React - компоненты

Контролируемые и неконтролируемые компоненты
React вспомогательные API
Встроенные компоненты
Передача компонентов (children)

Контролируемые компоненты

Контролируемые компоненты — это компоненты, в которых состояние формы (например, значения полей ввода) контролируется React. Это означает, что React отслеживает состояние данных и обновляет их через state. Таким образом, каждое изменение в поле формы сопровождается обновлением состояния в компоненте.

Здесь состояние value контролируется компонентом, и каждое изменение в поле ввода вызывает обновление состояния через setValue.

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

function ControlledComponent() {
  const [value, setValue] = useState('');

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

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

export default ControlledComponent;
```

Неконтролируемые компоненты

Неконтролируемые компоненты, напротив, не управляют состоянием формы через React. Вместо этого используется прямой доступ к DOM-элементу с помощью рефов (refs), чтобы получить текущее значение поля ввода.

Здесь React не управляет состоянием формы, а для получения значения используется реф inputRef.

В чем разница?

- Контролируемые компоненты: React полностью управляет состоянием формы, что делает компонент более предсказуемым и удобным для валидации и синхронизации данных.
- Неконтролируемые компоненты: состояние контролируется непосредственно в DOM, что может быть проще в простых случаях, но делает компонент менее управляемым и сложнее в поддержке.

```jsx
import React, { useRef } from 'react';

function UncontrolledComponent() {
  const inputRef = useRef(null);

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Input value: ${inputRef.current.value}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}

export default UncontrolledComponent;
```

React вспомогательные API

Помимо Hooks и Components, пакет react экспортирует еще несколько API, полезных для определения компонентов.

- **createContext** позволяет определить и предоставить контекст дочерним компонентам. Используется вместе с **useContext**.
- **forwardRef** позволяет вашему компоненту отображать узел DOM как ссылку на родительский. Используется с **useRef**.
- **lazy** позволяет отложить загрузку кода компонента до его первого отображения.
- **memo** позволяет компоненту пропускать повторные рендеры с теми же реквизитами. Используется с **useMemo** и **useCallback**.
- **startTransition** позволяет пометить обновление состояния как несрочное. Аналогично **useTransition**.

createContext

createContext позволяет вам создать контекст, который компоненты могут предоставить или прочитать.

const SomeContext = createContext(defaultValue);

createContext возвращает объект контекста.

Сам объект контекста не содержит никакой информации. Он представляет какой контекст читают или предоставляют другие компоненты.

- **SomeContext.Provider** позволяет вам предоставлять значение контекста компонентам.
- **SomeContext.Consumer** является альтернативным и редко используемым способом чтения значения контекста.

createContext возвращает объект контекста. Компоненты могут читать контекст, передавая его в **useContext()**

forwardRef

forwardRef позволяет вашему компоненту передать узел DOM родительскому компоненту с помощью ref.

const SomeComponent = forwardRef(render);

forwardRef возвращает React-компонент, который вы можете отобразить в JSX. В отличие от компонентов React, определяемых как простые функции, компонент, возвращаемый **forwardRef**, также может принимать пропс ref.

lazy

lazy позволяет отложить загрузку кода компонента до его первого отображения.

const SomeComponent = lazy(load);

lazy возвращает компонент React, который вы можете отобразить в своем дереве. Пока код ленивого компонента загружается, попытка его рендеринга будет приостановлена. Используйте **<Suspense>** для отображения индикатора загрузки во время загрузки.

memo

memo позволяет пропустить повторное отображение компонента, если его пропсы неизменны.

const MemoizedComponent = memo(SomeComponent, arePropsEqual?)

memo возвращает новый компонент React. Он ведет себя так же, как и компонент, переданный в memo, за исключением того, что React не будет всегда перерисовывать его, когда перерисовывается его родитель, если его пропсы не изменились.

startTransition

startTransition позволяет обновлять состояние без блокировки пользовательского интерфейса.

startTransition(scope);

Функция **startTransition** позволяет пометить обновление состояния как переход. **startTransition** ничего не возвращает.

Встроенные компоненты

React предлагает несколько встроенных компонентов, которые вы можете использовать в своем JSX.

- **<Fragment>**, альтернативно записываемый как **<>...</>**, позволяет группировать несколько узлов JSX вместе.
- **<Profiler>** позволяет программно измерить производительность рендеринга дерева React.
- **<Suspense>** позволяет отображать откат во время загрузки дочерних компонентов.
- **<StrictMode>** включает дополнительные проверки, предназначенные только для разработчиков, которые помогают находить ошибки на ранней стадии.

Fragment

<Fragment>, часто используемый через синтаксис <>...</>, позволяет группировать элементы без узла-обертки.

Оберните элементы в <Fragment>, чтобы сгруппировать их вместе в ситуациях, когда вам нужен один элемент. Группировка элементов в Fragment не влияет на результирующий DOM; он такой же, как если бы элементы не были сгруппированы. Пустой JSX-тег <></> в большинстве случаев является сокращением для <Fragment></Fragment>.

