ASTON

Интенсив-курс по React JS

astondevs.ru

Занятие 4. Hooks and routers

1. HOOKS дополнительные
2. Custom Hooks (Пользовательские хуки)
3. React-router

useLayoutEffect()

Хук useLayoutEffect() похож на хук useEffect(), за исключением того, что он запускает эффект перед отрисовкой компонента.

Данный хук предназначен для запуска эффектов, влияющих на внешний вид DOM, незаметно для пользователя. Эта функция имеет такую же сигнатуру, что и useEffect().

В подавляющем большинстве случаев для запуска побочных эффектов используется useEffect().

useLayoutEffect()

Пример использования хука useLayoutEffect()

```
const LayoutEffect = () => {
  const [randomInt, setRandomInt] = useState(0)
  const [effectLogs, setEffectLogs] = useState([])
  const [count, setCount] = useState(1)

  useLayoutEffect(() => {
    setEffectLogs((prev) => [...prev, `Вызов функции номер ${count}.`])

    setCount(count + 1)
  }, [randomInt])

  return (
    <>
      <h3>{randomInt}</h3>
      <button onClick={() => setRandomInt(~~(Math.random() * 10))}>
        Получить случайное целое число!
      </button>
      <ul>
        {effectLogs.map((effect, i) => (
          <li key={i}>{' 😈 '.repeat(i) + effect}</li>
        ))}
      </ul>
    </>
  )
}
```

useContext()

Принимает объект контекста (значение, возвращенное из React.createContext) и возвращает текущее значение контекста для этого контекста.
Текущее значение контекста определяется пропом value ближайшего <MyContextProvider> над вызывающим компонентом в дереве.

Когда ближайший <MyContextProvider> над компонентом обновляется, этот хук вызовет повторный рендер с последним значением контекста, переданным этому провайдеру MyContext. Даже если родительский компонент использует React.memo или реализует shouldComponentUpdate, то повторный рендер будет выполняться, начиная с контекста, использующего useContext.

useContext()

Создание контекста

```
import { createContext } from 'react'

export const ContextName = createContext()
```

Передача значения контекста нижележащим компонентам:

```
<ContextName.Provider value={initialValue}>
  <App />
</ContextName.Provider>
```

Получение значения контекста:

```
import { useContext } from 'react'
import { ContextName } from './ContextName'

const contextValue = useContext(ContextName)
```

useContext()

Пример использования хука useContext()

```
const ChangeTheme = () => {
  const [mode, setMode] = useState('light')

  const handleClick = () => {
    setMode(mode === 'light' ? 'dark' : 'light')
  }

  const ThemeContext = createContext(mode)

  const theme = useContext(ThemeContext)

  return (
    <div
      style={{
        background: theme === 'light' ? '#eee' : '#222',
        color: theme === 'light' ? '#222' : '#eee',
        display: 'grid',
        placeItems: 'center',
        minWidth: '320px',
        minHeight: '320px',
        borderRadius: '4px'
      }}
    >
      <p>Выбранная тема: {theme}.</p>
      <button onClick={handleClick}>Поменять тему оформления</button>
    </div>
  )
}
```

useMemo()

Хук useMemo() является альтернативой хука useCallback(), но принимает любые значения, а не только функции.

```
useMemo(() => {
  fn,
  [deps]
}) // deps - зависимости
```

Вы можете использовать useMemo как оптимизацию производительности, а не как семантическую гарантию. В будущем React может решить «забыть» некоторые ранее мемоизированные значения и пересчитать их при следующем рендере, например, чтобы освободить память для компонентов вне области видимости экрана.

```
const memoizedValue = useMemo( factory: () => computeExpensiveValue(a, b),  deps: [a, b]);
```

useMemo()

Пример использования хука useMemo()

```jsx
const BasicMemo = () => {
  const [age, setAge] = useState(19)

  const handleClick = () => { setAge(age < 100 ? age + 1 : age) }

  const getRandomColor = () => `#${((Math.random() * 0xfff) << 0).toString(16)}`

  const memoizedGetRandomColor = useMemo(() => getRandomColor, [])

  return (
    <>
      <Age age={age} handleClick={handleClick} />
      <Guide getRandomColor={memoizedGetRandomColor} />
    </>
  )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <p>Мне {age} лет.</p>
      <p>Нажми на кнопку 👇</p>
      <button onClick={handleClick}>Стать старше!</button>
    </div>
  )
}

const Guide = memo(({ getRandomColor }) => {
  const color = getRandomColor()

  return (
    <div style={{ background: color, padding: '.4rem' }}>
      <p style={{ color: color, filter: 'invert()' }}>
        Следуй инструкциям максимально точно.
      </p>
    </div>
  )
})
```

useCallback()

