РАЗРАБОТКА ВЕБ-САЙТОВ*, REACTJS*, JAVASCRIPT*

Оптимизация производительности в React

ПЕРЕВОД

vtikunov 15 января 2017 в 01:10 15,8k

Оригинал: @OMantere, @lucas-ar…

Продолжение серии переводов раздела "Продвинутые руководства" (Advanced Guides) официальной документации библиотеки React.js.

Оптимизация производительности в React

Внутренне, React использует несколько продвинутых техник, сводящих к минимуму количество дорогостоящих операций DOM, необходимых для обновления пользовательского интерфейса. Для большинства приложений, использующих React, быстродействие получаемого интерфейса достаточно без дополнительных действий для оптимизации производительности. Тем не менее,

есть несколько способов, с помощью которых вы можете ускорить ваше приложение React.

Использование окончательной (production) сборки

Если вы тестируете производительность или испытываете проблемы с производительностью в вашем приложении React, убедитесь, что вы тестируете минифицированную окончательную (production) сборку:

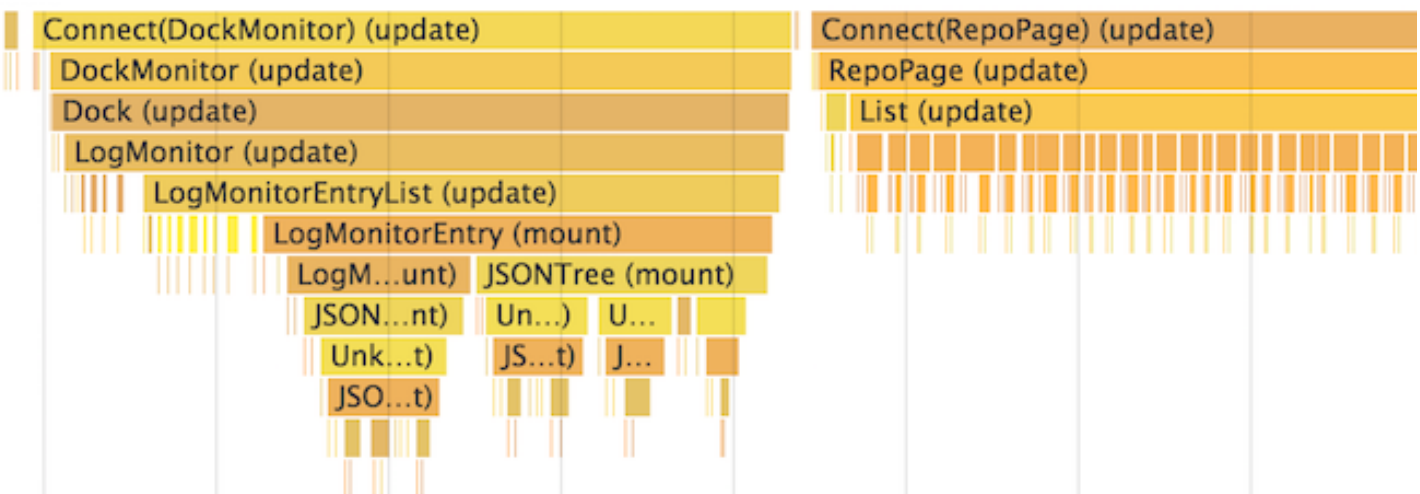
- Для создания сборки приложения React, вам необходимо запустить `npm run build` и следовать дальнейшим инструкциям.
- Для однофайловых сборок мы предлагаем подготовленные для окончательной сборки версии React, имеющими расширение `.min.js`.
- Для сборщика Browserify, вам необходимо запустить сборку с параметром `NODE_ENV=production`.
- Для сборщика Webpack, вам необходимо добавить следующие плагины в конфигурационный файл окончательной сборки (production config):

```
new webpack.DefinePlugin({
  'process.env': {
    NODE_ENV: JSON.stringify('production')
  }
}),
new webpack.optimize.UglifyJsPlugin()
```

Сборка для разработки (development) включает в себя дополнительные предупреждения, которые помогают разрабатывать приложение, но они замедляют работу приложения проводя дополнительные действия.

Профилирование компонентов с помощью Timeline в Chrome

Используя инструменты производительности в поддерживаемых браузерах, в **development** режиме, вы можете визулизировать как компоненты монтируются, обновляются и размонтируются. Например:



Выполните следующие действия в Chrome:

1. Загрузите ваше приложение с параметром `?react_perf` в строке запроса (например: `http://localhost:3000/?react_perf`).

2. Откройте вкладку **Timeline** в Инструментах разработчика Chrome и нажмите **Record** (запись).

