React.js Course

Занятие 4. Функциональные компоненты, хуки, контекст, рендер-пропсы

Автор программы и преподаватель - Помазкин Илья

Структура занятия

1. Рендер-пропсы
2. Контекст
3. Функциональные компоненты
4. Хуки
5. Библиотека для работы с формами - react-hook-forms
6. Библиотека для работы с запросами - react-query

Рендер-пропсы

Рендер-проп - это проп-функция, которая возвращает дерево React-элементов. По факту - это проп, который умеет рендерить что-то.

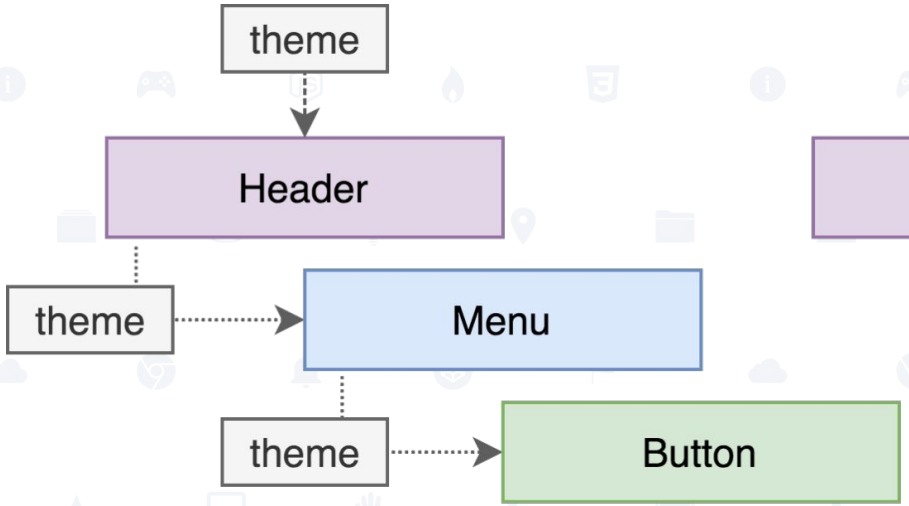
Компонент, который принимает такой проп вызывает функцию и рендерит то, что она вернет. Он может передавать в рендер-проп какие-то параметры, вычисляемые данные, что угодно. А сам рендер проп может использовать эти параметры при рендере.

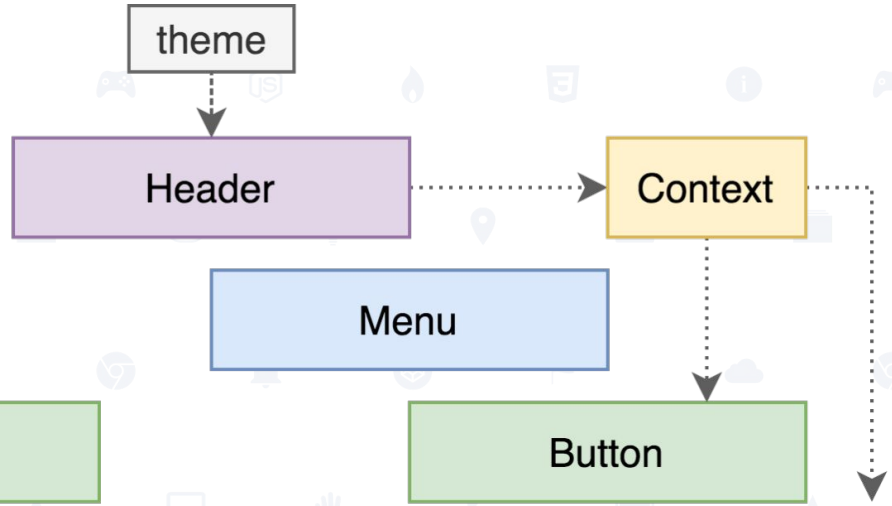
Рендер-пропы позволяют:
- переиспользовать какую-то логику в разных компонентах
- разделять разные блоки рендера и вставлять их в разные “слоты”.

Примеры:
- слежение за позицией курсора
- генерация CSS для анимаций
- обертка хедера, футера и контента в какой-то макет страницы

Контекст

Контекст нужен что бы передавать данные на любой уровень вложенности, без промежуточной передачи пропсов.





