React Reconciliation
Хуки

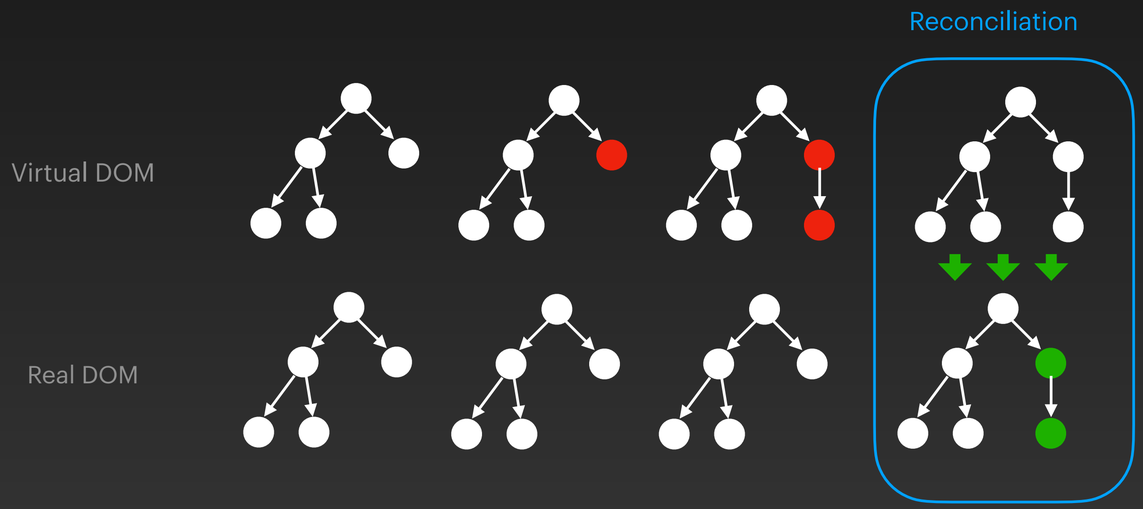
Кирилл Талецкий

TeachMeSkills
28 сентября 2023

Reconciliation

Reconciliation

- Применение изменений из vDOM к документу



About 216.000 results (0,39 seconds)

English – detected

Russian

reconciliation

ˌrekənˌsilēˈāSH(ə)n

примирение

primireniye

Translations of reconciliation

noun

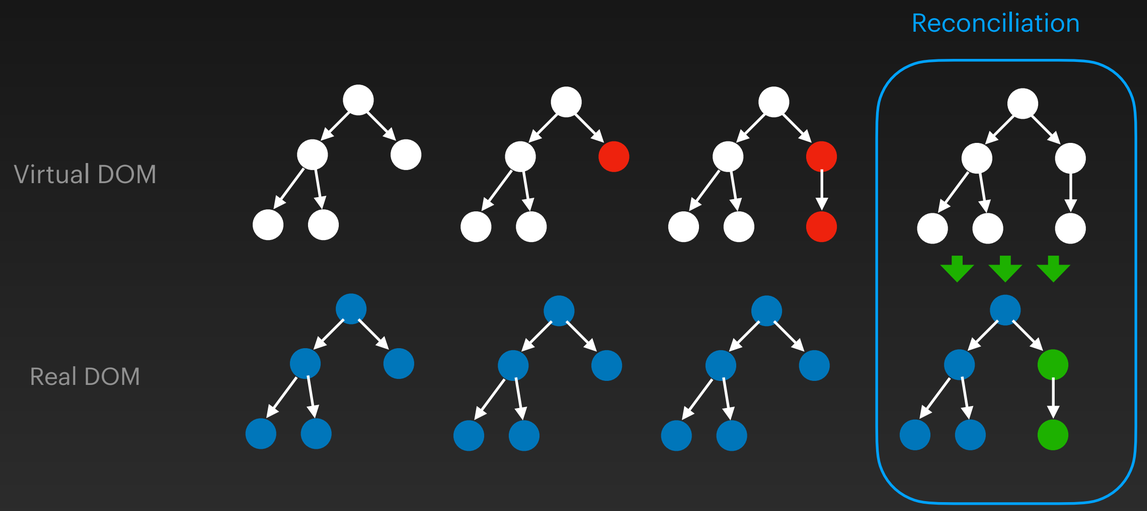
примирение
reconciliation, conciliation, accommodation, propitiation, reunion, reconcilement

