React
Hook and HOC

Пользовательские хуки
HOC (компонент высшего порядка)
Правила React-а
Плагин для ESLint

Пользовательские хуки

Пользовательские хуки в React — это функции, которые позволяют вам извлекать и переиспользовать логику состояния или побочных эффектов между различными компонентами. Они помогают сделать ваш код более чистым, модульным и повторно используемым.

- **Переиспользование логики:** Иногда логика состояния или эффекты могут быть одинаковыми в нескольких компонентах. Вместо того чтобы дублировать код, вы можете вынести эту логику в пользовательский хук.
- **Разделение ответственности:** Логика состояния и UI могут быть четко разделены, что делает код более понятным и поддерживаемым.
- **Изоляция кода:** Пользовательские хуки помогают изолировать побочные эффекты или логику от компонентов, что упрощает их тестирование и повторное использование.

Правила пользовательских хуков

Вы должны следовать этим соглашениям об именовании:

- Имена компонентов React должны начинаться с заглавной буквы, например, StatusBar и SaveButton. Компоненты React также должны возвращать что-то, что React умеет отображать, например, кусок JSX.
- Имена хуков должны начинаться с use, за которым следует заглавная буква, например useState (встроенный) или useOnlineStatus (пользовательский, как ранее на этой странице). Хуки могут возвращать произвольные значения.

Пользовательские хуки позволяют вам делиться логикой состояния, а не самим состоянием

Только Hooks и компоненты могут вызывать другие Hooks!

Инструкция для создания собственного хука (HOOK)

Определите цель вашего хука: Первое, что нужно сделать, — это четко понять, какую задачу будет решать ваш пользовательский хук. Обычно хуки создаются для:

- Управления состоянием.
- Взаимодействия с внешними API или браузерными API.
- Повторного использования логики.

Создайте функцию для хука: Пользовательские хуки — это обычные функции JavaScript. Имя функции должно начинаться с префикса use, чтобы React мог автоматически применять правила хуков к ней.

function useCustomHook() { // Логика хука }

Внутри функции используйте встроенные хуки React: Внутри вашей функции вы можете использовать любые встроенные хуки, такие как useState, useEffect, useCallback и т.д., для выполнения задачи вашего хука.

Возвращайте значения из хука: Решите, что ваш хук будет возвращать. Это могут быть состояния, функции для их изменения или любые другие значения. Эти возвращаемые значения будут доступны в компоненте, где хук используется.

Используйте хук в компоненте: Теперь ваш пользовательский хук готов к использованию. Его можно применять в функциональных компонентах так же, как и любой встроенный хук.

Типовые пользовательские хуки

useFetch

Хук useFetch используется для загрузки данных с сервера и управления состоянием загрузки. Он позволяет легко выполнять запросы и обрабатывать их результаты.

useLocalStorage

Хук useLocalStorage позволяет синхронизировать состояние компонента с localStorage. Это удобно для сохранения данных, которые должны сохраняться между перезагрузками страницы.

useDebounce

Хук useDebounce используется для предотвращения частых вызовов функции в течение короткого времени. Это полезно для предотвращения избыточных запросов при вводе текста в поле поиска.

useFetch

```jsx
import React from 'react';
import useFetch from './useFetch';

function App() {
  const { data, loading, error } = useFetch('https://api.example.com/data');

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error.message}</p>;

  return (
    <div>
      <h1>Data:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default App;
```

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        const result = await response.json();
        setData(result);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
}

export default useFetch;
```

useLocalStorage

```jsx
import React from 'react';
import useLocalStorage from './useLocalStorage';

function App() {
  const [name, setName] = useLocalStorage('name', '');

  const handleChange = (e) => {
    setName(e.target.value);
  };

  return (
    <div>
      <input type="text" value={name} onChange={handleChange} placeholder="Enter your name" />
      <p>Your name is: {name}</p>
    </div>
  );
}

export default App;
```

```jsx
import { useState } from 'react';

function useLocalStorage(key, initialValue) {
  const [storedValue, setStoredValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      return initialValue;
    }
  });

  const setValue = (value) => {
    try {
      const valueToStore = value instanceof Function ? value(storedValue) : value;
      setStoredValue(valueToStore);
      window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) {
      console.error(error);
    }
  };

  return [storedValue, setValue];
}

export default useLocalStorage;
```

useDebounce

```jsx
import React, { useState, useEffect } from 'react';
import useDebounce from './useDebounce';

function App() {
  const [searchTerm, setSearchTerm] = useState('');
  const debouncedSearchTerm = useDebounce(searchTerm, 500);

  useEffect(() => {
    if (debouncedSearchTerm) {
      // Выполните запрос к API или другую операцию с дебаунсированным значением
      console.log('Fetching data for:', debouncedSearchTerm);
    }
  }, [debouncedSearchTerm]);

  return (
    <div>
      <input
        type="text"
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
        placeholder="Search..."
      />
    </div>
  );
}

export default App;
```

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

function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}

export default useDebounce;
```