Контекст

Сначала нужно создать объект контекста. Для этого есть метод React.createContext. Этот объект будет иметь свойство “Provider”, что является компонентом для рендеринга. При его ренедере нужно передать проп “value”, что бы установить значение контекста.

```
import React from 'react';

const theme = { color: "red" }; // определим что храним в контексте

const ThemeContext = React.createContext(theme); // создадим объект контекста и передадим дефолтное
значение

class App extends React.Component {
 render() {
   return (
     <div className="app">
       <ThemeContext.Provider value={theme}> {/* Отрендерим провайдер и передадим в него текущее значение
*/}
         ...
       </ThemeContext.Provider>
     </div>
   );
 }
}
```

Контекст

Что бы получить доступ к контексту внутри какого-то компонента нужно отрендерить этот компонент внутри Provider. И при этом есть 2 способа, как достучаться до контекста внутри компонента.

```
class Button extends React.Component {
 render() {
   // получим доступ к контексту
   let theme = this.context;
   return (
     <button style={{color: theme.color}}>
       ...
     </button>
   );
 }
}

// Определим классу статичное свойство contextType
и запишем в него ссылку на оъект контекста
Button.contextType = ThemeContext;
```

```
class Button extends React.Component {
 render() {
   return (
     // используем комопонент Consumer для доступа
к контексту
     <ThemeContext.Consumer>
       {(theme) => ( // получим доступ к контексту
в рендер-пропе
         <button style={{color: theme.color}}>
           ...
         </button>
       )}
     </ThemeContext.Consumer>
   );
 }
}
```

Контекст

Важные моменты:

- при изменении значения контекста все компоненты, которые его используют будут перерендерены. Результат метода shouldComponentUpdate в этом случае будет проигнорирован
- контексты можно вкладывать друг в друга, компонент возьмет значение из ближайшего родителя-провайдера

Функциональные компоненты

Функциональный компонент - это функция, которая принимает на вход пропсы и возвращает дерево React-элментов. По факту - это почти то же самое, что метод render в классовых компонентах.

```
import React from 'react';
import PropTypes from "prop-types";

export function Button({
 // получим доступ к пропсам и сразу установим дефолтные значения
 children = null,
 ...rest
}) {
 return (
   <button {...rest}>{children}</button>
 );
}

// добавим валидацию пропсов
Button.propTypes = {
 children: PropTypes.node,
};
```

Хуки

Хуки - это функции, которые позволяют использовать состояние и жизненый цикл внутри функциональных компонентов. Они могут быть встроенными в React и пользовательскими.

Название хука начинается со слова *use:* useState, useEffect.

Нельзя менять порядок вызова хуков. Их нельзя вызывать внутри условий, циклов и вложенных функций. Иначе React не будет понимать, какой вызов соответсвует какому из хуков.

Хуки можно вызывать только из:
- тела самой функции функционального компонента
- из других хуков

Хуки - useState

Реализует работу с состоянием.

```
export function Header(props) {
 const [        // useState возвращает массив с 2 элементами
   isOpen,      // само состояние
   setIsOpen    // функция для обновления
 ] = React.useState(
   false // тут определяем значение по умолчанию
 );

 return (
  <header className="header">
    <div className="header__actions">
      <button onClick={() => setIsOpen(!isOpen)}>
        {
          isOpen ? "Close" : "Open" // использование состояния при рендере
        }
      </button>
    </div>
    {isOpen && ( // использование состояния при рендере
      <div className="header__menu"></div>
    )}
  </header>
 );
}
```

Хуки - useEffect

Позволяет работать с сайд-эффектами при:

- монтировании (аналог componentDidMount)
- обновлении состояния или пропсов (аналог componentDidUpdate)
- размонтировании (аналог componentWillUnmount)

Срабатывает гарантированно после обновления DOM для текущего компонента.

```
export function Header(props) {
 const [isOpen, setIsOpen] = useState(false);

 useEffect(() => { // передаем в хук функцию, которая запустит эффект
   console.log('run effect here'); // сделаем в ней что-то
   return () => { // вернем функцию для отмены эффекта
     console.log('cancel effect here');
   };
 }, [ // передадим массив зависимостей
   isOpen,  // указываем, что эффект нужно запускать если изменилось значение isOpen
 ]);

 return (
  <header className="header">
    <div className="header__actions">
      <button onClick={() => setIsOpen(!isOpen)}>{isOpen ? "Close" : "Open"}</button>
    </div>
    {isOpen && <div className="header__menu">menu</div>}
  </header>
 );
}
```

