GeekBrains

State, Props. Жизненный цикл React компонента.

React

Оглавление

State

На предыдущем уроке уже упоминалось, что компоненты могут не только отображать переданные им статические данные, но и инкапсулировать некоторую логику. Это достигается, в том числе, за счет использования переменных состояния, или стейта. Стейт (state) - специальная переменная, сохраняющая переданное ей значение внутри компонента.

App.js

```
import React from 'react';

export function Counter() {
 return (
   <div>
```

```
      <span className="counter">0</span>
      <button className="counter-button">Click!</button>
    </div>
  )
}

export default Counter;
```

Здесь на странице рендерится кнопка и некоторый счетчик - значение переменной counter. Добавим интерактивности - пусть по нажатию на кнопку отображаемое число увеличивается. Для этого сперва значение переменной count сохраним в стейт с помощью специальной функции-хука useState:

App.js

```
import React, { useState } from 'react';

export function Counter() {
const [count, setCount] = useState(0);
return (
  <div>
    <span className="counter">{count}</span>
    <button className="counter-button">Click!</button>
  </div>
)
}

export default Counter;
```

useState - один из основных хуков React (хуки - специальные функции, используемые в функциональных компонентах. Он принимает в себя начальное значение переменной и возвращает массив из двух значений - первым элементом этого массива является текущее значение переменной, а вторым - функция-сеттер для обновления значения этой переменной. Присвоение полученных из useState значений переменным обычно происходит с помощью деструктуризации массива:

```
const [variable, setVariable] = useState('initial');
// то же самое, что
```

```
const variableState = useState('initial');
const variable = variableState[0];
const setVariable = variableState[1];
```

Как правило, функцию-сеттер называют так же, как переменную, но с префиксом set (например, для переменной messages функцию называют setMessages). В эту функцию требуется передать значение, которое необходимо установить переменной стейта (в данном случае переменной count).
Добавим теперь функцию upCount и назначим ее кнопке обработчиком события onClick:

```
import React, { useState } from 'react';

export function Counter() {
 const [count, setCount] = useState(0);

 const upCount = () => {
   setCount(count + 1);
 }

 return (
   <div>
     <span className="counter">{count}</span>
     <button onClick={upCount}
className="counter-button">+1</button>
   </div>
 )
}

export default Counter;
```

Теперь при нажатии на кнопку значение, хранящееся в переменной count, будет увеличено, компонент обновится и отобразит обновленное значение в браузере.

Важно:

Функцию **setCount** можно вызывать не только со значением, которое требуется установить переменной, но и с коллбэком:

```
const upCount = () => {
  setCount((number) => number + 1);
}
```

В этом случае аргументом в коллбэк будет передано (автоматически) предыдущее значение переменной (в данном случае count), а вернуть из функции требуется новое значение. Такой подход используют, как правило, когда необходимо изменить обновить стейт, учитывая предыдущее его значение. Необходимость использования такой записи связана, в частности, с тем, что обновление состояния происходит асинхронно, и к моменту вызова setCount(count + 1) переменная count может иметь устаревшее значение. Вызывая setCount с функцией, мы всегда будем иметь доступ к актуальному значению count (number в примере).

State в классовых компонентах

Аналогично, данный функционал можно реализовать на классовых компонентах. В этом случае используется объект this.state, хранящий значения всех переменных состояния, а для обновления используется метод this.setState:

```
import React from 'react';

export class Counter extends React.Component {
 state = {
   count: 0, // начальное значение
 };

 upCount = () => {
   const { count } = this.state; // деструктуризация стейта
   this.setState({ count: count + 1 });
 };

 render() {
   return (
     <div>
       <span className="counter">{this.state.count}</span>
       <button className="counter-button" onClick={this.upCount}>
```

```
          Click!
        </button>
      </div>
    );
  }
}

export default Counter;
```

