</p>
    <button onClick={saveProfile}>Save Profile</button>
  </div>
);
```

Fix for pitfall #2 - remove unnecessary effect

The notification should appear when the user clicked the button, not because the page was displayed.Simply delete the useEffect will work.

```
const saveProfile = () => {
  saveUser({ ... user, name: fullName });
  showNotification('User profile is updated');
}

return (
  <div>
    <p>Name: {fullName}</p>
    <button onClick={saveProfile}>Save Profile</button>
  </div>
);
```

Name

useContext

Short description

useContext is a React Hook that lets you read and subscribe to context from your component. It's a great way to share data (state) across your application.

Syntax/Anatomy

Value defined in the context, you can use any valid JS object, such as string, number, array and function.

```
const { mode } = useContext(ThemeContext);
```

The context object, which is created by React.createContext().

Basic usage

```
const ThemeContext = createContext<ThemeType>({mode: "light"});    1


const App = () => {
  return (
    <ThemeContext.Provider value={{ mode: "light" }}>              2
      <Payment amount={20.4} />
    </ThemeContext.Provider>
  );
}


const Payment = () => {
  const { mode } = useContext(ThemeContext);                       3
  // ...
}
```

Usage - change context value

To update context, you need to combine it with state. Declare a state variable in the parent component, and pass the current state down as the context value to the provider.

```
const App = () => {
  const [mode, setMode] = useState<ThemeMode>("light");

  return (
    <ThemeContext.Provider value={{ mode }}>
      <Payment amount={20.4} />
      <button
        onClick={() => {
          setMode("dark");
        }}
      >
        Switch theme
      </button>
    </ThemeContext.Provider>
  );
};
```

Usage - nesting contextes

You can have as many context as you want, each of them should responsible for a particular aspect of the functionaility.

```
export function App() {
  return (
    <ThemeContext.Provider value={{ mode: "light" }}>
      <PaymentContext.Provider
        value={{
          strategy: new PaymentStrategy("$")
        }}
      >
        <Payment amount={20.4} />
      </PaymentContext.Provider>
    </ThemeContext.Provider>
  );
}
```

Name

useCallback

Short description

useCallback is a React Hook that lets you cache a function definition between re-renders. You might need it when doing performance tunning.

Syntax/Anatomy

Cached function, you can call it just like the original function.

```
const fetchUser = useCallback((type: string) => {
  fetch(`https://api.service/${userId}/${type}`);
}, [userId]);
```

The origin function need to be cached

Dependencies list, a new function will be returned whenever the list changed.

Basic usage

```
const Todos = () => {
  const [todos, setTodos] = useState([]);

  const addTodo = useCallback((content: string) => {
    const item = { id: uuid(), content };
    setTodos(todos => [item, ...todos]);
  }, []);

  return (
    <div>
      <input type="text"/>
      <button onClick={() => addTodo("new todo")}></button>
    </div>
  );
};
```

Name

useMemo

Short description

useMemo is a React Hook that lets you cache a value between re-renders. You might need it when doing performance tunning just like **useCallback** for functions.

Syntax/Anatomy

Cached value, you can use it just like another other value.

```
const displayTodos = useMemo(() => {
  return filterTodos(todos, keywords);
}, [todos, keyworks]);
```

The origin calculation need to be cached

Dependencies list, the function will be evaluated whenever the list changed.

Basic usage

```
const Todos = () => {
  const [todos, setTodos] = useState([]);

  const displayTodos = useMemo(() => {
    return filterTodos(todos, keywords);
  }, [todos, keyworks]);

  return (
    <div>
      <input type="text" />
      {displayTodos.map((todo) => (
        <div>{todo.content}</div>
      ))}
    </div>
  );
};
```

@JuntaoQiu