```
function Post() {
    return (
        <>
            <PostTitle />
            <PostBody />
        </>
    );
}
```

Profiler

<Profiler> позволяет программно измерить производительность рендеринга дерева React.

Пропсы

- **id:** Строка, идентифицирующая часть пользовательского интерфейса, которую вы измеряете.
- **onRender:** Обратный вызов onRender, который React вызывает каждый раз, когда компоненты в профилированном дереве обновляются. Он получает информацию о том, что было отрисовано и сколько времени это заняло.

Профилирование добавляет дополнительные накладные расходы, поэтому по умолчанию оно отключено в производственной сборке.

```
<Profiler id="App" onRender={onRender}>
    <App />
</Profiler>
```

```
function onRender(
    id,
    phase,
    actualDuration,
    baseDuration,
    startTime,
    commitTime
) {
    // Aggregate or log render timings...
}
```

Profiler callback - onRender

React будет вызывать ваш обратный вызов onRender с информацией о том, что было отрисовано.

- id: Строковый id пропс дерева <Profiler>, которое только что было зафиксировано. Это позволяет определить, какая часть дерева была зафиксирована, если вы используете несколько профилировщиков.
- phase: "mount", "update" или "nested-update". Это позволяет узнать, было ли дерево только что смонтировано в первый раз или было перерендерировано из-за изменения пропсов, состояния или хуков.
- actualDuration: Количество миллисекунд, потраченных на рендеринг <Profiler> и его потомков для текущего обновления. Это показывает, насколько хорошо поддерево использует мемоизацию (например, memo и useMemo). В идеале это значение должно значительно уменьшиться после первоначального монтажа, поскольку многие потомки будут нуждаться в повторном рендеринге только в случае изменения их специфических пропсов.
- baseDuration: Число миллисекунд, определяющее, сколько времени потребуется для повторного отображения всего поддерева <Profiler> без каких-либо оптимизаций. Оно вычисляется путем суммирования последних длительностей рендеринга каждого компонента в дереве. Это значение оценивает стоимость рендеринга в худшем случае (например, при первоначальном монтаже или для дерева без мемоизации). Сравните actualDuration с ним, чтобы узнать, работает ли мемоизация.
- startTime: Числовая метка времени, когда React начал рендеринг текущего обновления.
- endTime: Числовая метка времени, когда React зафиксировал текущее обновление. Это значение разделяется между всеми профилировщиками в коммите, что позволяет группировать их при желании.

Suspense

<Suspense> позволяет отображать фалбэк до тех пор, пока его дочерние элементы не закончат загрузку.

Свойства

- **children:** Фактический пользовательский интерфейс, который вы собираетесь рендерить. Если children приостановится во время рендеринга, граница Suspense переключится на рендеринг fallback.
- **fallback:** Альтернативный пользовательский интерфейс, который будет отображаться вместо реального пользовательского интерфейса, если он не закончил загрузку. Принимается любой допустимый узел React, хотя на практике запасной вариант - это легковесное представление-заполнитель, например, загрузочный спиннер или скелет. Приостановка будет автоматически переключаться на fallback, когда children приостанавливает работу, и обратно на children, когда данные будут готовы. Если fallback приостанавливает работу во время рендеринга, он активирует ближайшую родительскую границу Suspense.

```jsx
import React, { Suspense } from 'react';

// Динамическая загрузка компонента
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <h1>My App</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

Suspense нужен ли?

Когда компонент внутри Suspense сталкивается с асинхронной операцией, он сообщает об этом Suspense, который временно прерывает рендеринг и показывает запасной интерфейс (fallback). Как только асинхронная операция завершена, Suspense продолжает рендеринг компонента.