Какой важный момент мы можем заметить, что у нас может быть несколько свойств внутри this.state и получается что при обновлении одного из них не требуется переделывать весь объект

```
import React from 'react';

export class Counter extends React.Component {
 state = {
   count: 0, // начальное значение
   name: 'Alex'
 };

 updateCount = () => {
   const { count } = this.state; // деструктуризация стейта
   this.setState({ count: count + 1 });
   // this.state.name не изменится
 };

 render() {
   return (
     <div>
       <span className="name">{this.state.name}</span>
       <span className="counter">{this.state.count}</span>
                                <button    className="counter-button"
onClick={this.updateCount}>
         Click!
       </button>
     </div>
```

```
    );
  }
}

export default Counter;
```

Здесь в стейте хранятся переменные count и name, и происходит вызов setState с объектом, содержащим только count. Однако после такого вызова name не исчезает и по прежнему равна установленному ранее значению.
Кроме того, как и в функциональных компонентах, можно вызывать setState с функцией.
Как уже упоминалось, setState выполняется асинхронно. Из этого следует две важные вещи:

```
updateCount = () => {
  this.setState({ count: 3 });
  this.setState({ name: 'noname' });
  this.setState({ count: 2 });
  console.log(this.state);
};
```

1. В данном примере React сможет объединить вызовы setState и выполнить одно обновление компонента (эквивалентно одному вызову такого вида: this.setState({ count: 2, name: ‘noname’ });
2. console.log выполнится до обновления стейта.

Если существует необходимость выполнить некоторую функцию после того как обновился стейт, в классовом компоненте следует передать эту функцию вторым аргументом this.setState:

```
updateCount = () => {
  this.setState({ count: 2, name: 'noname' },
    () => {
      console.log(this.state);
    });
};
```

Стейт следует изменять только с помощью вызова setState (в классовых компонентах) или функции-сеттера, возвращаемой из useState (в функциональных).

Не изменяйте переменные состояния напрямую! Использование записи вида this.state = { count: 1 } допустимо только в конструкторе классового компонента. После вызова setState React, при необходимости, обновит UI в соответствии с изменениями состояния. При присваивании значений в стейт напрямую такого обновления выполнено не будет.

Контролируемые формы

Достаточно часто стейт используется для создания контролируемых форм ввода.

```
import React, { useState } from 'react';

export function Counter() {
 // const [value, setValue] = useState('');
 const [value, setValue] = useState()

 const handleChange = (event) => {
   setValue(event.target.value);
 }

 return (
   <div>
     <input type="text" value={value} onChange={handleChange} />
   </div>
 )
}

export default Counter;
```

Здесь элементу input устанавливаются атрибуты value - значение, отображаемое в поле - и onChange - функция, вызываемая при изменении этого значения. В качестве value в этом случае устанавливается значение переменной стейта, а для onChange - функция, изменяющая эту переменную. В этом случае значение, введенное пользователем в поле ввода, хранится внутри компонента, и для доступа к нему нет необходимости получать его из DOM.

Обратите также внимание на то, что при изменении значения в поле ввода (событие onChange) аргументом в функцию-обработчик попадает специальный объект

события. Это т.н. SyntheticEvent - специальная обертка над браузерным событием, которую React использует для поддержания кросс-браузерности.