3. Выполните действия, которые вы хотите профилировать. Не записывайте действия больше чем 20 секунд — иначе Chrome может зависнуть.

4. Остановите запись.

5. События React будут сгруппированы под ярлыком **User Timing**.

Обратите внимание, что **эти цифры являются относительными, т.к. компоненты будут отображаться быстрее в режиме production**. В конечном итоге, это должно помочь вам представить в каком месте пользовательский интерфейс получает обновления по ошибке, и насколько глубоко и часто происходит обновление вашего пользовательского интерфейса.

В настоящее время эту возможность поддерживают только Chrome, Edge и IE, но, т.к. мы используем стандарт User Timing API, мы ожидаем, что и другие браузеры добавят поддержку этой возможности.

Избегайте излишней перерисовки

React создает и поддерживает в актуальном состоянии свое внутреннее представление отображаемого пользовательского интерфейса. Оно включает в себя элементы React возвращаемые вашими компонентами. Это позволяет React избегать создания уже существующих узлов DOM и доступа к ним сверх необходимости, что может быть медленнее операции с JavaScript

объектами. Иногда эту модель называют "виртуальным DOM", хотя Нативный React (React Native) работает также.

Когда свойства компонента (props) или состояние изменяются, React сравнивает новую возвращаемую версию элемента с предыдущей отображенной в DOM, и в случае если они не эквивалентны — обновляет DOM.

В некоторых случаях, ваш компонент можно ускорить путем переопределения функции жизненного цикла `shouldComponentUpdate`, которая вызывается перед началом процесса переотображения (ререндеринга). Определение функции по умолчанию возвращает `true`, позволяя React совершать обновление:

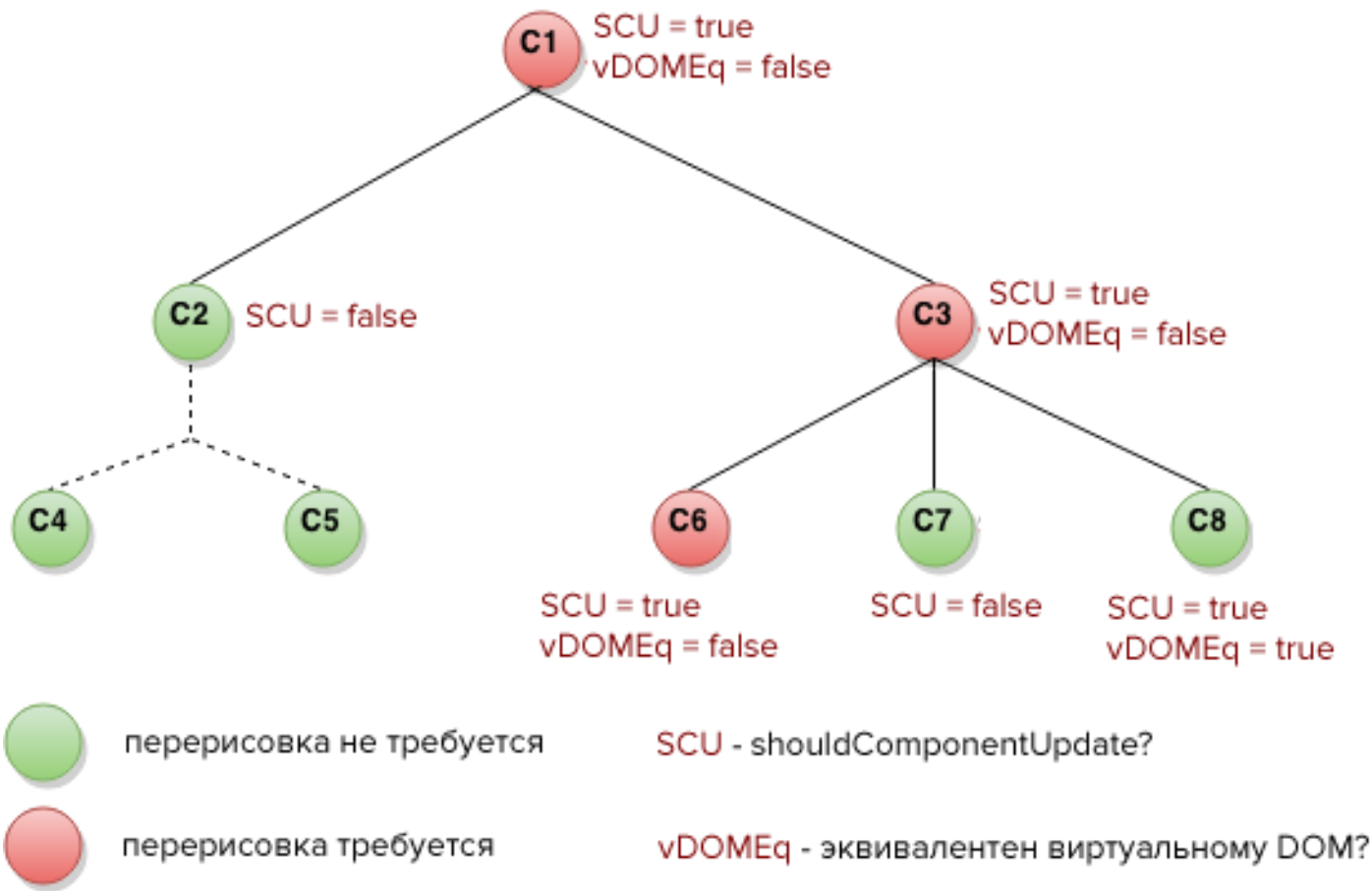
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

