Правила хуков

Хуки — нововведение в React 16.8, которое позволяет использовать состояние и другие возможности React без написания классов.

Хуки — обычные JavaScript-функции, но существует два правила, которым нужно следовать. Чтобы автоматически их применять мы создали плагин для линтера:

Используйте хуки только на верхнем уровне

Не используйте хуки внутри циклов, условных операторов или вложенных функций. Вместо этого всегда используйте хуки только на верхнем уровне React-функций. Исполнение этого правила гарантирует, что хуки вызываются в одинаковой последовательности при каждом рендере компонента. Это позволит React правильно сохранять состояние хуков между множественными вызовами `useState` и `useEffect`. (Если вам интересно, подробное объяснение ниже.)

Вызывайте хуки только из React-функций

Не вызывайте хуки из обычных функций JavaScript. Вместо этого можно:

- ✅ Вызывать хуки из функционального компонента React.
- ✅ Вызывать хуки из пользовательского хука (мы научимся делать это на следующей странице).

Следуя этому правилу, можно гарантировать, что вся логика состояния компонента чётко видна из исходного кода.

Плагин для ESLint

Мы выпустили плагин для ESLint `eslint-plugin-react-hooks`, который принуждает к соблюдению этих двух правил. Если хотите испытать его в деле, добавьте этот плагин в ваш проект.

```
npm install eslint-plugin-react-hooks --save-dev
// Конфигурация ESLint
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Проверяем правила хуков
    "react-hooks/exhaustive-deps": "warn" // Проверяем зависимости эффекта
  }
}
```

Этот плагин включён по умолчанию в Create React App.

Вы можете пропустить остаток этой страницы и перейти к созданию собственного хука. Но если вам интересно, ниже приведено объяснение, почему правила хуков необходимы.

Объяснение

Как мы ранее узнали, хуки состояния или эффектов в одном и том же компоненте можно использовать многократно:

```
function Form() {
  // 1. Используем переменную состояния name
  const [name, setName] = useState('Мэри');

  // 2. Используем эффект для сохранения данных формы
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Используем переменную состояния surname
  const [surname, setSurname] = useState('Поппинс');

  // 4. Используем эффект для обновления заголовка страницы
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}
```

Итак, как же React сопоставляет переменные состояния с вызовами `useState`? Ответ таков: **React полагается на порядок вызова хуков**. Наш пример работает, потому что порядок вызова хуков одинаков при каждом рендере.

```
// ------------
// Первый рендер
// ------------
useState('Мэри')           // 1. Инициализируем переменную name значением 'Mary'
useEffect(persistForm)     // 2. Добавляем эффект для сохранения данных формы
useState('Поппинс')        // 3. Инициализируем переменную surname значением
'Poppins'
useEffect(updateTitle)     // 4. Добавляем эффект для обновления заголовка
страницы

// -------------
// Второй рендер
// -------------
useState('Мэри')           // 1. Читаем переменную состояния name (аргумент
игнорируется)
useEffect(persistForm)     // 2. Заменяем эффект сохранения данных формы
useState('Поппинс')        // 3. Читаем переменную состояния surname (аргумент
игнорируется)
useEffect(updateTitle)     // 4. Заменяем эффект обновления заголовка страницы

// ...
```

До тех пор пока порядок вызова хуков одинаков в каждом рендере, React может сопоставить некое внутреннее состояние с каждым из них. Но что случится, если мы поместим вызов хука (например, эффект `persistForm`) внутрь условного оператора?

```
  // ● Нарушаем первое правило, помещая хук в условие
  if (name !== '') {
    useEffect(function persistForm() {
      localStorage.setItem('formData', name);
    });
  }
```

Условие `name !== ''` равняется `true` при первом рендере, поэтому хук выполняется. Тем не менее, при следующем рендере пользователь может обратить это условие в `false`, очистив поля формы. Теперь во время рендера хук будет пропущен и порядок вызовов хуков изменится.

```
useState('Мэри')           // 1. Читаем переменную состояния name (аргумент
игнорируется)
// useEffect(persistForm)  // ● Хук пропускается!
useState('Поппинс')        // ● 2 (но ранее был 3). Ошибка при чтении переменной
состояния surname
useEffect(updateTitle)     // ● 3 (но ранее был 4). Ошибка при замене эффекта
```

React не будет знать, что вернуть для второго вызова хука `useState`. React ожидал, что второй вызов хука в этом компоненте соответствует

эффекту `persistForm`, так же как при предыдущем рендере, но это больше не так. Начиная с этого момента, вызов каждого хука, следующего за пропущенным, также будет сдвинут на один назад, что приведёт к ошибкам.

Вот почему хуки должны вызываться на верхнем уровне компонента. Если мы хотим запускать эффект по условию, то можем поместить это условие *внутрь* хука:

```
useEffect(function persistForm() {
  // 👍 Первое правило больше не нарушается
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

Эта проблема не будет вас беспокоить, если вы включите в свой проект наше правило линтера. Но теперь вы знаете, *почему* хуки работают таким образом и какие проблемы это правило предотвращает.