Хук useCallback() возвращает мемоизированную версию переданной функции обратного вызова. Данный хук принимает колбек и массив зависимостей. колбек повторно вычисляется только при изменении значений одной из зависимостей.

```
useCallback(
  fn,
  [deps]
) // deps - dependencies, зависимости
```

```
24  function Counter({a, b}) {
25
26    const memoizedCallback = useCallback(
27      callback: () => {
28        doSomething(a, b);
29      },
30      deps: [a, b],
31    );
```

Передайте встроенный колбэк и массив зависимостей. Хук useCallback вернёт мемоизированную версию колбэка, который изменяется только, если изменяются значения одной из зависимостей. Это полезно при передаче колбэков оптимизированным дочерним компонентам, которые полагаются на равенство ссылок для предотвращения ненужных рендеров например, shouldComponentUpdate

useCallback()

Пример использования хука useCallback()

```jsx
const BasicCallback = () => {
  const [age, setAge] = useState(19)

  const handleClick = () => { setAge(age < 100 ? age + 1 : age) }

  const getRandomColor = useCallback(
    () => `#${((Math.random() * 0xfff) << 0).toString(16)}`,
    []
  )

  return (
    <>
      <Age age={age} handleClick={handleClick} />
      <Guide getRandomColor={getRandomColor} />
    </>
  )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <p>Мне {age} лет.</p>
      <p>Нажми на кнопку 👇</p>
      <button onClick={handleClick}>Стать старше!</button>
    </div>
  )
}

// `React.memo()` позволяет зафиксировать состояние компонента
const Guide = memo(({ getRandomColor }) => {
  const color = getRandomColor()

  return (
    <div style={{ background: color, padding: '.4rem' }}>
      <p style={{ color: color, filter: 'invert()' }}>
        Следуй инструкциям максимально точно.
      </p>
    </div>
  )
})
```

Важно! useCallback(fn, deps) — это эквивалент useMemo(() => fn, deps)

React.memo()

React.memo — это компонент высшего порядка.

Если компонент всегда рендерит одно и то же при неменяющихся пропсах, можно обернуть его в вызов React.memo для повышения производительности в некоторых случаях, мемоизируя тем самым результат. Это значит, что React будет использовать результат последнего рендера, избегая повторного рендеринга.

```
const MyComponent = React.memo(function MyComponent(props) {
  /* рендер с использованием пропсов */
});
```

React.memo()

Чтобы настроить сравнение свойств, вы можете использовать второй аргумент, чтобы указать функцию проверки равенства

```
React.memo(Component, [areEqual(prevProps, nextProps)]);
```

areEqual(prevProps, nextProps) функция должна возвращать - ***true*** если prevProps и nextProps равны

```
function MyComponent(props) {
  /* рендер с использованием пропсов */
}
function areEqual(prevProps, nextProps) {
  /*
  возвращает true, если nextProps рендерит
  тот же результат что и prevProps,
  иначе возвращает false
  */
}
export default React.memo(MyComponent, areEqual);
```

useRef()

useRef возвращает изменяемый ref объект, свойство .current которого инициализируется переданным аргументом (initialValue). Возвращённый объект будет сохраняться в течение всего времени жизни компонента.

Данный хук также может использоваться для сохранения любого мутирующего значения.

Создание хука: `const node = useRef()`.

Добавление ссылки: `<tagName ref={node}></tagName>`.

```
const refContainer = useRef(initialValue);
```

useRef()

Пример использования хука useRef()

```jsx
const DomAccess = () => {
  const textareaEl = useRef(null)

  const handleClick = () => {
    textareaEl.current.value = 'Изучай хуки внимательно! Они не так просты, как кажется'
    textareaEl.current.focus()
  }

  return (
    <>
      <button onClick={handleClick}>Получить сообщение.</button>
      <label htmlFor='message'>
        После нажатия кнопки в поле для ввода текста появится сообщение.
      </label>
      <textarea ref={textareaEl} id='message' />
    </>
  )
}
```

```jsx
function TextInputWithFocusButton() {
  const inputEl = useRef( initialValue: null);
  const onButtonClick = () => {
    // `current` указывает на смонтированный элемент `input`
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Установить фокус на поле ввода</button>
    </>
  );
}

```

useReducer()

Альтернатива для useState. Принимает редюсер типа (state, action) => newState и возвращает текущее состояние в паре с методом dispatch. (Если вы знакомы с Redux, вы уже знаете, как это работает.)

