Жизненный цикл компонента в React

Этап № 1: монтирование

Как известно, компоненты, созданные на основе классов, тоже являются классами. Именно поэтому первый запускаемый метод — constructor(). Как правило, именно в constructor() мы выполняем инициализацию состояния компонента.

Далее компонент запускает getDerivedStateFromProps(), потом запускается render(), возвращающий JSX. React «монтируется» в DOM. Следующий этап — запуск метода componentDidMount(). Тут происходит выполнение всех асинхронных вызовов к базам данных. Итак, компонент «рожден».

Этап № 2: обновление

Данный этап запускается во время каждого изменения состояния либо свойств. Как и в случае с монтированием, происходит вызов метода getDerivedStateFromProps(), однако в этот раз уже без constructor().

Потом происходит запуск shouldComponentUpdate(). Тут можно выполнять сравнение старых свойств с новым перечнем свойств либо сравнивать состояния. При этом мы можем указать, надо ли отображать компонент заново, возвращая true либо false — это даст возможность сделать приложение более эффективным благодаря уменьшению числа лишних отображений. Если же shouldComponentUpdate() возвращает false, этап обновлений завершается.

В обратном случае React отобразится заново, а потом запустится getSnapshotBeforeUpdate(). Потом React запустит componentDidUpdate(). Как и в случае с componentDidMount(), его можно применять для асинхронных вызовов либо управления DOM.

Этап № 3: размонтирование

Все хорошее имеет тенденцию заканчиваться. Когда компонент прожил свою жизнь, наступает размонтирование — последний жизненный этап. Во время удаления компонента из DOM React выполняет запуск componentWillUnmount() непосредственно перед удалением. Данный метод применяется при закрытии всех открытых соединений типа web-сокетов либо тайм-аутов.

Хуки

jsx

Именно благодаря ему у функциональных компонентов появилось состояние.

```
const App = () => {
  const [value, valueChange] = useState(0);

  return (
    <div>
      {value}
      <button onClick={() => valueChange(value + 1)}>
        Увеличить значение на 1
      </button>
    </div>
  );
};
```

useContext

Чтобы передать какие-то данные в компонент, мы можем использовать props. Но есть и альтернативный способ – context. Используются, чтобы не протаскивать (drops drilling) через все дочерние porps.

Все вложенные в компонент который использует данный хук, компоненты смогут получить доступ к данным, которые мы передаем, помещая их в параметр value.

```
import {createContext, useContext} from "react";

const MyContext = createContext("without provider");

const External = () => {
  return (
    <MyContext.Provider value="Hello, i am External">
      <Intermediate />
    </MyContext.Provider>
  );
};

const Intermediate = () => {
  return <Internal />;
};

const Internal = () => {
  const context = useContext(MyContext);

  return `I am Internal component. I have got the message from External:
"${context}"`;
};
```

useEffect, useLayoutEffect

В случае с useLayoutEffect React не запускает рендеринг построенного DOM дерева до тех пор, пока не отработает useLayoutEffect. Если же мы берём useEffect, то React сразу запускает рендеринг построенного DOM, не дожидаясь запуска useEffect.

useEffect принимает в себя два аргумента:

- callback. Внутри него вся полезная нагрузка, которую мы хотим описать. Например, можно делать запросы на сервер, задание обработчиков событий на документ или что-то ещё;
- массив, состоящий из аргументов. При изменении значения внутри любого из них будет запускаться наш callback. Именно благодаря этому аргументу мы можем имитировать методы жизненного цикла.

  ```
  useEffect(() => {
   console.log("componentDidUpdate");
  }, [data]);
  ```

  При размонтирование или ререндере компонента, нам нужно отвязать обработчики событий

```
const App = ({data}) => {
  useEffect(() => {
    return () => {
      console.log("componentWillUnmount");
    };
  }, []);

  return null;
};
```

useRef

Бывают ситуации, когда необходимо обратиться к какому-то DOM-объекту напрямую. Для этого существует хук useRef.

```
const App = () => {
  const ref = useRef();

  useEffect(() => {
    console.log(ref.current);
  }, []);

  return <div ref={ref} />;
};
```

Далее мы можем взаимодействовать с Dom-объектом напрямую, как если бы мы нашли его с помощью селектора.

useReducer

Разработчикам React так понравился Redux, что они решили добавить его аналог в состав React. Этот хук позволяет вынести данные из компонентов.

```
import {useReducer} from "react";

const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return {
        ...state,
        count: state.count + 1,
      };
    case "decrement":
      return {
        ...state,
        count: state.count - 1,
      };
    default:
```

```
      throw new Error();
  }
}

const App = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      {state.count}
      <button onClick={() => dispatch({type: "decrement"})}>-</button>
      <button onClick={() => dispatch({type: "increment"})}>+</button>
    </>
  );
};
```

У него есть преимущество: вне зависимости от того, как компоненты нашего приложения будут вложены друг в друга, мы сможем отобразить данные в любом компоненте.

useMemo

Этот хук позволяет не производить одни и те же вычисления много раз. В этой ситуации компонент перерендеривается в том случае, если изменяется один из параметров – a или b. Представим, что у нас много раз изменяется параметр b, при этом параметр a остаётся прежним. В таком случае мы много раз вычисляем одно и то же произведение, которое помещаем в переменную sqrt. Тут нам и помогает useMemo

```
const MyComponent = ({a, b}) => {
  const sqrt = useMemo(() => a * a, [a]);

  return (
    <div>
      <div>A в квадрате: {sqrt}</div>
      <div>B: {b}</div>
    </div>
  );
};
```

useCallback

В силу того, что функциональный компонент – это функция, при каждом рендеринге запускается всё, что объявлено в ней. Предположим, что мы создаем внутри компонента функцию и передаем ее в дочерний компонент. Это самая обыкновенная практика. Она часто встречается, когда нам нужно из дочернего компонента изменить что-то в родительском.

В данном случае у нас возникает проблема: при каждой отрисовке компонента App мы будем заново создавать метод changer. Хотя сигнатура у метода будет одинаковой, каждый раз будет создан новый метод, следовательно, у ControlPanel будут происходить повторные рендеринги, но по сути ничего не меняется.

```
const ControlPannel = memo(({changer}) => {
  return (
    <div>
      <button onClick={changer}>+</button>
    </div>
  );
});

const App = () => {
  const [value, valueChange] = useState(Math.random());

  const changer = () => valueChange(Math.random());

  return (
    <div>
      {value}
      <ControlPannel changer={changer} />
    </div>
  );
};
```

Избежать этого поможет useCallback.

```
const ControlPannel = memo(({increment}) => {
  return (
    <div>
      <button onClick={increment}>+</button>
    </div>
  );
});

const App = () => {
  const [value, valueChange] = useState(Math.random());

  const increment = useCallback(() => valueChange(Math.random()), []);

  return (
    <div>
      {value}
      <ControlPannel increment={increment} />
    </div>
  );
};
```

HOC

Компонент высшего порядка - функция которая принимает компонент и возвращает новый компонент.

Это удобный способ переиспользования логики.

Используется паттерн **Прокси**.

```
const withLayout = <T extends Record<string, unknoun>>(Component:
FunctionComponent<T>) => {
    return function widthLayoutComponent(props: T): JSX.Element {
        return <Component {...props}>
    }
}
```