Жизненный цикл компонента

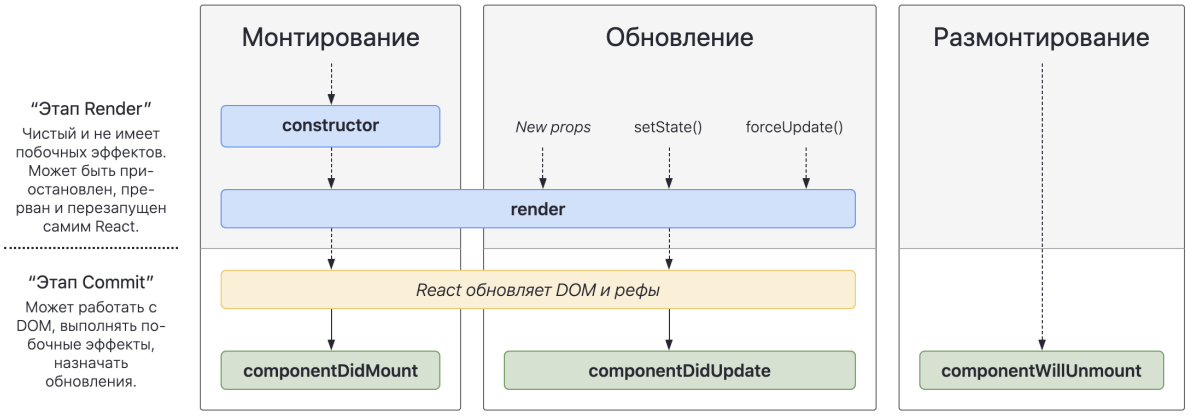
Как мы увидели из предыдущих примеров, при изменении переменной стейта с помощью вызова setState происходит обновление компонента. Компонент обновляется за счет внутренних механизмов React, при этом могут происходить некоторые дополнительные действия (например, React вызовет некоторые функции или методы компонентов).

Каждый компонент за время своего существования проходит определенные этапы, которые называют стадиями жизненного цикла:

1. Монтирование (mount): компонента не было на странице и он добавляется туда в первый раз (либо он существовал, был удален и появляется заново).
2. Обновление (update): компонент уже существует, но обновились данные, от которых он зависит - стейт или пропсы.
3. Размонтирование (unmount): компонент существует и происходит его удаление из DOM.

Методы жизненного цикла

На каждом этапе жизненного цикла React вызывает соответствующие методы классовых компонентов или коллбэки хуков в функциональных. Рассмотрим сперва на примере классового компонента:



Как видно, на начальном этапе будет вызван конструктор класса, затем метод render, затем componentDidMount (обратите внимание - componentDidMount вызывается после того как компонент отрендерен и существует в DOM). В этом методе, как правило, запускаются побочные эффекты компонента - отправляются запросы на сервер, ставятся таймеры, подписки и т.п.

При изменении стейта (через setState) или пропсов будут вызваны методы render и componentDidUpdate. Этот метод первыми двумя аргументами получает значения пропсов и стейта до обновления, и при необходимости здесь можно выполнить сравнение пропсов или стейта до и после обновления.
При размонтировании (перед тем как компонент будет удален из памяти) будет вызван метод componentWillUnmount - он служит в первую очередь для отмены ранее отправленных в компоненте запросов, таймеров, подписок и т.п. Это позволяет избежать утечек памяти.
Рассмотрим следующий пример:

App.js

```
export class Counter extends React.Component {
 constructor(props) {
   super(props);

   console.log('constructor');
 }

 componentDidMount() {
   console.log('commponentDidMount');
 }

 render() {
   console.log('render');

   return (
     <div>rendered!</div>
   );
 }
}

export default Counter;
```

При такой записи в консоли отобразится следующее:

```
constructor
render
commponentDidMount
```

Усложним пример, добавив дочерний компонент:

```
export class Counter extends React.Component {
 constructor(props) {
   super(props);
   console.log('constructor');
 }
 componentDidMount() {
   console.log('componentDidMount');
 }
 render() {
   console.log('render');
   return (
     <div>
       rendered!
       <Child />
     </div>
   );
 }
}

class Child extends React.Component {
 constructor(props) {
   super(props);
   console.log("child constructor");
 }
 componentDidMount() {
   console.log("child componentDidMount");
 }
 render() {
   console.log("child render");
   return <div>rendered!</div>;
 }
}
```

```
constructor
render
child constructor
child render
child componentDidMount
componentDidMount
```

Обратите внимание, что сперва вызывается метод componentDidMount дочернего, а затем родительского компонента.

