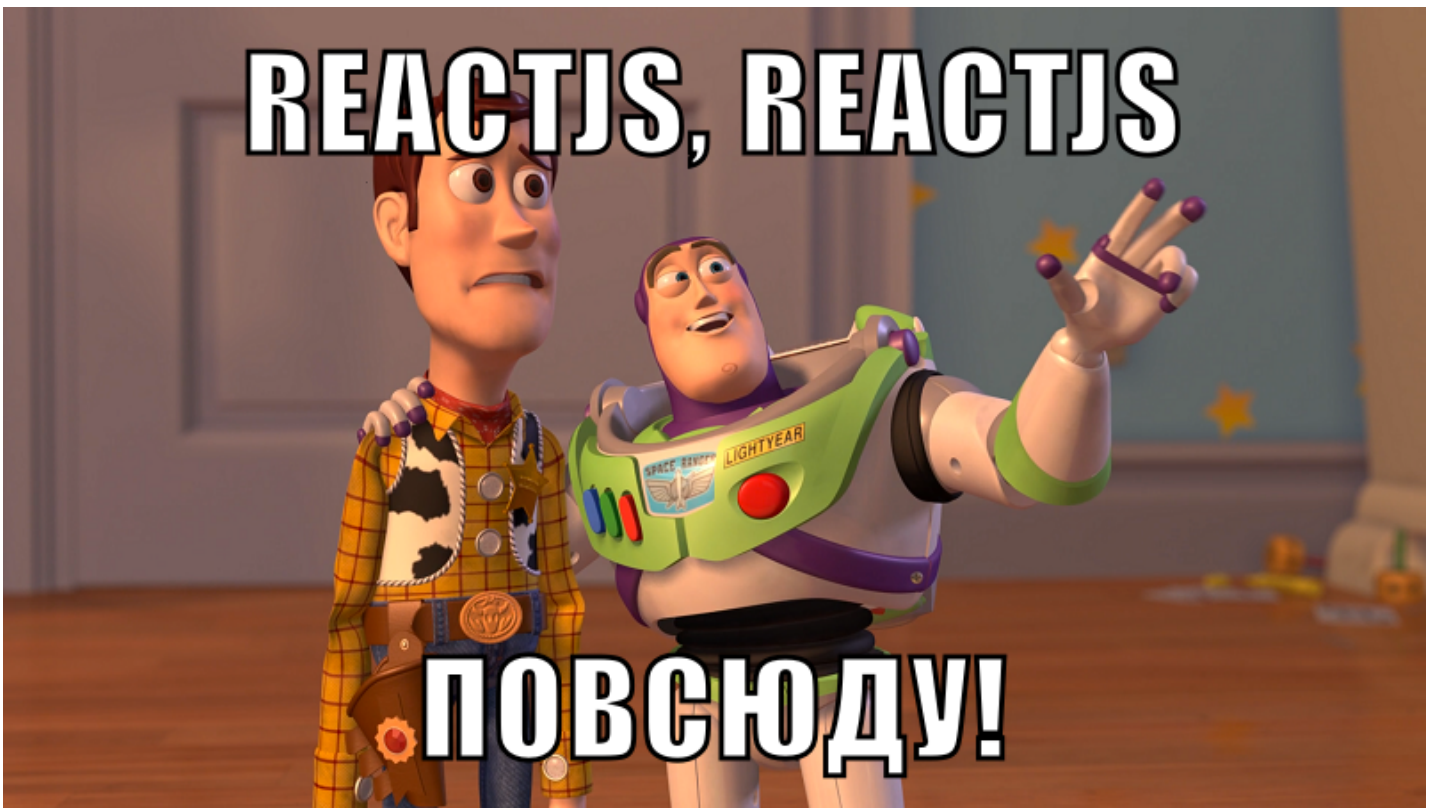
РАЗРАБОТКА ВЕБ-САЙТОВ*, ВЫСОКАЯ ПРОИЗВОДИТЕЛЬНОСТЬ*, REACTJS*, JAVASCRIPT*, БЛОГ КОМПАНИИ ТУТУ.РУ

Как избежать проблем с производительностью при создании React-приложений

megazazik 13 февраля в 11:13 9,4k



О производительности react

React не зря считается очень производительным фреймворком. Он позволяет создавать быстрые динамические страницы с большим количеством элементов.