Хук useReducer обычно предпочтительнее useState, когда у нас сложная логика состояния, которая включает в себя несколько значений, или когда следующее состояние зависит от предыдущего.

```
181   const [state, dispatch] = useReducer(reducer, initialState, initFn)
182  // dispatch({ type: 'actionType', payload: 'actionPayload' }) используется для
183  // отправки операции в редуктор (для обновления состояния)
184  // initFn - функция для "ленивой" установки начального состояния
```

useReducer()

Ленивая инициализация или что скрывается за третьим аргументом useReducer?

```
const [state, dispatch] = useReducer(reducer, initialCount, init);
```

Мы можем передать функцию init в качестве третьего аргумента. Начальное состояние будет установлено равным результату вызова init(initialArg). Это позволяет извлечь логику для расчёта начального состояния за пределы редюсера.

```
function init(initialCount) {
  return {count: initialCount};
}
```

useReducer()

Пример использования хука useReducer()

```
188  // начальное состояние
189  const initialState = { width: 30 }
190
191  // редуктор
192  const reducer = (state, action) => {
193    switch (action) {
194      case 'plus':
195        return { width: Math.min(state.width + 30, 600) }
196      case 'minus':
197        return { width: Math.max(state.width - 30, 30) }
198      default:
199        throw new Error('Что происходит?')
200    }
201  }
202
203  const BasicReducer = () => {
204    const [state, dispatch] = useReducer(reducer, initialState)
205
206    const [color, setColor] = useState('#f0f0f0')
207
208    useEffect(() => {
209      const randomColor = `#${((Math.random() * 0xffff) << 0).toString(16)}`
210      setColor(randomColor)
211    }, [state])
212
213    return (
214      <>
215        <div
216          style={{
217            margin: '0 auto',
218            background: color,
219            height: '100px',
220            width: state.width
221          }}
222        ></div>
223        <button onClick={() => dispatch('plus')}>
224          Увеличить ширину контейнера.
225        </button>
226        <button onClick={() => dispatch('minus')}>
227          Уменьшить ширину контейнера.
228        </button>
229      </>
230    )
231  }
```

useImperativeHandle()

useImperativeHandle настраивает значение экземпляра, которое предоставляется родительским компонентам при использовании ref. Как всегда, в большинстве случаев следует избегать императивного кода, использующего ссылки. useImperativeHandle должен использоваться с forwardRef.

В этом примере родительский компонент, который отображает <FancyInput ref={inputRef} />, сможет вызывать inputRef.current.focus().

```
14  function FancyInput(props, ref) {
15    const inputRef = useRef();
16    useImperativeHandle(ref,  init: () => ({
17      focus: () => {
18        inputRef.current.focus();
19      }
20    }));
21    return <input ref={inputRef} />;
22  }
23  FancyInput = forwardRef(FancyInput);
```

useTransition()

Специальный хук, который откладывает обновление компонента до полной готовности и убирает промежуточное состояние загрузки.

Вызываем хук useTransition и указываем тайм-аут в миллисекундах. Если данные не придут за указанное время, то мы всё равно покажем loader. Но если получим их быстрее, произойдёт моментальный переход.

```
function App() {
    const [resource, setResource] = useState(initialResource)
    const [startTransition, isPending] = useTransition({ timeoutMs: 2000 })
    return <>
        <Button text='Далее' disabled={isPending} onClick={() => {
            startTransition(() => {
                setResource(fetchData())
            })
        }}>
        <Page resource={resource} />
    </>
}
```

useDeferredValue()

Этот хук принимает значение, для которого мы хотим получить отложенную версию, и задержку в миллисекундах.

```
250  function Page({ resource }) {
251      const deferredResource = useDeferredValue(resource, { timeoutMs: 1000 })
252      const isDeferred = resource !== deferredResource;
253      return (
254          <Suspense fallback={<h1>Loading user...</h1>}>
255              <User resource={resource} />
256              <Suspense fallback={<h1>Loading posts...</h1>}>
257                  <Posts resource={deferredResource} isDeferred={isDeferred}/>
258              </Suspense>
259          </Suspense>
260      )
261  }
```

useId()

Это новый хук для генерации уникальных идентификаторов как на клиенте, так и на сервере, избегая при этом несоответствия при гидратации. В первую очередь он полезен для библиотек компонентов, интегрирующихся с accessibility API, которые требуют уникальных идентификаторов.

```
264  const id = useId();
265
266  function Checkbox() {
267      const id = useId();
268      return (
269        <>
270          <label htmlFor={id}>Do you like React?</label>
271          <input id={id} type="checkbox" name="react"/>
272        </>
273      );
274    };
```

Custom hooks (пользовательские хуки)

Пользовательский хук — это JavaScript-функция, имя которой начинается с «use», и которая может вызывать другие хуки.