Пользовательский state manager (store)

Вот пример простого state manager'a для React, основанного на useSyncExternalStore. В этом примере мы создадим небольшой state manager с функциями для получения состояния, подписки на изменения и обновления состояния.

Пояснения:

- Функция getState: Возвращает текущее состояние.
- Функция setState: Обновляет состояние и уведомляет всех подписчиков об изменении.
- Функция subscribe: Позволяет компонентам подписываться на изменения состояния и возвращает функцию для отмены подписки.
- Хук useStore: Использует useSyncExternalStore для подписки на изменения состояния и выбора нужных данных из состояния.
- Компоненты Display и IncrementButton: Показывают, как можно использовать useStore для получения состояния и setState для обновления его.

Основные шаги

- Создание хранилища (store).
- Определение методов для работы с состоянием (get, set, subscribe).
- Использование useSyncExternalStore для подписки на изменения состояния.

Пример простого store-а

```
import { useSyncExternalStore } from 'react';

function useStore(store) {
  return useSyncExternalStore(
    store.subscribe,
    store.getState
  );
}
```

```
function createStore(initialState) {
  let state = initialState;
  let listeners = new Set();

  function getState() {
    return state;
  }

  function setState(newState) {
    state = newState;
    listeners.forEach(listener => listener());
  }

  function subscribe(listener) {
    listeners.add(listener);
    return () => listeners.delete(listener);
  }

  return {
    getState,
    setState,
    subscribe,
  };
}
```

```
const counterStore = createStore(0);

function Counter() {
  const count = useStore(counterStore);

  const increment = () => {
    counterStore.setState(count + 1);
  };

  const decrement = () => {
    counterStore.setState(count - 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}
```

HOC

В React **HOC (Higher-Order Component, компонент высшего порядка)** — это продвинутая техника для повторного использования логики компонентов. HOC — это функция, которая принимает компонент и возвращает новый компонент с добавленными возможностями.

HOC полезны в следующих ситуациях:

- **Повторное использование логики:** Когда вам нужно повторно использовать одну и ту же логику в нескольких компонентах, HOC позволяет инкапсулировать эту логику и использовать её в разных местах.
- **Инъекция поведения:** HOC может добавлять или изменять поведение компонента без необходимости изменять его исходный код.
- **Оборачивание компонентов:** HOC может добавлять визуальные элементы или оборачивать компонент дополнительной разметкой.

Преимущества и недостатки HOC

Преимущества:

- **Переиспользование логики:** HOC позволяют легко переиспользовать одну и ту же логику в разных компонентах.
- **Чистый код:** Компоненты остаются чистыми и фокусируются на своей основной задаче, в то время как HOC инкапсулирует дополнительную логику.

Недостатки:

- **Проблема "переноса пропсов" (Props Proxy):** HOC могут запутывать передачу пропсов, особенно если они изменяют пропсы или добавляют новые.
- **"Обёртки адские" (Wrapper Hell):** Множество вложенных HOC может усложнить структуру кода и затруднить его понимание.
- **Проблемы с рефами:** Если HOC не корректно передает ref внутрь оборачиваемого компонента, это может привести к проблемам при работе с рефами.

Инструкция для создания компонента обертки (HOC)

Определите цель вашего HOC - Прежде чем создавать HOC, определите, что именно он будет делать. HOC можно использовать для:

- Добавления общего функционала в несколько компонентов.
- Передачи дополнительных пропсов.
- Оборачивания компонентов для управления состоянием или логикой.

Создайте HOC: HOC — это функция, которая принимает компонент и возвращает новый компонент с добавленной функциональностью. HOC можно определить как обычную функцию JavaScript, которая возвращает новый компонент.

Используйте HOC в компоненте: После создания HOC вы можете использовать его, оборачивая компонент, который должен получить дополнительную функциональность.

Комбинирование нескольких HOC: Вы можете комбинировать несколько HOC, чтобы создать более сложные компоненты с несколькими слоями логики.