Но бывают ситуации, когда элементов на странице становится очень много и встроенной производительности react не хватает.

Тогда приходится применять различные приемы для оптимизации.

Страница, написанная на react, состоит из отдельных компонентов. Каждый из них отвечает за внешний вид какой-то определенной части страницы. Этот внешний вид зависит от параметров (свойств), которые были переданы в компонент. При наступлении каких-либо событий (например, при каких-либо действиях пользователя или получении данных по сети) свойства могут меняться. Если свойства какого-либо компонента изменились, он должен быть отрисован заново, чтобы эти изменения отобразились на экране пользователя.

При наступлении каких-либо событий, части страницы могут быть отрисованы заново, даже если свойства компонентов, которые за них отвечают, не были изменены. Если таких частей много, то при любом действии пользователя отрисовка может выполняться значительное время, и могут появляться заметные задержки при взаимодействии с интерфейсом страницы. Основной способ борьбы с такими проблемами производительности — это отмена перерисовки компонентов, если их свойства не были изменены. То есть, при наступлении каких-либо событий, на странице должны быть перерисованы только те элементы, на свойства и внешний вид которых это событие повлияло. Для этого при создании react-компонентов нужно определить метод shouldComponentUpdate или использовать в качестве родительского класса React.PureComponent вместо React.Component.

Метод shouldComponentUpdate должен возвращать false, когда перерисовка не нужна. А в классе React.PureComponent этот

метод уже реализован. В нем происходит проверка всех поступивших свойств, и компонент не будет перерисован, если ни одно его свойство не изменилось.

Пример shouldComponentUpdate:

```
import * as React from 'react';

class Component extends React.Component {
    shouldComponentUpdate(nextProps) {
        return this.props.value !== nextProps.value
    }

    render() {
        return <div>...</div>;
    }
}
```

Пример React.PureComponent:

```
import * as React from 'react';

class Component extends React.PureComponent {
    render() {
        return <div>...</div>;
    }
}
```

Когда оптимизация не работает

Принцип работы shouldCompenentUpdate и React.PureComponent основан на сравнении старых свойств с новыми. Если свойства не изменились, обновление внешнего вида компонента не производится.

Поэтому стоит следить за тем, чтобы свойства изменялись только при необходимости.
Но бывают ситуации, при которых компонент ненамеренно получает новые свойства тогда, когда это не требуется.

Это может происходить при использовании в методе render функции bind, стрелочных функций, литеральных объектов или других конструкций, которые создают новые объекты при каждом вызове.

Когда оптимизация не работает:

```
class ParentComponent extends React.Component {
    _onClick() {
        doSomehing();
    }

    render() {
        return (
            <div>
                <Child1 onClick={this._onClick.bind(this)}/>
                <Child2 onClick={() => this._onClick()}/>
                <Child3 data={{id: this.props.id, value:
this.props.value}}/>
                <Child4 items={this.props.items.map((item) =>
{.....})}/>
            </div>
        );
    }
}
```

Функция bind и стрелочные функции при каждом выполнении возвращают новый экземпляр функции.

Сравнение результатов выполнения bind:

```
this._onClick.bind(this) === this._onClick.bind(this) // false
() => this._onClick() === () => this._onClick() //false
```

Поэтому здесь первые два компонента будут каждый раз получать новое свойство onClick.
Child3 будет получать новый объект в свойстве data, а четвертый — новый массив в items.