согласование
matching, agreement, coordination, negotiation, reconciliation, adjustment

улаживание
reconciliation, reconcilement

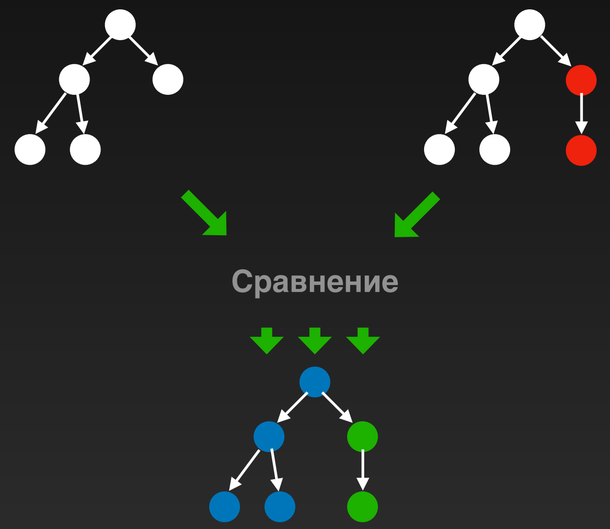
Open in Google Translate • Feedback

Reconciliation



Предыдущий Virtual DOM
aka
Current Tree

Новый Virtual DOM
aka
Work in Progress Tree



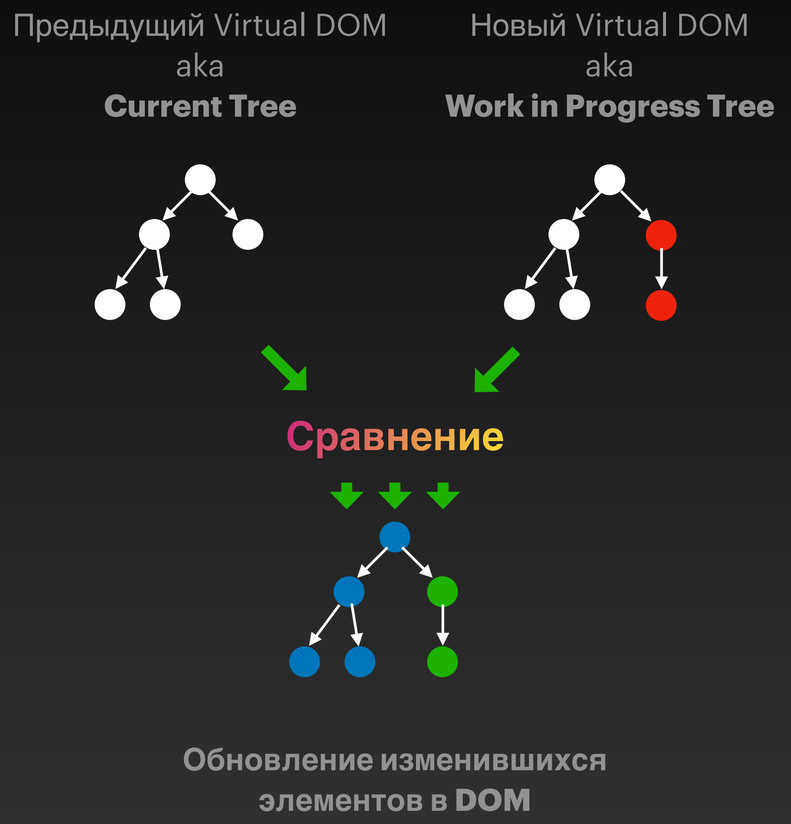
Сравнение

Обновление изменившихся элементов в DOM

Сравнение деревьев

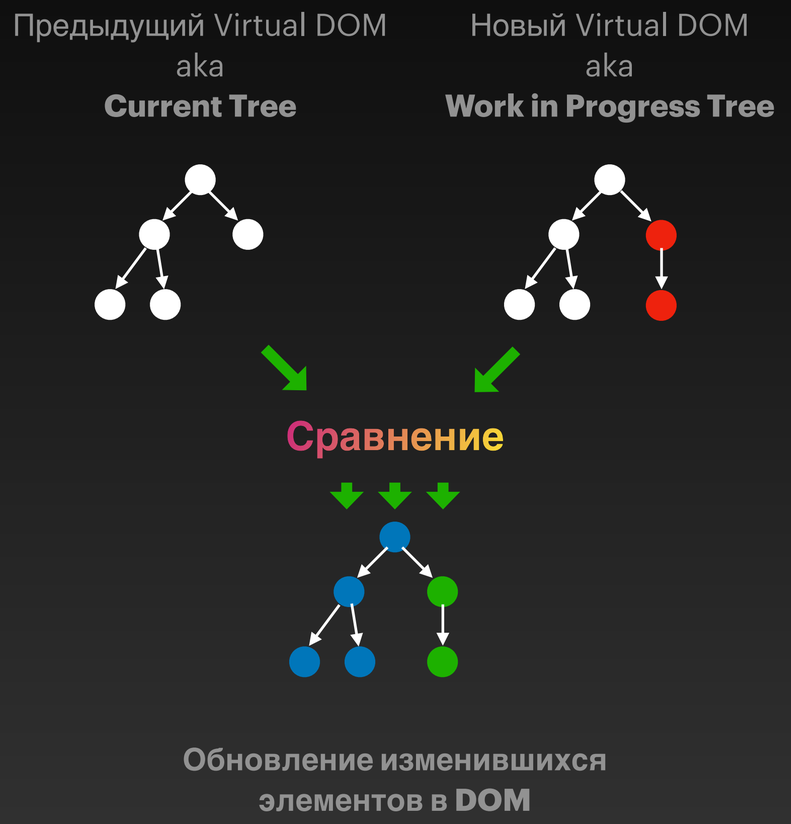
- Любой современный алгоритм сравнения двух произвольных деревьев имеет сложность O(n^3)

- То есть, если в дереве количество элементов со временем изменится с 10 до 10 тысяч, то производительность просядет в **миллиард** раз



Сравнение деревьев

- Для решения этой проблемы, реакт использует свой, эвристический алгоритм сравнения деревьев со сложностью **O(n)**
- Для этого используется два правила
  - Если у элемента поменялся тип, то считается, что всё дерево его потомков полностью поменялось
  - При рендере динамических наборов детских элементов нужно помечать каждый из них специальным пропсом key.



Reconciliation: следствия

Динамическое изменение типа элемента

- Эвристики, которые использует реакт для сравнения деревьев
  - **Если у элемента поменялся тип, то считается, что всё дерево его потомков полностью поменялось**