Типовые HOC

withLoading

HOC withLoading добавляет индикатор загрузки к компоненту. Это полезно, когда нужно показать пользователю, что данные загружаются или какой-то процесс выполняется.

withErrorBoundary

HOC withErrorBoundary оборачивает компонент в "границу ошибок", чтобы отловить любые ошибки, произошедшие в процессе рендеринга, и показать пользователю сообщение об ошибке.

withAuthorization

HOC withAuthorization используется для управления доступом к компонентам на основе определенных условий, таких как права пользователя.

HOC withLoading

```jsx
import React from 'react';

// HOC
function withLoading(Component) {
  return function WithLoadingComponent({ isLoading, ...props }) {
    if (isLoading) {
      return <div>Loading...</div>;
    }
    return <Component {...props} />;
  };
}

export default withLoading;
```

```jsx
import React, { useState, useEffect } from 'react';
import withLoading from './withLoading';

function MyComponent({ data }) {
  return (
    <div>
      <h1>Data:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

const MyComponentWithLoading = withLoading(MyComponent);

function App() {
  const [data, setData] = useState(null);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    setTimeout(() => {
      setData({ name: 'John Doe' });
      setIsLoading(false);
    }, 2000);
  }, []);

  return <MyComponentWithLoading isLoading={isLoading} data={data} />;
}

export default App;
```

withErrorBoundary

```jsx
import React from 'react';
import withErrorBoundary from './withErrorBoundary';

function MyComponent() {
  // Вызовет ошибку для демонстрации
  throw new Error('Test Error');
  return <div>Content</div>;
}

const MyComponentWithErrorBoundary = withErrorBoundary(MyComponent);

function App() {
  return <MyComponentWithErrorBoundary />;
}

export default App;
```

```jsx
import React from 'react';

// Error Boundary Component
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.error('Error caught by ErrorBoundary:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <div>Something went wrong.</div>;
    }

    return this.props.children;
  }
}

// HOC
function withErrorBoundary(Component) {
  return function WithErrorBoundaryComponent(props) {
    return (
      <ErrorBoundary>
        <Component {...props} />
      </ErrorBoundary>
    );
  };
}

export default withErrorBoundary;
```

withAuthorization

```jsx
import React from 'react';

// HOC
function withAuthorization(Component, allowedRoles) {
  return function WithAuthorizationComponent({ user, ...props }) {
    if (!allowedRoles.includes(user.role)) {
      return <div>Access Denied</div>;
    }
    return <Component {...props} />;
  };
}

export default withAuthorization;
```

```jsx
import React from 'react';
import withAuthorization from './withAuthorization';

function AdminPanel() {
  return <div>Welcome to the Admin Panel</div>;
}

const AdminPanelWithAuthorization = withAuthorization(AdminPanel, ['admin']);

function App() {
  const user = { role: 'user' }; // Можно изменить на 'admin' для доступа

  return <AdminPanelWithAuthorization user={user} />;
}

export default App;
```

Плагин для ESLint

React поставляется с утилитой eslint-plugin-react-hooks, которая может помочь автоматизировать проверку соблюдения этих правил. Она выдает предупреждения и ошибки, если хуки используются неправильно.

Соблюдение правил хуков — это не только требование, но и способ убедиться, что ваше приложение работает стабильно и предсказуемо. Они обеспечивают единообразие и помогают избежать множества ошибок, особенно в сложных приложениях. Соблюдение этих правил также делает код более понятным для других разработчиков, что способствует лучшей поддержке и развитию проекта.

```
{
  "plugins": ["react-hooks"],
  "rules": {
    "react-hooks/rules-of-hooks": "error", // Проверяет правила хуков
    "react-hooks/exhaustive-deps": "warn"  // Проверяет зависимости эффектов
  }
}
```

Команда для установки

npm install eslint-plugin-react-hooks --save-dev

Команда для проверки

npm run lint

npx eslint .

Правила React-а

Как в разных языках программирования есть свои способы выражения концепций, так и в React есть свои идиомы - или правила - для выражения паттернов таким образом, чтобы их было легко понять и получить качественные приложения.

Основные правила:

- **Компоненты и хуки должны быть чистыми**
- **React вызывает компоненты и хуки**
- **Правила использования хуков**

Разработчики React настоятельно рекомендуют использовать Strict Mode вместе с плагином ESLint для React, чтобы помочь вашей кодовой базе следовать правилам React. Следуя правилам React, вы сможете найти и устранить эти ошибки и сохранить работоспособность вашего приложения.