Хуки - useEffect

Важные моменты:

- после монтирования будут вызваны все эффекты
- перед размонтированием будут вызваны все отмены эффектов
- если в зависимости передать пустой массив - эффект будет запущен только один раз - после монтирования
- эффекты будут запущенны сразу после обновленя DOM для текущего компонента
- код эффекта выполняется асинхронно - не блокируется отрисовка браузера

Хуки - useLayoutEffect

Работает точно так же, как и useEffect за исключением 2 моментов:

- useLayoutEffect будет запущен до useEffect
- код эффекта выполняется сихнронно - блокируется отрисовка браузера

Хуки - useContext

Позвлоляет читать значение контекста.

```
function Button(props) {
 const theme = useContext(ThemeContext);
 return (
   <button style={{color: theme.color}}>
     ...
   </button>
 );
}
```

Хуки - useCallback

Создает мемоизированный коллбек.

Если значение зависимостей не поменялось - будет использоваться одна и та же функция. При передаче ее в пропсы - ссылка будет указывать на одну и туже функцию, и React при сравнении пропрсов не будет лишний раз вызывать рендер.

Массив зависимостей не передается в качестве аргументов коллбека.

```
function Button(props) {
 const [postID, setPostID] = useState(null);

 const handleClick = useCallback(() => {
   console.log('->', postID);
 }, [postID]);

 return (
   <button onClick={handleClick}>
     ...
   </button>
 );
}
```

Хуки - useMemo

Создает мемоизированное значение. Полезно для оптимизации, по аналогии с useCallback.

Массив зависимостей не передается в качестве аргументов функции.

```
function Button(props) {
 const [factorialBorder, setFactorialBorder] = useState(1);
 const factorial = useMemo(() => {
   let result = 1;
   for (let i = 1; i <= factorialBorder; i++) {
     result *= i;
   }
   return result;
 }, [factorialBorder]);

 return (
   <button onClick={() => setFactorialBorder(factorialBorder + 1)}>
     Factorial of {factorialBorder} is {factorial}
   </button>
 );
}
```

Хуки - useRef

Позволяет сохранять что-то на протяжении всего жизненного цикла компонента. Например ссылку на DOM-узел, таймер, еще что-то. Это похоже на свойство класса.

При изменении поля current не вызывает обновление.

```
function Form() {
 let inputRef = useRef(null); // создадим реф

 useEffect(() => {
   console.log('->', inputRef.current); // читаем текущее значение рефа
 });

 return (
   <form>
     <input type="text" ref={inputRef}/> {/* Передадим реф для DOM-узла */}
   </form>
 );
}
```

Хуки - useImperativeHandle

Позволяет настраивать перенаправляемый реф. Используется вместе с forwardRef.

```
function TextInput({ label, ...rest }, ref) { // перенаправленный реф придет 2 аргументом
 const inputRef = useRef(null); // используем отдельный, внутренний реф для поля

 useImperativeHandle(ref, () => { // настроим, что будет внутри перенаправленного рефа
   return {
     node: inputRef.current,
     focus: () => inputRef.current.focus(),
     blur: () => inputRef.current.blur(),
   };
 });

 return (
   <div className="form__input">
     <label>{label}: <input {...rest} ref={inputRef}/></label>{/* Передадим внутренний реф */}
   </div>
 );
}
TextInput = React.forwardRef(TextInput); // используем перенарпавление рефа
```

Хуки - пользовательские хуки

Вы можете создавать свои хуки. При этом есть только 2 условия:

- название хука должно начинаться с *use*
- для хуков внутри пользовательскоо хука: нельзя менять порядок их вызова (нельзя вызывать внутри условий, циклов и вложенных функций)

А в остальном, пользовательские хуки - это обычные функции.

react-hook-form

Библиотека, которая ооооочень сильно упрощает работу с формами и берет на себя реализацию таких фич:

- валидация
- привязка полей через рефы
- сбор данных с полей и передача их в обработчик сабмита
- установка дефолтных значений
- логика для повторяющихся форм
- и много другое

Детальна документация тут.

react-query

Библиотека, которая позволяет удобно хранить данные с сервера и делать запросы. Задачи, которые она решает:

- хранение состояния
- кеширование
- переотправка неудачных запросов
- параллельные / последовательные запросы
- и еще много всего

Детальная документация по АПИ тут.