  - При рендере динамических наборов детских элементов нужно помечать каждый из них специальным пропсом key.

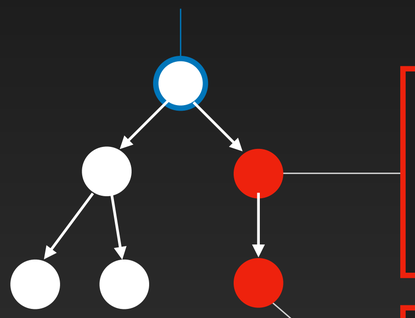
Динамическое изменение типа элемента

```tsx
interface IncrementButtonProps {
  count: number;
  increment: () => void;
}

const IncrementButton: FC<IncrementButtonProps> = ({ count, increment }) => {
  return (
    <button onClick={increment}>
      <span>{`Current count is: ${count}`}</span>
      <br />
      <span>{`Click to increment`}</span>
    </button>
  );
};

interface ResetButtonProps {
  reset: () => void;
}

const ResetButton: FC<ResetButtonProps> = ({ reset }) => {
  const [clickCount, setClickCount] = useState(0);

  const handleClick = () => {
    reset();
    setClickCount((count) => ++count);
  };

  return (
    <button onClick={handleClick}>
      {clickCount > 5 ? "Слишком много ресетов, астанавитесь!" : "Ресет"}
    </button>
  );
};
```

```tsx
export const Counter: FC = () => {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount((c) => c + 1);
  };

  const reset = () => {
    setCount(0);
  };

  const Component = count >= 3 ? "span" : "div";

  return (
    <Component>
      <IncrementButton count={count} increment={increment} />
      <br />
      <br />
      <ResetButton reset={reset} />
    </Component>
  );
};
```