Плюсы:

- Улучшение UX: Позволяет пользователям видеть, что приложение загружает данные или компоненты, вместо того чтобы оставлять пустой экран.
- Оптимизация производительности: Рендеринг компонента начинается только после завершения асинхронной задачи, что снижает нагрузку на приложение.
- Гибкость: Позволяет легко обрабатывать различные состояния загрузки данных и компонентов.

Минусы:

- Ограниченная поддержка: На данный момент Suspense полностью поддерживает только динамическую загрузку компонентов. Поддержка асинхронной загрузки данных (например, через API) пока ограничена.
- Усложнение структуры: Использование Suspense может усложнить структуру компонентов, особенно если они вложены.

StrictMode

Режим **<StrictMode>** позволяет находить распространенные ошибки в компонентах на ранних стадиях разработки.

Используйте StrictMode для включения дополнительных поведений разработки и предупреждений для внутреннего дерева компонентов.

Строгий режим включает следующие модели поведения, доступные только для разработчиков:

- Ваши компоненты будут перерендериваться дополнительно для поиска ошибок, вызванных нечистым рендерингом.
- Ваши компоненты будут перезапускать эффекты дополнительно, чтобы найти ошибки, вызванные отсутствием очистки эффектов.
- Ваши компоненты будут проверяться на использование устаревших API.

Все эти проверки предназначены только для разработки и не влияют на производственную сборку.

Передача компонентов (children)

children в React — это специальный проп, который позволяет передавать дочерние элементы компоненту. Этот проп является ключевым в создании гибких и переиспользуемых компонентов, позволяя разработчикам вложить любые React-элементы внутрь другого компонента.

```
const Wrapper = ({ children }) => {
  return <div className="wrapper">{children}</div>;
};

const App = () => {
  return (
    <Wrapper>
      <p>This is a child element</p>
    </Wrapper>
  );
};
```

Рендеринг динамического контента

children можно использовать для рендеринга динамического контента внутри компонентов, что полезно для создания компонентов-контейнеров или макетов.

```
const Layout = ({ header, content, footer }) => {
  return (
    <div>
      <header>{header}</header>
      <main>{content}</main>
      <footer>{footer}</footer>
    </div>
  );
};

const App = () => {
  return (
    <Layout
      header={<h1>Header</h1>}
      content={<p>Main content here</p>}
      footer={<small>Footer text</small>}
    />
  );
};
```

Работа с массивами детей:

Если children содержит массив элементов, их можно перебрать с помощью стандартных методов массива.

React.Children.map — это специальный метод, который помогает безопасно работать с children в массиве, учитывая возможные случаи, когда children может быть единичным элементом или даже null.

```
const List = ({ children }) => {
  return <ul>{React.Children.map(children, (child) => <li>{child}</li>)}</ul>;
};

const App = () => {
  return (
    <List>
      <span>Item 1</span>
      <span>Item 2</span>
      <span>Item 3</span>
    </List>
  );
};
```

Передача функций как дочерних элементов

Вы можете передавать функцию в children, что позволяет создавать компоненты высшего порядка (HOC) и паттерн "Render Props":

```
const DataProvider = ({ children }) => {
  const data = { name: "John", age: 30 };
  return children(data);
};

const App = () => {
  return (
    <DataProvider>
      {(data) => <div>{`Name: ${data.name}, Age: ${data.age}`}</div>}
    </DataProvider>
  );
};
```

Взаимодействие с React.Children

React.Children.map: Итерирует по каждому элементу children и вызывает функцию обратного вызова для каждого элемента.

React.Children.forEach: Аналогично map, но не возвращает массив.

React.Children.count: Возвращает количество элементов в children.

React.Children.only: Проверяет, что children содержит только один элемент, и возвращает его. Если элементов больше или меньше одного, выбрасывается ошибка.

React.Children.toArray: Преобразует children в массив, что удобно для обработки.

Ресурсы

Код урока: исходный код - ТЫК | деплой - ТЫК

createContext документация - ТЫК

forwardRef документация - ТЫК

lazy документация - ТЫК

memo документация - ТЫК

startTransition документация - ТЫК

Fragment документация - ТЫК

Profiler документация - ТЫК

Suspense документация - ТЫК

StrictMode документация - ТЫК

Children документация - ТЫК