Таких конструкций стоит избегать. При их использовании оптимизация производительности, произведенная с помощью shouldComponentUpdate или React.PureComponent, не будет работать из-за получения новых свойств.

В принципе, можно написать такую реализацию shouldComponentUpdate, при которой изменения некоторых свойств будут игнорированы, и обновление происходить не будет. Но такой способ можно использовать не во всех ситуациях. Во-первых, если свойств много, такая реализация может содержать довольно большое количество проверок, а это может быть причиной ошибок и багов. Также этот код сложнее поддерживать, так как при добавлении новых свойств нужно будет дорабатывать метод shouldComponentUpdate. Во-вторых, при таком подходе компонент, в котором реализован shouldComponentUpdate, должен точно знать, где и при каких обстоятельствах он будет использоваться, чтобы точно определить, изменения каких именно свойств можно игнорировать. А в том месте, где он будет использоваться, нужно знать его внутреннюю реализацию, чтобы не передавать лишний раз новые значения в те свойства,

изменение которых будет приводить к ненужной перерисовке. Это сильно ухудшает инкапсуляцию компонентов, что чаще всего нежелательно.

Результаты тестирования производительности

Часто во время написания кода и во время проведения code review возникают сомнения, нужно ли тратить свое время и обращать внимание на подобные проблемы. Чтобы определить влияние, которое передача новых свойств в компоненты оказывает на производительность, была написана небольшая тестовая страница. Она размещена по адресу https://megazazik.github.io/react-perf-test/

Проект позволяет измерить время обновления большого количества маленьких компонентов в ситуациях, когда они каждый раз получают новые свойства, а также когда свойства остаются неизменными. Для тестирования использовались разные компоненты:

- stateful — наследуемые от React.Component
- stateful — наследуемые от React.PureComponent
- stateless — простые функции
- html-элементы (div)

При открытии страницы проекта происходит отображение списка из тысячи элементов. При клике на любой элемент или при наведении мыши выбранный элемент становится активным, и список перерисовывается заново. При этом, в зависимости от

выбранных параметров, элементы, чье состояние не изменилось, получают либо те же самые свойства, либо — новые, из-за использования стрелочных функций.

Время обновления 1000 элементов:

Устройство, Браузер	Оптимизация	Pure, мс	Stateful, мс	Stateless, мс	html, мс
Desktop, Chrome	нет	6,5	6,3	5,8	3,5
Desktop, Chrome	да	3,3	4,9	4,3	2,5
Desktop, Firefox	нет	13,7	11,9	11,5	7,3
Desktop, Firefox	да	5,4	8,5	7,6	4,4
Desktop, Edge	нет	19,6	16,2	13,5	8,6
Desktop, Edge	да	8,1	11,5	8,7	4,7
Mobile, Chrome	нет	31,1	30,2	29,6	19,3
Mobile, Chrome	да	16,7	25,8	21,5	14,8

Конкретные числа могут сильно зависеть от устройства и браузера, на котором открывается страница, а также от состояния памяти и загрузки процессора. Но относительная разница в производительности отрисовки, с оптимизацией и без, должна всегда оставаться примерно на одном уровне.

Можно сравнить полученные результаты с теми, что будут актуальны для вашего устройства.

Оптимальным временем отрисовки можно считать такое время, при котором браузер не будет пропускать кадры при обновления экрана устройства. Большинство современных экранов поддерживают частоту обновления минимум 60 кадров в секунду. Получается, экран обновляется минимум каждые 16,6 миллисекунды. Поэтому, с учётом возможной дополнительной

нагрузки желательно, чтобы обновление всех компонентов страницы происходило не более, чем за 10 миллисекунд.

По результатам тестов видно, что время отрисовки компонентов, унаследованных от PureComponent и html-элементов, существенно уменьшается, когда свойства не изменяются.
При этом в тестах неизменяемость свойств также оказывает влияние на производительность других видов компонентов. Но оно существенно меньше и связано не с отменой перерисовки, а с тем, что полученные ими свойства в итоге передаются в те же div-элементы в разметке, которые уже в свою очередь оптимизированы.