В отличие от React компонента, пользовательский хук не обязательно должен иметь конкретную сигнатуру. Мы можем решить, что он принимает в качестве аргументов, и должен ли он что-либо возвращать.

Так же как и в компонентах, убедитесь, что вы не используете другие хуки внутри условных операторов и вызываете их на верхнем уровне вашего хука.

```
277  import { useLayoutEffect } from 'react'
278
279  export function useTheme(theme) {
280    useLayoutEffect(() => {
281      for (const [prop, val] in theme) {
282        document.documentElement.style.setProperty(`--${prop}`, val)
283      }
284    }, [theme])
285  }
```

Custom hooks (пользовательские хуки)

Еще один пример пользовательского хука - useFriendStatus

Он принимает в качестве аргумента friendID и возвращает статус друга в сети.

```
import { useState, useEffect } from 'react';
import {ChatAPI} from "./Api";

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState( initialState: null);

  useEffect( effect: () => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

BrowserRouter

Это Router, который использует API истории HTML5 (pushState, replaceState и событие popstate) для синхронизации вашего пользовательского интерфейса с URL-адресом.

BrowserRouter определяет набор маршрутов и, когда к приложению, приходит запрос, то он выполняет сопоставление запроса с маршрутами. И если какой-то маршрут совпадает с URL запроса, то этот маршрут выбирается для обработки запроса. Это основа нашего роутинга им следует оборачивать наше приложение.

```
<BrowserRouter
  basename={optionalString}
  forceRefresh={optionalBool}
  getUserConfirmation={optionalFunc}
  keyLength={optionalNumber}
>
  <App />
</BrowserRouter>
```

Route

Каждый маршрут представляет объект Route. Данный компонент имеет такие основные пропсы:

• path – шаблон адреса, с которым будет сопоставляться запрошенный адрес URL.

• component – компонент который будет отрисовываться при совпадении path и запрошенного адреса URL.

• exact - допускает только точное совпадение маршрута со строкой запроса. Например, строка запроса может представлять путь "/" или путь "/about" или путь "/contact/dsdf", и все эти пути будут соответствовать маршруту с шаблоном адреса "/". Но пропс exact позволяет рассматривать точное совпадение, то есть когда шаблону "/" соответствует только строка запроса "/".

```
return (
  <BrowserRouter>
    <div className="App">
      <Header isLogin={isLogin} />
      <div>
        <button onClick={toggleLogin}>
          {isLogin ? "Log out" : "Log in"}
        </button>
      </div>
      <Switch>
        <Route
          path="/"
          component={Home}
          exact
        />
        <Route path="/about" component={About} />
        <Route
          path="/profile"
          component={Profile}
        >
          <Profile isLogin={isLogin} />
        </Route>
        <Route path="/post/:id" component={Post} />
        <Route component={NotFound} />
      </Switch>
    </div>
  </BrowserRouter>
);
}
```

Switch

Использование этой обертки для наших компонентов Route дает возможность при совпадении запрошенного URL и path выводить компонент только первого совпавшего Route.

Таким образом, допустим, если у нас есть 3 компонента Route с path “/”, “about” и “/profile”. То при запросе адреса “/” без компонента Switch наше приложение выдало бы все 3 компонента Home, About и Profile. С компонентом Switch в таком случае на UI вывелся только компонент Home
Но, будте внимательны, если у Route с path=“/” убрать пропс exact, то при переходе на другие страницы “/about”, ”/profile” у нас всегда будет выводиться Home.
Решение – добавить exact или переместить Route с корневым path в конец списка роутов.

```
return (
  <BrowserRouter>
    <div className="App">
      <Header isLogin={isLogin} />
      <div>
        <button onClick={toggleLogin}>
          {isLogin ? "Log out" : "Log in"}
        </button>
      </div>
      <Switch>
        <Route
          path="/"
          component={Home}
          exact
        />
        <Route path="/about" component={About} />
        <Route
          path="/profile"
          component={Profile}
        >
          <Profile isLogin={isLogin} />
        </Route>
        <Route path="/post/:id" component={Post} />
      </Switch>
    </div>
  </BrowserRouter>
```

Link

Link предоставляет декларативную доступную навигацию по нашему приложению.

Использование компонента Link для навигации по нашему приложению предотвращает обновление вкладки браузера при переходе между страницами нашего приложения.