Динамическое изменение типа элемента

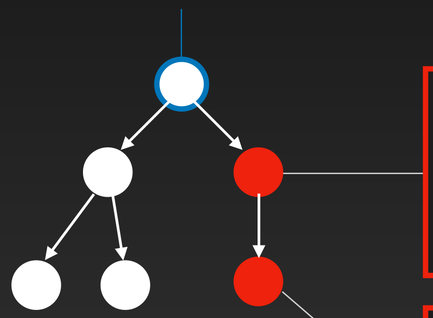
```
{
  name: "Counter",
  state: 0,
  children: [
    div
  ]
}
```



```
{
  name: “div”
}
```

```
{
  name: “ResetButton”,
  state: 42,
  children: [
    button
  ]
}
```

```
{
  name: "Counter",
  state: 1,
  children: [
    span
  ]
}
```



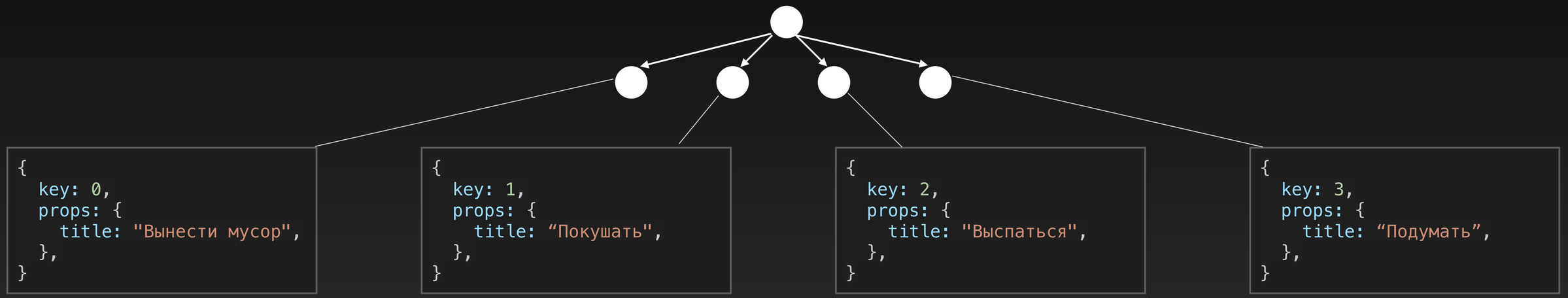
```
{
  name: “span”
}
```

```
{
  name: “ResetButton”,
  state: 0,
  children: [
    button
  ]
}
```

Рендер списков

- Эвристики, которые использует реакт для сравнения деревьев
  - Если у элемента поменялся тип, то считается, что всё дерево его потомков полностью поменялось
  - **При рендере динамических наборов детских элементов нужно помечать каждый из них специальным пропсом key**

Как ключи-индексы ломают реакт



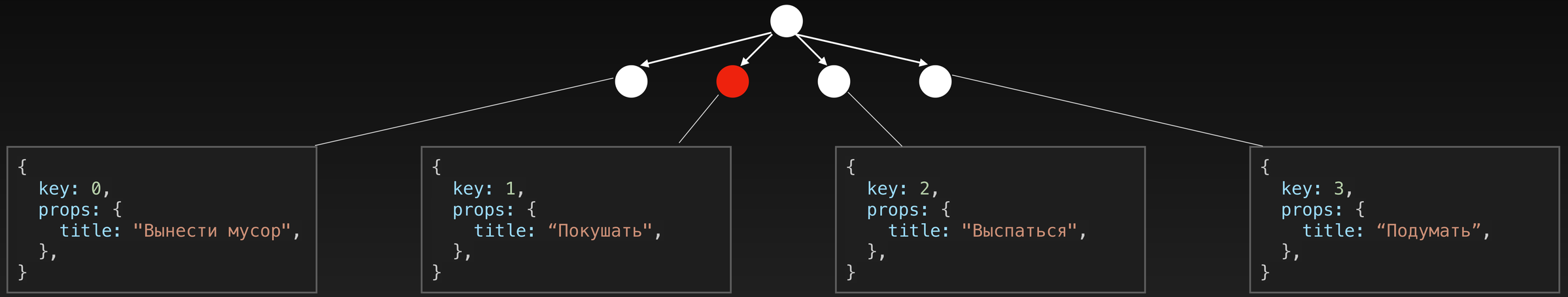
```
{
  key: 0,
  props: {
    title: "Вынести мусор",
  },
}
```

```
{
  key: 1,
  props: {
    title: “Покушать",
  },
}
```

```
{
  key: 2,
  props: {
    title: "Выспаться",
  },
}
```

```
{
  key: 3,
  props: {
    title: “Подумать”,
  },
}
```