Хуки для работы с жизненным циклом

В функциональных компонентах не существует методов жизненного цикла, однако аналогичное поведение достигается использованием хука useEffect. Он принимает два аргумента - коллбэк и массив зависимостей. Коллбэк useEffect будет выполнен:

1. При монтировании компонента
2. При изменении одной из зависимостей

То есть, при использовании следующей записи

```
import React, { useEffect } from 'react';

export function Example(props) {
 const { name } = props.name;

 useEffect(() => {
   console.log('useEffect');
 }, [name]);

 return (
   <div>
     {name}
   </div>
 )
}
```

```
export default Example;
```

коллбэк будет выполнен при первом вызове функции-компонента, а затем только при изменении пропса name.
Если указать вторым аргументом пустой массив - функция будет выполнена один раз, только после монтирования компонента (аналогично componentDidMount). Если не передавать второй аргумент - коллбэк исполняется при каждом обновлении (аналогично componentDidUpdate, но, в отличие от этого метода, коллбэк useEffect выполнится и на первом рендере тоже).

```
export function Example(props) {
 const { name } = props.name;

 useEffect(() => {
   console.log('like didMount');
 }, []);

 useEffect(() => {
   console.log("like didUpdate");
 });

 return (
   <div>
     {name}
   </div>
 )
}
```

С использованием хука эффекта можно легко решить задачу выполнения некоторой функции в ответ на изменение переменной стейта:

```
import React, { useEffect, useState } from 'react';

export function Counter() {
 const [count, setCount] = useState(0);
 const updateCount = () => {
   setCount((prevCount) => prevCount + 1);
```

```
 }
 useEffect(() => {
   console.log(count);
 }, [count]);
 return (
   <div>
     <span className="counter">{count}</span>
     <button className="counter-button"
onClick={updateCount}>Click!</button>
   </div>
 )
}

export default Counter;
```

console.log(count) будет выполняться каждый раз, когда изменяется значение переменной count.

Props

На прошлом уроке мы уже использовали пропсы, чтобы передавать дочернему компоненту статические данные. Рассмотрим теперь случай, когда пропсами передается значение переменной стейта:

```
function Child(props) {
 return (
   <span>{props.number}</span>
 )
}
export function Example() {
 const [count, setCount] = useState(0);

 return (
   <div>
     <Child number={count} />
   </div>
 )
```