Компоненты и хуки должны быть чистыми

Чистые функции выполняют только вычисления и ничего больше. Это облегчает понимание и отладку кода, а также позволяет React автоматически оптимизировать компоненты и хуки.

- **Компоненты должны быть идемпотентными** - предполагается, что компоненты React всегда возвращают один и тот же результат относительно своих входов - props, state и context.
- **Побочные эффекты должны выполняться вне рендера** - Побочные эффекты не должны выполняться во время рендера, так как React может рендерить компоненты несколько раз, чтобы создать наилучший пользовательский опыт.
- **Пропсы и состояние неизменяемы** - Пропсы и состояние компонента являются неизменяемыми моментальными снимками по отношению к одному рендеру. Никогда не изменяйте их напрямую.
- **Возвращаемые значения и аргументы хуков неизменяемы** - После передачи значений в хук их не следует изменять. Как и пропсы в JSX, значения становятся неизменяемыми при передаче в хук.
- **Значения неизменяемы после передачи в JSX** - Не изменяйте значения после того, как они были использованы в JSX. Переместите мутацию до создания JSX.

React вызывает компоненты и хуки

React отвечает за рендеринг компонентов и хуков, когда это необходимо для оптимизации пользовательского опыта. Он является декларативным: вы указываете React, что нужно отобразить в логике вашего компонента, а React сам решает, как лучше показать это пользователю.

Никогда не вызывайте функции компонентов напрямую

Никогда не передавайте хуки как обычные значения

Никогда не вызывайте функции компонентов напрямую

Компоненты должны использоваться только в JSX. Не вызывайте их как обычные функции. Их должен вызывать React.

React должен решить, когда будет вызвана функция вашего компонента во время рендеринга. В React вы делаете это с помощью JSX.

Позволяя React оркестровать рендеринг, вы также получаете ряд преимуществ:

- **Компоненты становятся больше чем функциями.** React может дополнить их такими возможностями, как локальное состояние с помощью хуков, которые привязаны к идентификатору компонента в дереве.
- **Типы компонентов участвуют в согласовании.** Позволяя React вызывать ваши компоненты, вы также сообщаете ему больше о концептуальной структуре вашего дерева. Например, когда вы переходите от рендеринга <Feed> к странице <Profile>, React не будет пытаться использовать их повторно.
- **React может улучшить пользовательский опыт.** Например, он может позволить браузеру выполнять некоторую работу между вызовами компонентов, чтобы повторный рендеринг большого дерева компонентов не блокировал основной поток.
- **Лучшая история отладки.** Если компоненты являются гражданами первого класса, о которых знает библиотека, мы можем создать богатые инструменты разработчика для интроспекции в процессе разработки.
- **Более эффективное согласование.** React может точно определить, какие компоненты в дереве нуждаются в повторном рендеринге, и пропустить те, которые не нуждаются. Это делает ваше приложение более быстрым и шустрым.

```
function BlogPost() {
    return (
        <Layout>
            <Article />
        </Layout>
    ); // ✅ Good: Only use components in JSX
}
```

```
function BlogPost() {
    return <Layout>{Article()}</Layout>; // 🔴 Bad: Never call them directly
}
```

Никогда не передавайте хуки как обычные значения

Хуки должны вызываться только внутри компонентов или хуков. Никогда не передавайте их как обычные значения.

Хуки позволяют дополнить компонент функциями React. Они всегда должны вызываться как функция и никогда не передаваться как обычное значение. Это позволяет использовать локальные рассуждения, или возможность для разработчиков понять все, что может делать компонент, рассматривая его в отдельности.

Нарушение этого правила приведет к тому, что React не будет автоматически оптимизировать ваш компонент.

- **Не мутируйте динамически хуки**
- **Не используйте хуки динамически**

```
function ChatInput() {
    // 🔴 Bad: don't write higher order Hooks
    const useDataWithLogging = withLogging(useData);
    const data = useDataWithLogging();
}
```

```
function ChatInput() {
    // ✅ Good: Create a new version of the Hook
    const data = useDataWithLogging();
}

function useDataWithLogging() {
    // ... Create a new version of the Hook and inline the logic here
}
```

```
function ChatInput() {
    // 🔴 Bad: don't pass Hooks as props
    return <Button useData={useDataWithLogging} />;
}
```

```
function ChatInput() {
    return <Button />;
}

function Button() {
    const data = useDataWithLogging(); // ✅ Good: Use the Hook directly
}

function useDataWithLogging() {
    // If there's any conditional logic to change the Hook's behavior,
    // it should be inlined into the Hook
}
```

Правила работы с хуками

Хуки определяются с помощью функций JavaScript, но они представляют собой особый тип многократно используемой логики пользовательского интерфейса с ограничениями на то, где они могут быть вызваны.