При тестировании использовались самые маленькие компоненты. При увеличении их размера выигрыш в производительности должен только увеличиваться. Это влияние можно примерно оценить, если в тестовом проекте включить опцию «Разбиение списка». В этом режиме весь список разбивается на группы, по 10 элементов в каждой. Потом каждые 10 групп свою очередь объединяются в группы верхнего уровня. В итоге, страница состоит из 10 групп, которые содержат по 10 подгрупп, в каждой из которых 10 элементов.

Такое разбиение больше похоже на то, из чего состоят реальные страницы. Обычно страница содержит несколько больших компонентов, которые содержат вложенные компоненты поменьше.

В таком режиме можно оценить разницу в производительности между наихудшей ситуацией, когда при любом событии перерисовывается каждый элемент страницы, и ситуацией. когда обновляются только те элементы, свойства, которых изменились.

Устройство, Браузер	Оптимизация	Pure, мс	Stateful, мс	Stateless, мс	html, мс
Desktop, Chrome	нет	8,7	8,3	8,0	6,1
Desktop, Chrome	да	0,9	0,9	0,9	0,8
Desktop, Firefox	нет	19,1	16,3	15,4	12,6
Desktop, Firefox	да	1,5	1,5	1,5	1,3
Desktop, Edge	нет	23,1	20,6	18,1	13,0
Desktop, Edge	да	2,9	2,9	2,9	2,8
Mobile, Chrome	нет	37,1	35,0	34,5	24,9
Mobile, Chrome	да	5,2	5,6	5,3	4,7

В данном случае не имеет большого значения, тип тестируемого компонента, так как в группах, на которые разбит список, реализован собственный метод shouldComponentUpdate.

Как избежать создания новых функций в render

На практике чаще всего ненужная генерация новый свойств компонентов происходит из-за использования стрелочных функций или функции bind. Рассмотрим, как можно эту проблему решить. Чаще всего, это сделать достаточно просто. Но иногда требуется потратить некоторое время.

Самый простой способ не создавать новые функции при каждом обновлении — это создать их один раз в момент инициализации экземпляра компонента.

Для этого можно вызвать функцию bind не в методе render, а в конструкторе класса, и сохранить результат в свойстве объекта, для последующего обращения в методе render.

Пример bind в конструкторе:

```
class ParentComponent extends React.Component {
    constructor(props) {
        super(props);

        this._onClick = this._onClick.bind(this);
    }

    _onClick() {
        doSomehing();
    }

    render() {
        return <ChildComponent onClick={this._onClick}/>;
    }
}
```

Или можно использовать стрелочную функцию и также сохранить результат в свойстве объекта.

Пример со стрелочной функцией:

```
class ParentComponent extends React.Component {
    _onClick = () => {
        doSomehing();
    }

    render() {
        return <ChildComponent onClick={this._onClick}/>;
    }
}
```

Предыдущий способ простой и быстрый в реализации. Его и стоит использовать в большинстве случаев.

Но бывают ситуации, когда функция обратного вызова передается списку дочерних компонентов, например, в цикле, и при ее выполнении должно быть известно, в каком именно из элементов произошел вызов. Такая функция может принимать в качестве аргумента, например, индекс элемента массива.

Пример со списком:

```
class ParentComponent extends React.Component {
    _onClick = (index) => {
        doSomehing(index);
    }

    render() {
        return (
            <div>
                {this.props.items.map((item, index) => (
                    <ChildComponent
                        onClick={() => this._onClick(index)}
                        item={item}
                    />
                ))}
            </div>
        );
    }
}

class ChildComponent extends React.PureComponent {
    render() {
        return <div onClick={this.props.onClick}>...</div>;
    }
}
```

Здесь каждый дочерний компонент из списка должен получить свою уникальную функцию, в которой через замыкание хранится

индекс элемента.