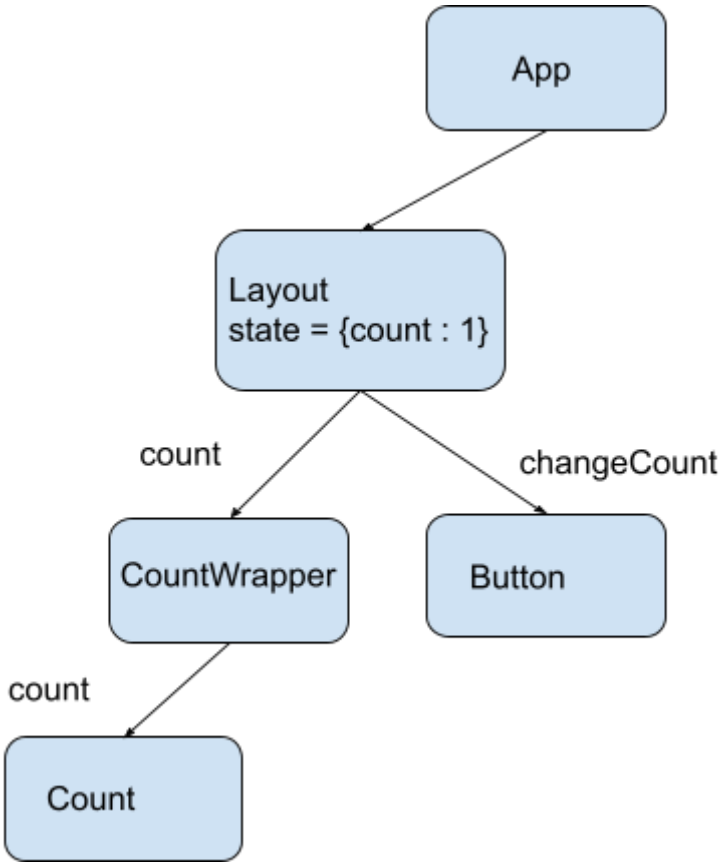
```
}
export default Example;
```

Теперь дочерний компонент получает значение переменной из стейта родителя. Мы можем передавать эту переменную и другим дочерним компонентам. Кроме того, пропсами можно передавать функции - например, для изменения переменной стейта. Рассмотрим пример, где один из дочерних компонентов получает и отображает значение переменной, а другой - коллбэк для ее изменения.

```
function Child(props) {
 return (
   <span>{props.number}</span>
 )
}
function Button(props) {
 return (
   <button onClick={props.onClick}>Click +1</button>
 )
}
export function Example() {
 const [count, setCount] = useState(0);
 const changeCount = () => {
   setCount(prevCount => prevCount + 1);
 }
 return (
   <div>
     <Child number={count} />
     <Button onClick={changeCount} />
   </div>
 )
}
export default Example;
```

Заметьте, что, хотя переменная count отображается в компоненте Child, ее значение хранится в родительском компоненте Example - а компонент Button получает функцию для изменения этой переменной. Такой подход называется поднятие стейта - если два компонента должны иметь доступ к одной и той же переменной стейта, ее помещают в ближайшего общего родителя этих компонентов.

Компоненты должны хранить и передавать переменные только сверху вниз - от родительских компонентам дочерним. Изменение данных родительского компонента из дочернего следует осуществлять только с помощью переданных дочернему компоненту пропсами коллбэков.



Компонентам Count и Button, не связанным между собой, требуется доступ к переменной count. Эта переменная помещена в стейт их ближайшего общего предка - Layout. Из него необходимые данные передаются пропсами соответствующим компонентам.
Как видно из схемы жизненного цикла компонента (в предыдущем разделе), при вызове setCount произойдет обновление компонента и вызов функции-компонента Example (для классовых компонентов - вызов метода render). При этом все переменные, созданные в теле функционального, создаются заново. Исключение составляют переменные и функции, созданные с использованием хуков - таким образом переменная count, созданная с помощью useState, сохраняет свое значение между рендерами.

Правила хуков. Другие хуки React.

Хуки - специальные функции, которые вызываются в теле функционального компонента и используются, в первую очередь, для сохранения значений

переменных между вызовами этой функции. Кроме уже описанных выше useState и useEffect, существует множество других, наиболее часто встречающимися из которых являются:

1. **useCallback** - используется для оборачивания функций, объявленных в теле функционального компонента. Функция, обернутая в useCallback, “запоминается” и будет пересоздаваться только в случае изменения любой из зависимостей - переменных, указанных в массиве, переданным вторым аргументом. Как правило, useCallback используют для функций, которые:
    - Передаются пропсами другому компоненту
    - Используются в других хуках в том же компоненте
2. **useMemo** - служит для мемоизации значений. К примеру, если в компоненте используется результат вычисления достаточно “тяжелой” функции (проход по массиву и т.п.), имеет смысл обернуть это вычисление в useMemo, чтобы не выполнять его на каждом рендере. Как и useCallback, и useEffect, функция будет выполняться только на первом рендере, а затем при изменении одной из зависимостей, однако, в отличие от useCallback, который возвращает переданную ему функцию, useMemo возвращает результат вычисления переданной функции.
3. **useRef** - возвращает специальный объект с мутируемым свойством current. Зачастую используется для получения ссылки на DOM-элемент