Вызывайте хуки только на верхнем уровне

Вызывайте хуки только из функций React

Пользовательские хуки могут вызывать другие хуки (в этом и заключается их назначение). Это работает, потому что пользовательские хуки также должны вызываться только во время рендеринга компонента функции.

Вызывайте хуки только на верхнем уровне

Не вызывайте хуки внутри циклов, условий, вложенных функций или блоков try/catch/finally. Вместо этого всегда используйте хуки на верхнем уровне вашей функции React, перед любым ранним возвратом. Вы можете вызывать хуки только во время рендеринга компонента функции React:

✅ Вызывайте их на верхнем уровне в теле функционального компонента.

✅ Вызовите их на верхнем уровне в теле пользовательского хука.

🔴 Не вызывайте хуки внутри условий или циклов.

🔴 Не вызывайте хуки после условного оператора возврата.

🔴 Не вызывайте хуки в обработчиках событий.

🔴 Не вызывайте хуки в компонентах классов.

🔴 Не вызывайте хуки в функциях, передаваемых в useMemo, useReducer или useEffect.

🔴 Не вызывайте хуки внутри блоков try/catch/finally.

```
function Counter() {
    // ✅ Good: top-level in a function component
    const [count, setCount] = useState(0);
    // ...
}

function useWindowWidth() {
    // ✅ Good: top-level in a custom Hook
    const [width, setWidth] = useState(window.innerWidth);
    // ...
}
```

```
function Bad({ cond }) {
    if (cond) {
        // 🔴 Bad: inside a condition (to fix, move it outside!)
        const theme = useContext(ThemeContext);
    }
    // ...
}

function Bad() {
    for (let i = 0; i < 10; i++) {
        // 🔴 Bad: inside a loop (to fix, move it outside!)
        const theme = useContext(ThemeContext);
    }
    // ...
}

function Bad({ cond }) {
    if (cond) {
        return;
    }
    // 🔴 Bad: after a conditional return (to fix, move it before the return!)
    const theme = useContext(ThemeContext);
    // ...
}

function Bad() {
    function handleClick() {
        // 🔴 Bad: inside an event handler (to fix, move it outside!)
        const theme = useContext(ThemeContext);
    }
    // ...
}

function Bad() {
    const style = useMemo(() => {
        // 🔴 Bad: inside useMemo (to fix, move it outside!)
        const theme = useContext(ThemeContext);
        return createStyle(theme);
    });
    // ...
}

class Bad extends React.Component {
    render() {
        // 🔴 Bad: inside a class component (to fix, write a function component instead of a class!)
        useEffect(() => {});
        // ...
    }
}

function Bad() {
    try {
        // 🔴 Bad: inside try/catch/finally block (to fix, move it outside!)
        const [x, setX] = useState(0);
    } catch {
        const [x, setX] = useState(1);
    }
}
```

Вызывайте хуки только из функций React

Не вызывайте хуки из обычных функций JavaScript. Вместо этого вы можете:

✅ Вызывать хуки из компонентов функций React.

✅ Вызывать хуки из пользовательских хуков.

Следуя этому правилу, вы гарантируете, что вся логика с состоянием в компоненте будет четко видна из его исходного кода.

```
function FriendList() {
    const [
        onlineStatus,
        setOnlineStatus,
    ] = useOnlineStatus(); // ✅
}

function setOnlineStatus() {
    // ❌ Not a component or custom Hook!
    const [
        onlineStatus,
        setOnlineStatus,
    ] = useOnlineStatus();
}
```

Ресурсы

Код с урока: исходник - ТЫК | деплой - ТЫК

Создание пользовательских хуков - ТЫК

Переиспользование логики с помощью пользовательских хуков - ТЫК

Правила хуков - ТЫК

Мыслим как React - ТЫК

Компоненты и хуки должны быть чистыми - ТЫК

React вызывает компоненты и хуки - ТЫК

Правила работы с хуками - ТЫК

Компоненты высшего порядка - ТЫК

Библиотека пользовательских хуков (**react-use**) - ТЫК

“Склад” пользовательских хуков (**useHooks**) - ТЫК