В этом случае избавиться от создания новых функций при каждой отрисовке сложнее. Существует несколько способов это сделать.

Изменение интерфейса дочернего компонента

Для применения первого подхода нужно создать только одну функцию обратного вызова и передать ее во все дочерние компоненты. При этом ответственность за передачу в функцию нужных аргументов будет лежать на дочерних компонентах.

Пример с передачей id:

```
class ParentComponent extends React.Component {
    _onClick = (id) => {
        doSomehing(id);
    }

    render() {
        return (
            <div>
                {this.props.items.map((item) => (
                    <ChildComponent
                        onClick={this._onClick}
                        item={item}
                    />
                ))}
            </div>
        );
    }
}

class ChildComponent extends React.PureComponent {
    _onClick = () => {
        this.props.onClick(this.props.item.id)
    }

    render() {
```

```
        return <div onClick={this._onClick}>...</div>;
    }
}
```

Если внутри дочернего компонента нет данных, необходимых для функции обратного вызова, то их нужно будет в него передавать через свойства.

Пример с добавлением index в свойства:

```
class ParentComponent extends React.Component {
    _onClick = (index) => {
        doSomehing(index);
    }

    render() {
        return (
            <div>
                {this.props.items.map((item, index) => (
                    <ChildComponent
                        onClick={this._onClick}
                        item={item}
                        index={index}
                    />
                ))}
            </div>
        );
    }
}

class ChildComponent extends React.PureComponent {
    _onClick = () => {
        this.props.onClick(this.props.index)
    }

    render() {
        return <div onClick={this._onClick}>...</div>;
    }
}
```

Данный способ не всегда можно использовать. Во-первых, у нас может не быть доступа к исходному коду дочернего компонента. Во-вторых, добавление новых свойств может быть нежелательно из-за того, что это может отрицательно сказаться на понятности и целостности его интерфейса.

Выделение компонента

Если использовать предыдущий способ нет возможности или если это нежелательно, то можно создать отдельный компонент, который будет отвечать за передачу в функцию обратного вызова дополнительного параметра.

Пример выделения компонента:

```
class ParentComponent extends React.Component {
    _onClick = (index) => {
        doSomehing(index);
    }

    render() {
        return (
            <div>
                {this.props.items.map((item, index) => (
                    <ChildWrapperComponent
                        onClick={this._onClick}
                        item={item}
                        index={index}
                    />
                ))}
            </div>
        );
    }
}

class ChildWrapperComponent extends React.PureComponent {
    _onClick = () => {
        this.props.onClick(this.props.index)
```

```
    }

    render() {
        return (
            <ChildComponent
                onClick={this._onClick}
                item={item}
            />
        );
    }
}
```

Этот метод можно применять практически в любых ситуациях. Пожалуй, единственный его недостаток — это то, что он относительно трудоемкий. Придется написать отдельный класс, только чтобы избавиться от создания функций при вызове render.

Передача значений в качестве свойств нативным элементам

Также может быть использован способ, при котором в дочерние html-элементы (такие, например, как button и input) передается индекс массива (или другой ключ), а при наступлении события в функции обратного вызова эти данные извлекаются из html-элемента.

Пример с html-элементами:

```
class ParentComponent extends React.Component {
    _onClick = (e) => {
        doSomehing(e.target.value);
    }

    render() {
        return (
            <div>
```

```
                {this.props.items.map((item, index) => (
                    <button
                        type="button"
                        onClick={this._onClick}
                        value={index}
                    >
                        {item.title}
                    </button>
                ))}
            </div>
        );
    }
}
```

У html-элементов мы не можем реализовать shouldComponentUpdate, но, как показали тесты, у них есть встроенный механизм оптимизации и неизменяемость их свойств также может существенно уменьшить время обновления таких элементов.