Как ключи-индексы ломают реакт



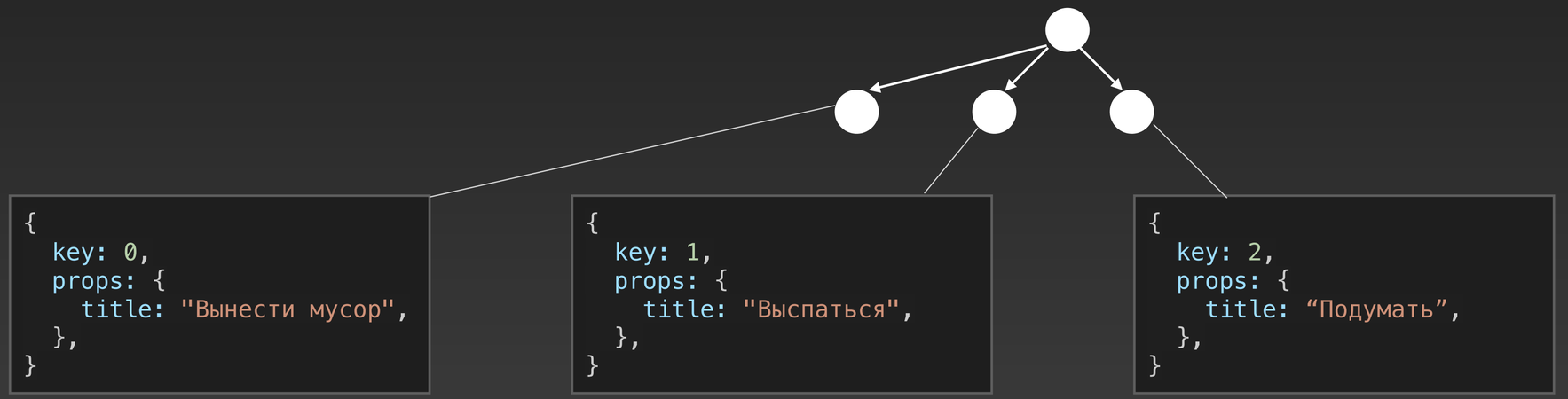
```
{
  key: 0,
  props: {
    title: "Вынести мусор",
  },
}
```

```
{
  key: 1,
  props: {
    title: “Покушать",
  },
}
```

```
{
  key: 2,
  props: {
    title: "Выспаться",
  },
}
```

```
{
  key: 3,
  props: {
    title: “Подумать”,
  },
}
```

Пользователь удаляет
задачу про “покушать”



```
{
  key: 0,
  props: {
    title: "Вынести мусор",
  },
}
```

```
{
  key: 1,
  props: {
    title: "Выспаться",
  },
}
```