Если вам известно, что в некоторых случаях ваш компонент не нуждается в обновлении, вы можете вернуть `false` из функции `shouldComponentUpdate` для пропуска процесса переотображения (ререндеринга), включая вызов метода `render()` в текущем компоненте и ниже по иерархии.

shouldComponentUpdate в действии

На рисунке представлено дерево компонентов. Метка `SCU` отображает результат возвращаемый функцией

`shouldComponentUpdate`, а метка `vDOMEq` — эквивалентен ли элемент React предыдущему представлению. Цвет кругов отображает необходимость перерисовки компонента.



В компоненте C2 функция `shouldComponentUpdate` вернула `false`, в следствие чего все поддерево, начиная от C2 и ниже, React не будет перерисовывать (вызывать функцию render), и для этого даже не пришлось вызывать функцию `shouldComponentUpdate` в компонентах C4 и C5.

Для C1 и C3 — `shouldComponentUpdate` вернула `true`, поэтому React пришлось спуститься ниже и проверить потомков. Для C6 `shouldComponentUpdate` вернула `true`, и поскольку указанные

элементы не экивалентны виртуальному DOM — React пришлось обновить DOM.

Ну и последний интересный вариант — это C8. React пришлось отрендерить компонент, но т.к. новое внутреннее представление React элемента эквивалентно предыдущему, DOM обновлять не потребовалось.

Обратите внимание, что React пришлось сделать изменения в DOM только для C6, которые были неизбежны. В случае с C8 — нас выручило сравнение отрендеренных элементов React, для дерева C2 и компонента C7 не пришлось даже сравнивать элементы — `shouldComponentUpdate` вернула `false` и метод `render` не вызывался.

Примеры

Представим вариант, что наш компонент изменяется только при изменении свойства `props.color` или состояния `state.count`. Мы можем реализовать проверку этих случаев в функции `shouldComponentUpdate`:

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
```

```
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count +
1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

В этом коде, функция `shouldComponentUpdate` проверяет любые изменения в свойстве `props.color` или состоянии `state.count`. Если их значения не изменились, компонент не обновляется. Если ваш компонент более сложный, вы можете использовать подобный шаблон, но производящий "поверхностное сравнение" всех свойств (полей) `props` и состояний `state` для определения необходимости обновления компонента. Этот шаблон используется достаточно часто, поэтому в React есть хелпер для реализации данной логики — просто наследуйте свой компонент от `React.PureComponent`. Следующий код достигает той же цели, что и предыдущий, но он действительно проще:

```
class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
```

```
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count +
1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

В большинстве случаев вы можете использовать `React.PureComponent` вместо того, чтобы писать собственную функцию `shouldComponentUpdate`. Однако, `React.PureComponent` реализует только поверхностное сравнение, и бывают случаи когда свойства (props) или состояния могут быть изменены таким образом, что поверхностного сравнения не достаточно — такая ситуация может происходить с более сложными структурами данных. Например, допустим, что вам необходимо реализовать компонент `ListOfWords` для отображения списка слов разделенных запятыми, с родительским компонентом `WordAdder`, который по нажатию на кнопку добавляет слово в список. Следующий код *не будет работать* корректно:

```
class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
```

```
  }

  handleClick() {
    // Этот раздел написан не верно и приведет к некорректной работе
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        <ListOfWords words={this.state.words} />
      </div>
    );
  }
}
```

Проблема заключается в том, что `PureComponent` делает простое поверхностное сравнение между старыми и новыми значениями массива `this.props.words`. Простое сравнение возвращает `true` если оба сравниваемых значения ссылаются на один и тот же объект (массив). Поскольку указанный код метода `handleClick` компонента `WordAdder` изменяет непосредственно массив `words`, сравнение старого и нового значения `this.props.words` вернет эквивалентность, несмотря на то, что сами слова в массиве изменились. Компонент `ListOfWords` не будет обновлен, имея при этом новые слова, которые необходимо отобразить.

Сила неизменяемых данных

Самый простой способ решения этой проблемы — не изменять (не мутировать) значения, которые вы используете в свойствах (props)

или состоянии. Например, метод `handleClick` может быть переписан с использованием метода `concat`, возвращающего поверхностную копию массива с присоединением новых значений:

```
handleClick() {
  this.setState(prevState => ({
    words: prevState.words.concat(['marklar'])
  }));
}
```

ES6 поддерживает разворачивающий (spread) синтаксис для массивов, который может сделать код еще проще. Если вы используете Create React App для создания своего приложения, этот синтаксис доступен по умолчанию.

```
handleClick() {
  this.setState(prevState => ({
    words: [...prevState.words, 'marklar'],
  }));
};
```

Таким же образом вы можете переписать код для избежания изменений (мутации) объектов. Например, представим, что у нас есть объект `colormap` и мы хотим написать функцию, которая изменяет значение `colormap.right` и устанавливает его равным `'blue'`. Мы могли бы ошибочно написать:

```
function updateColorMap(colormap) {
  colormap.right = 'blue';
}
```

Для реализации этого без мутации первоначального объекта, мы можем использовать метод Object.assign:

```
function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}
```

`updateColorMap` теперь возвращает новый объект, а не изменяет старый. `Object.assign` входит в ES6 и требует включения babel-polyfill.

В ES6 есть возможность разворачивания (spread) свойств объекта и поэтому мы можем реализовать обновление объекта без мутации еще проще:

```
function updateColorMap(colormap) {
  return {...colormap, right: 'blue'};
}
```

Если вы используете Create React App для создания своего приложения, `Object.assign` и разворачивающий синтаксис для объектов доступны по умолчанию.

Использование неизменяемых структур данных

Immutable.js — это еще один путь для решения этой проблемы. Эта библиотека предоставляет неизменяемые, постоянные коллекции, которые работают через структурный обмен:

- *Неизменяемая*: однажды созданная, коллекция не может быть изменена в другой момент времени.
- *Постоянная*: новая коллекция может быть создана из предыдущей коллекции и измененного набора данных. Первоначальная коллекция остается действующей после создания новой коллекции.
- *Структурный обмен*: новая коллекция создается с использованием максимально, насколько это возможно, такой же структуры, как и в первоначальной коллекции, снижая копирование к минимуму для улучшения производительности.

Неизменяемость позволяет сделать отслеживание изменений легким. Изменения всегда приводят к созданию нового объекта и нам остается только проверить изменилась ли ссылка на объект. Например, в нативном JavaScript коде:

```
const x = { foo: "bar" };
const y = x;
y.foo = "baz";
x === y; // true
```

Не смотря на то, что `y` было изменено, `y` ссылается на тот же объект, что и `x` — их сравнение возвращает `true`. Мы можем переписать этот код с использованием immutable.js:

```
const SomeRecord = Immutable.Record({ foo: null });
const x = new SomeRecord({ foo: 'bar'  });
const y = x.set('foo', 'baz');
x === y; // false
```

В данном случае, т.к. при изменении `x` была возвращена ссылка на новый объект, мы можем смело предполагать что `x` изменилась.

Еще две библиотеки, которые могут помочь использовать неизменяемые данные — это seamless-immutable и immutability-helper.

Неизменяемые структуры данных предоставляют вам самый простой способ отслеживания изменений в объектах — а это то, что нам необходимо для использования `shouldComponentUpdate`, которая в большинстве случаев даст вам хороший прирост производительности.

Предыдущие части:

- Неконтролируемые компоненты в React.
- Ref-атрибуты и DOM в React.
- PropTypes — проверка типов в React.
- JSX — подробности.

Первоисточник: React — Advanced Guides — Optimizing Performance

Комментарии (7)

Похожие публикации

Мыслим в стиле React

16

ИЗ ПЕСОЧНИЦЫ

vtikunov • 9 января 2017 в 00:46

React.js State of the art (интервью с Max Stoiber)

25

m1skam • 21 ноября 2016 в 17:15

React.js: собираем с нуля изоморфное / универсальное приложение. Часть 1: собираем стек

76

ИЗ ПЕСОЧНИЦЫ

yury-dymov • 14 сентября 2016 в 10:45

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета)

69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов 20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE 24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке 31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s 49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить

85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

139

marks • вчера в 14:19

Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16