Кеширование функций обратного вызова

Для применения этого способа нужно один раз создать список функций обратного вызова для всех дочерних компонентов, а при последующих отрисовках использовать ранее созданные функции. Один из вариантов такой реализации:

Кеширование колбеков:

```
class ParentComponent extends React.Component {
    _callbacks = {};

    _getOnClick = (index) => {
        if (!this._callbacks[index]) {
            this._callbacks[index] = () => doSomehing(index);
        }
```

```
        return this._callbacks[index];
    }

    render() {
        return (
            <div>
                {this.props.items.map((item, index) => (
                    <ChildComponent
                        onClick={this._getOnClick(index)}
                        item={item}
                    />
                ))}
            </div>
        );
    }
}
```

Данный способ является относительно трудоемким. Если в функцию обратного вызова помимо индекса необходимо передавать еще какие-нибудь данные, то для ее правильной работы потребуется реализовать более сложную логику и написать больше кода. Кроме того, эту функцию нужно будет копировать в каждый компонент, где это необходимо, и, возможно, вставлять этот код несколько раз в один и тот же класс. Поэтому, чтобы избежать дублирования кода в каждом классе, можно создать модуль, который выполнял бы необходимые действия.

Для примера таких модулей можно привести два пакета:

- cached-bind
- react-cached-callback

Оба пакета имеют одинаковые возможности, но у них разный интерфейс.
Они позволяют для каждого дочернего компонента из списка один

раз создать функцию обратного вызова и использовать ее при каждой последующей отрисовке.

Пример использования cached-bind:

```
import bind from 'cached-bind';

class ParentComponent extends React.Component {
    _onClick(index) {
        doSomehing(index);
    }

    render() {
        return (
            <div>
                {this.props.items.map((item, index) => (
                    <ChildComponent
                        onClick={bind(this, '_onClick', index)}
                        item={item}
                    />
                ))}
            </div>
        );
    }
}
```

Пример использования react-cached-callback:

```
import cached from 'react-cached-callback';

class ParentComponent extends React.Component {
    @cached
    _getOnClick(index) {
        return () => doSomehing(index);
    }

    render() {
        return (
            <div>
                {this.props.items.map((item, index) => (
                    <ChildComponent
```

```
                        onClick={this._getOnClick(index)}
                        item={item}
                    />
                ))}
            </div>
        );
    }
}
```

Чтобы понять, какую именно сохраненную функцию необходимо вернуть, оба пакета при вызове должны определить идентификатор дочернего компонента. В качестве идентификатора можно использовать, например, индекс элемента в массиве.

cached-bind должен получить такой идентификатор третьим аргументом при вызове функции bind.
A react-cached-callback по умолчанию считает идентификатором первый передаваемый в оригинальную функцию аргумент. Узнать подробности интерфейсов можно в описании пакетов.

В заключение

React — быстрый фреймворк, и значительная часть приложений будет работать с достаточной производительностью без дополнительных усилий. Но, при увеличении размера приложения, разработчику может потребоваться применять различные приемы оптимизации и самому следить за тем, чтобы свойства компонентов не изменялись без необходимости.

Многие библиотеки для управления состоянием (например, redux или MobX) в настоящее время имеют собственные способы

оптимизации react-компонентов. Но даже при их использовании необязательная генерация новых объектов и функций при отрисовке компонентов может привести к значительным задержкам при обновлении пользовательского интерфейса.

Похожие публикации

Как мы в Tutu.ru добиваемся эффективности каждого из 9000+ UI-тестов 12

Sakharov • 16 октября 2017 в 10:06

#!ProgMsk Meetup: Виртуализация и DevOps в офисе Tutu.ru 4

0xy • 27 июля 2017 в 13:21

CocoaHeads Russia в офисе Туту.ру 0

0xy • 20 июня 2017 в 10:08

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке 14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада 4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе 25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации 2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни — 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 — 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ — 40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству? — 8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения — 17

AlexeyNadezhin • вчера в 20:32