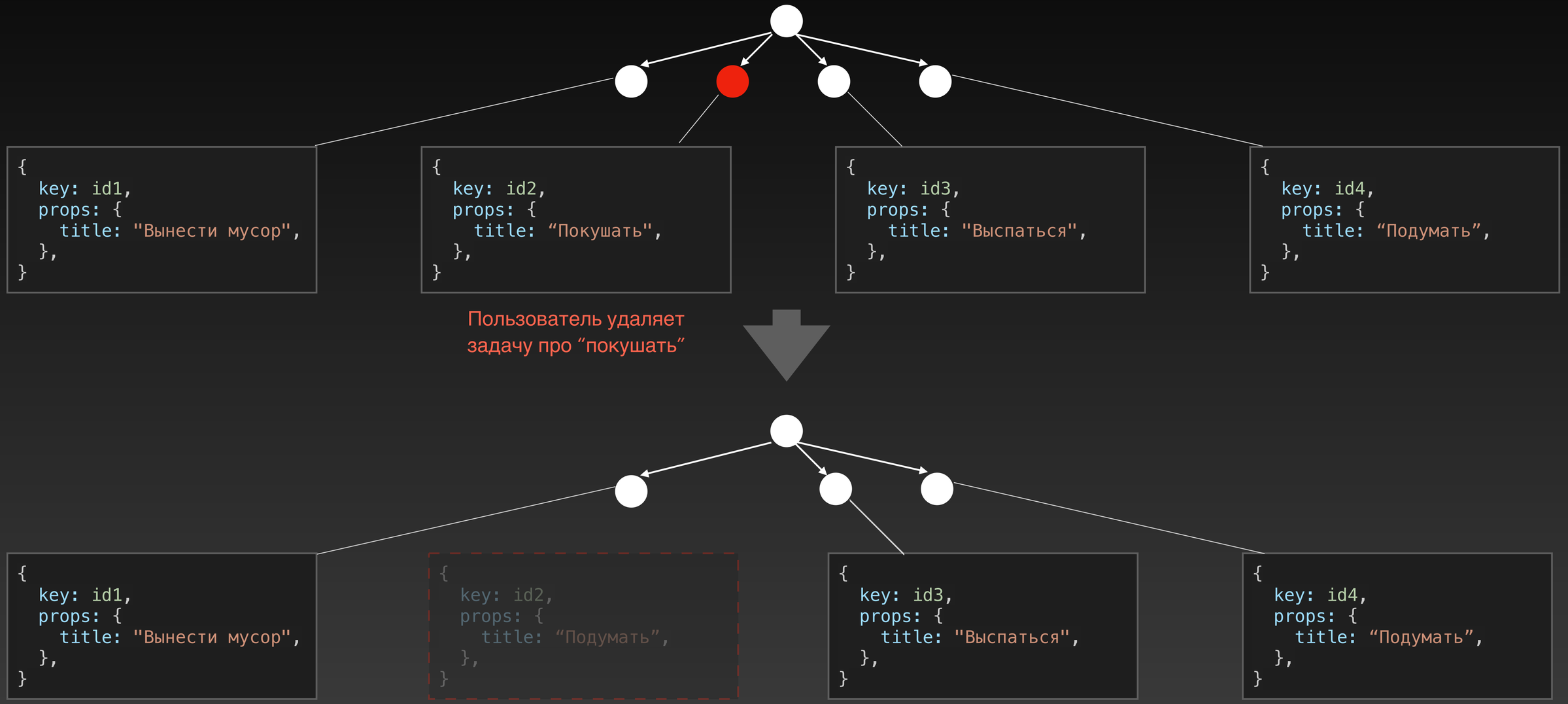
```
{
  key: 2,
  props: {
    title: “Подумать”,
  },
}
```

Мы удалили элемент с key = 1

Но React увидел, что “пропал” элемент
с key = 3 и отмонтировал именно его

```
{
  key: 3,
  props: {
    title: “Подумать”,
  },
}
```

Правильная работа



Хуки в React

Хуки

- Обычные JS функции
- Встроенные в React хуки позволяют добавлять в компоненты функционал
  - Хранить состояние
  - Выполнять код на различные этапы жизненного цикла компонент
  - и т.д.

Правила хуков

- Хуки можно использовать только внутри React компонент
- Хуки всегда должны идти в одинаковом порядке и их количество должно быть всегда одинаковым.

- Следовательно, хуки нельзя использовать:
  - В условных выражениях
  - В циклах
  - В вложенных функциях

- Конвенция — хуки принято называть начиная с `use...`, это позволяет легко отличать их от обычных функций в проекте

Состояние

useState()
useReducer()*

useState()

- Позволяет хранить состояние внутри компоненты
- Состояние реактивно — при его изменении для компоненты будет запущен ре-рендер

```
function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Керя');
  const [todos, setTodos] = useState(() => createTodos());
  // …
  <button onClick={() => setAge(18)}>Повзрослеть</button>
}
```

setState()

- Колбэк, возвращаемый вторым значением из useState()
- Обновляет состояние, вызывает перерисовку компоненты

```
const [age, setAge] = useState(10);

// ...
<button onClick={() => setAge(18)}>Повзрослеть</button>
```

- Реакт оптимизирует вызовы useState(), объединяя вызовы идущие подряд в один

setState()

- Ещё один вариант:

```
setState((prevValue) => {return nextValue})
```

```
const [age, setAge] = useState(18);

// ...
<button onClick={() => setAge((prevAge) => prevAge + 5)}>
  Повзрослеть 5 лет
</button>
```

- Всегда используйте этот вариант, если новое значение должно зависеть от предыдущего

Правило useState()