Для корректной работы с хуками следует соблюдать следующие правила:
Хуки должны вызываться только из функциональных компонентов на верхнем уровне (не из вложенных функций) или из других хуков.
Внутри функционального компонента хуки должны вызываться всегда в одном и том же порядке. То есть, нельзя оборачивать вызов хука в условия или циклы, а также после условного return. Для реализации работы хуков React отслеживает порядок их вызовов в компоненте, и нарушение этого порядка вызовет ошибку.
Указывайте в зависимостях хуков все переменные и пропсы, которые используются в коллбэке хука, в том числе и те, которые созданы с помощью других хуков. Исключение составляют переменные, объявленные вне тела компонента, а также функция-сеттер, возвращаемая из useState.

```
export function Example(props) {
 const badIdea = () => {
   const err = useCallback(() => {
     // вызовет ошибку
   }, [])
 }
```

```
 for (let i = 0; i < 10; i++) {
   // вызовет ошибку
   const res = useMemo(() => null, []);
 }
 if (!props.show) {
   return null
 }
 // ошибка!
 const dont = useRef(null);
 return (
   <div>
     Hooks
   </div>
 )
}
```

```
function Button(props) {
 const [count, setCount] = useState(0);

 const changeCount = useCallback(() => {
   // здесь переменная count всегда равна 0,
   // т.к. коллбэк не обновляется при ее изменении
   console.log(count);
 }, []);

 const changeCountCorrect = useCallback(() => {
   // здесь переменная count всегда актуальна,
   console.log(count);
 }, [count]);

 return (
   <div className="button" onClick={props.onClick}>Click!</div>
 )
}
```

Обратите внимание на то, что хуки можно вызывать из других хуков. Это связано с одним важным преимуществом хуков - вы можете создавать свои, “кастомные” хуки и переиспользовать их в различных компонентах. К примеру, с помощью следующего кастомного хука можно получить значение переменной на предыдущем рендере:

```
function usePrevious(value) {
 const ref = useRef();

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

 return ref.current;
}
```

Как итог

```
function Button() {
 const [count, setCount] = useState(0);

 const prevCount = usePrevious(count);

 useEffect(() => {
   if (prevCount !== count) {
     /* ... */
   }
 }, [prevCount, count]);

 const changeCount = useCallback(() => {
   setCount(1);
 }, []);

 return (
   <div className="button" onClick={changeCount}>
     Click!
   </div>
```

```
  );
}
```

Данная реализация позволяет использовать значение переменной count до последнего обновления, аналогично значению prevProps или prevState в методе componentDidUpdate классового компонента.

Рендер массивов. Фрагмент

Достаточно частой задачей является отображение списков (к примеру, список сообщений,товаров, и т.п.). Данные, как правило, хранятся в таком случае в массиве, а для рендера используется метод map:

```
export function MessagesList() {
  const [messages, setMessages] = useState([
    "message 1",
    "message 2",
    "message 3",
  ]);

  return messages.map((message) => <div>{message}</div>);
}
```

Если из компонента требуется вернуть несколько сиблингов, а добавление еще одного оборачивающего элемента (напр. div) нежелательно, следует использовать фрагмент. Фрагмент - это специальный компонент React, он служит исключительно для группировки нескольких элементов. При рендере в DOM он не будет добавлен как элемент:

```
function FragmentExample() {
  return (
    <>
      <span>This is right!</span>
      <div>Краткая запись фрагмента</div>
    </>
  );
}
// или
function FragmentExample2() {
  return (
```

```
    <React.Fragment>
      <span>This is right!</span>
      <div>Полная запись фрагмента</div>
    </React.Fragment>
  );
}
```

Итоги урока

Что мы можем заметить, конечно работа с React очень вдохновляет и содержит в себе огромное количество полезных особенностей, ключевыми из них являются хуки, конечно не стоит переживать что не получается запомнить все и сразу, основным является useState, с которым мы будем работать чаще всего, так что главное запомнить данный хук, а все остальные можно разобрать на мини примерах.