```jsx
const Header = ({ isLogin }) => {

  return (
    <>
      <h1>React Router Tutorial</h1>
      <ul className="nav">
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        {isLogin && (
          <li>
            <Link to="/profile">Profile</Link>
          </li>
        )}
      </ul>
    </>
  );
}
```

Link

Основной пропс этого компонента – to. Он может принимать как строку, так и объект.

```
<Link to="/courses?sort=name" />
```

С помощью объекта мы можем в поле state передать любые данные компоненту который будет отрисован по нашему роуту. Данный state будет доступен в пропсе location.

```
<Link
  to={{
    pathname: "/courses",
    search: "?sort=name",
    hash: "#the-hash",
    state: { fromDashboard: true }
  }}
/>
```

NavLink

Компонент реализует тот же функционал что и Link за одним отличием, с помощью NavLink мы можем стилизовать нашу ссылку когда она соответствует текущему URL-адресу.

```
const Header = ({ isLogin }) => {

  return (
    <>
      <h1>React Router Tutorial</h1>
      <ul className="nav">
        <li>
          <NavLink
            to="/"
            activeClassName="link--active"
            exact
          >
            Home
          </NavLink>
        </li>
        <li>
          <NavLink
            to="/about"
            activeClassName="link--active"
          >
            About
          </NavLink>
        </li>
        {isLogin && (
          <li>
            <NavLink
              to="/profile"
              activeClassName="link--active"
            >
              Profile
            </NavLink>
          </li>
        )}
      </ul>
    </>
  );
}
```

Redirect

С помощью Redirect можно перенаправить пользователя на другой URL. Так же при перенаправлянии на другой URL произойдет переопределение текущего местоположения (URL) в стеке истории браузера.

```
return (
  <BrowserRouter>
    <div className="App">
      <Header isLogin={isLogin} />
      <div>
        <button onClick={toggleLogin}>
          {isLogin ? "Log out" : "Log in"}
        </button>
      </div>
      <Switch>
        <Route
          path="/"
          component={Home}
          exact
        />
        <Route path="/about" component={About} />
        <Route
          path="/profile"
          component={Profile}
        >
          {isLogin ? <Profile /> : <Redirect to="/" />}
          <Profile isLogin={isLogin} />
        </Route>
        <Route path="/post/:id" component={Post} />
        <Route component={NotFound} />
      </Switch>
    </div>
  </BrowserRouter>
);
}
```

React Router DOM Hooks

В библиотеке React Router DOM доступны такие хуки:

useHistory;

useLocation;

useParams;

useLocation

useLocation возвращает объект location, представляющий текущий URL. При изменении URL-адреса данный хук возвращает новый location

```
{
  key: 'ac3df4', // not with HashHistory!
  pathname: '/somewhere',
  search: '?some=search-string',
  hash: '#howdy',
  state: {
    [userDefined]: true
  }
}
```

React Router DOM Hooks

useHistory

Возвращает объект history, который имеет такие поля как:

• length (число) – количество записей в стеке истории;

• action (строка) – текущее действие (PUSH, REPLACE, POP);

• объект location;

• push (функция) (path, [state]) – помещает новую запись в стеке истории;

• replace (функция) (path, [state]) – заменяет текущую запись в стеке истории;

• go (функция) (n) – перемещает указатель в стеке истории по записям;

• goBack (функция) – эквивалент go(-1);

• goForward (функция) – эквивалент go(1);

React Router DOM Hooks

useParams

Хук useParams возвращает объект с параметрами в виде ключ – значение.

```
function BlogPost() {
  let { slug } = useParams();
  return <div>Now showing post {slug}</div>;
}

ReactDOM.render(
  <Router>
    <Switch>
      <Route exact path="/">
        <HomePage />
      </Route>
      <Route path="/blog/:slug">
        <BlogPost />
      </Route>
    </Switch>
  </Router>,
  node
);
```

React Router DOM Hooks

useParams

Хук useParams возвращает объект с параметрами в виде ключ – значение.

```
function BlogPost() {
  let { slug } = useParams();
  return <div>Now showing post {slug}</div>;
}

ReactDOM.render(
  <Router>
    <Switch>
      <Route exact path="/">
        <HomePage />
      </Route>
      <Route path="/blog/:slug">
        <BlogPost />
      </Route>
    </Switch>
  </Router>,
  node
);
```