- Состояние *иммутабельно* — его никогда нельзя менять напрямую

```
let [age, setAge] = useState(10);

// ...
<button
  onClick={() => {
    age = 21;
  }}
>
  Повзрослеть
</button>;
```

```
const [items, setItems] =
  useState(["lol", "kek"]);

// ...
<button
  onClick={() =>
    setItems((items) => {
      items.splice(0, 1);
    })
  }
>
  Обрезать
</button>;
```

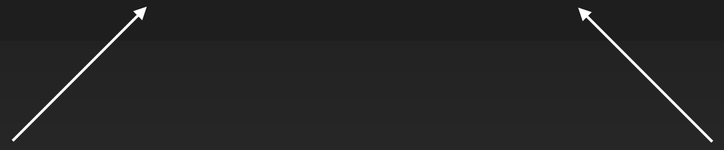
Выходим за рамки рендер цикла

useEffect()
useLayoutEffect()*

useEffect()

- Принимает колбэк и массив зависимостей

```
useEffect(doSomething, [dep1, dep2, ...])
```



Колбэк, который будет срабатывать при перерендерах при изменении зависимостей

Всегда будет запущен один раз на первый рендер

Массив зависимостей. Элементы сравниваются через оператор `===`.

useEffect()

- Колбэк может возвращать другой колбэк — cleanup
- React будет его запускать на каждое изменение зависимостей, со старыми значениями, перед тем как будет запущен новый колбэк

```
function cleanup() {/** unsubscribe */}

function doSomething() {
  return cleanup;
}

useEffect(doSomething, [dep1, dep2, ...])
```

1. Mount
2. doSomething() - новые пропсы и состояние

1. Re-render
2. Зависимости изменились
3. cleanup() - старые пропсы и состояние
4. doSomething() - новые пропсы и состояние

1. Unmount
2. cleanup() - старые пропсы и состояние

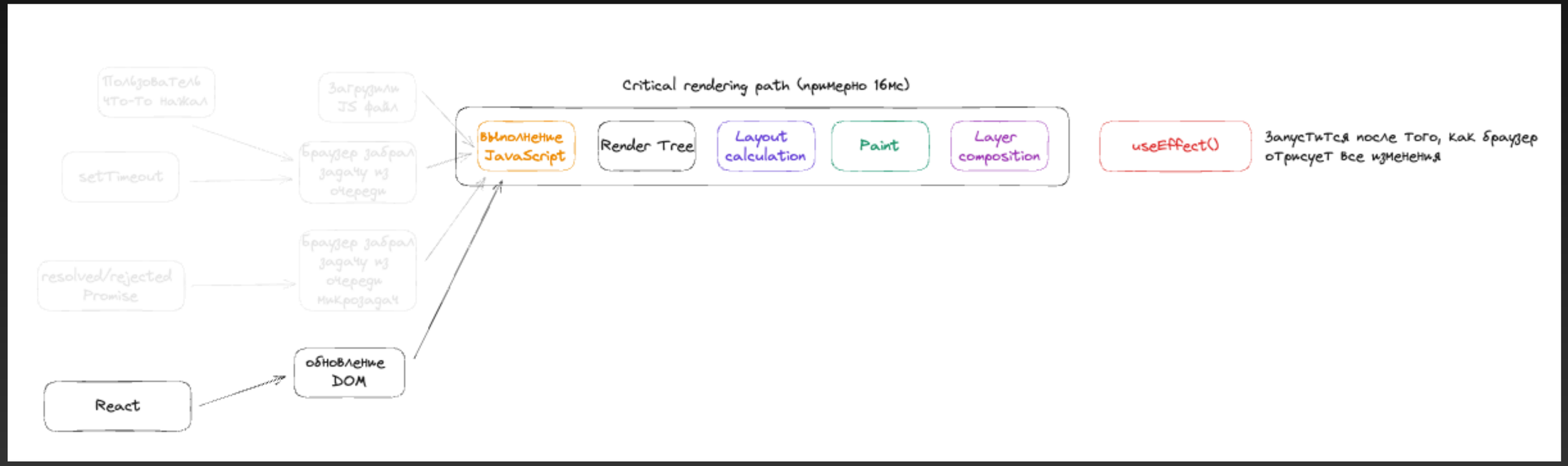
useEffect()

- Позволяет нам запускать код, имитируя componentDidMount и componentWillUnmount

```
useEffect(() => {
  console.log('Компонента примонтировалась');

  return () => {
    console.log(‘Компонента сейчас отмонтируется!’);
  }
}, [])
```

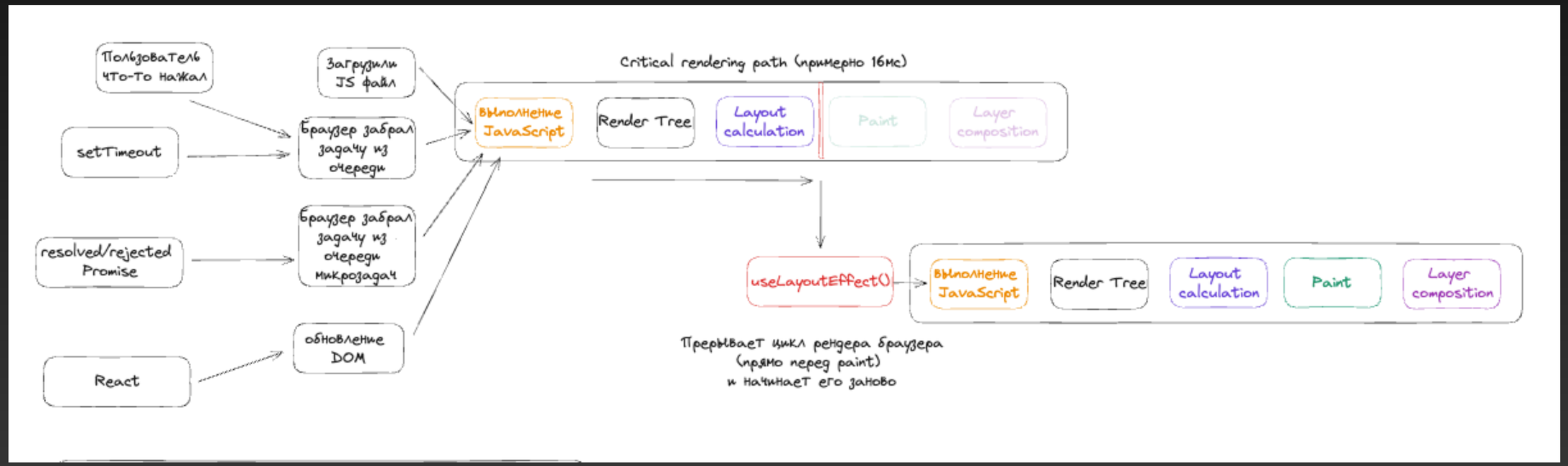
useEffect

Когда срабатывает



useLayoutEffect

- Работает так же, как useEffect, но срабатывает прямо перед paint фазой



Окно в императивный стиль

useRef()
useImperativeHandle()*

useRef()

- Принимает начальное значение
- Возвращает объект, где в поле `current` находится текущее значение

```
const inputRef = useRef(null)
```

Проп ref

- Позволяет “подключиться” напрямую к любому DOM элементу

```
const MyComponent: FC = () => {
  const inputRef = useRef<HTMLInputElement>(null);

  function handleClick() {
    inputRef.current?.focus();
  }

  return <div>
    <input ref={inputRef} />
    <button onClick={handleClick}>
      Focus the input
    </button>
  </div>
}
```

Домашнее задание

Динамическое изменение ключа

- Вы уже знаете, что Реакт при сверке vDOM’ов опирается на ключи, чтобы узнавать элементы
- Опишите, как будет происходить сверка DOM’ов и какие этапы жизненного цикла пройдёт, если мы ему динамически поменяем ключ.

- Hard Level: решите задачу не запуская код

```jsx
export const Counter = () => {
  const [count, setCount] = useState(0);

  return (
    <span>
      Current count is {count}{" "}
      <button
        onClick={() => {
          setCount((count) => ++count);
        }}
      >
        Increment
      </button>
    </span>
  );
};

export const Component = () => {
  const [dynamicKey, setDynamicKey] = useState("initial_value");

  return (
    <div>
      <Counter key={dynamicKey} />
      <br />
      {/**
       * Какие этапы жизненного цикла пройдёт Counter, когда
       * пользователь нажмёт на кнопку?
       */}
      <button onClick={() => setDynamicKey(getRandomString())}>✨</button>
    </div>
  );
};

function getRandomString() {
  return `${Math.floor(Math.random() * 1000000)}`;
}
```