REACTJS*

React медленный, React быстрый: оптимизация React-приложения на практике

ПЕРЕВОД

KarafiziArthur 25 апреля 2017 в 18:53 26,6k

Оригинал: François Zaninotto

Всем привет! Хочу поделиться своим переводом статьи React is Slow, React is Fast: Optimizing React Apps in Practice автора François Zaninotto. Надеюсь, это кому-то будет полезным.

Краткое содержание:

React может быть медленным. Я хочу сказать, что любое React приложение среднего размера может оказаться медленным. Но прежде, чем искать ему замену, вы должны знать, что и любое

среднее приложение на Angular или Ember может также оказаться медленным.

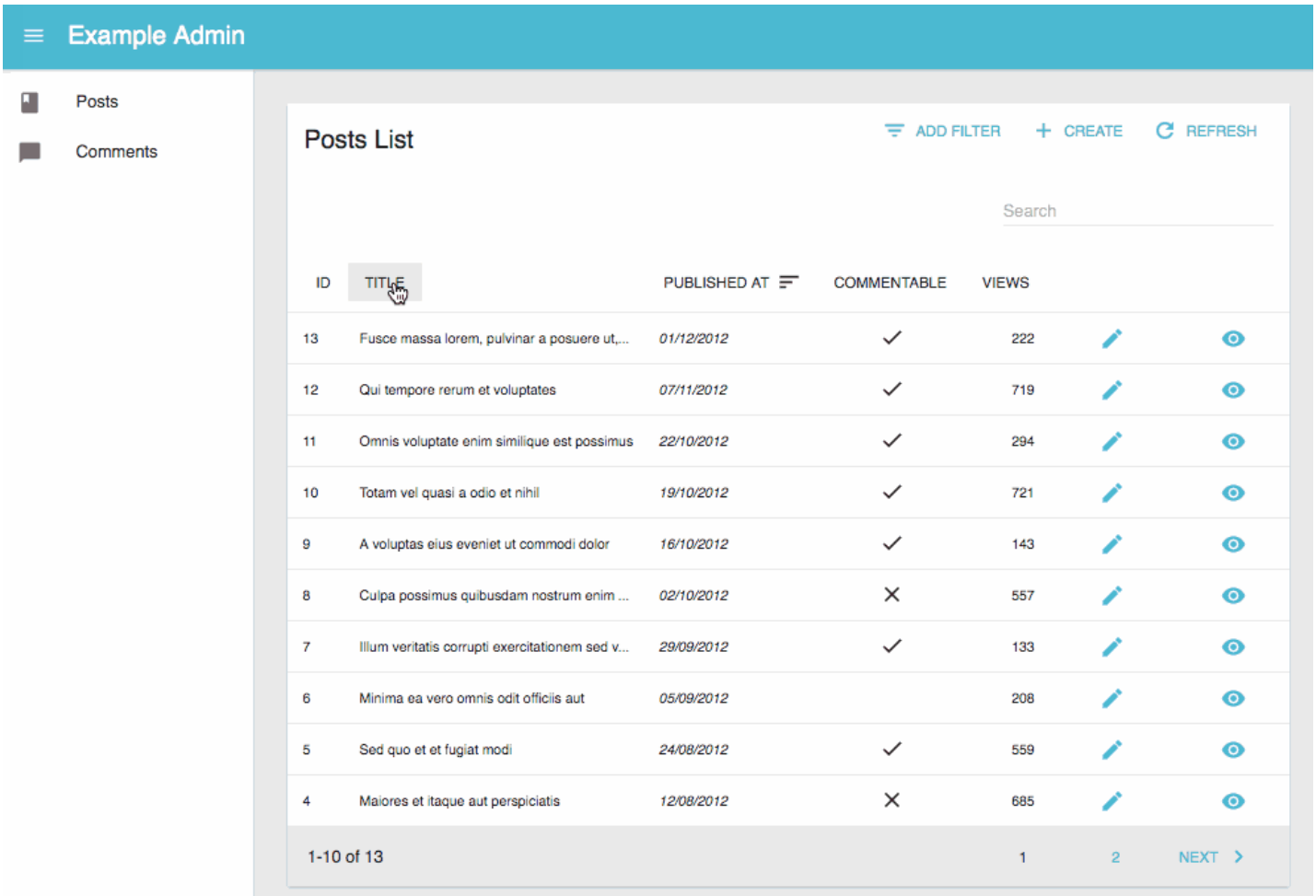
Хорошая новость в том, что если вы действительно заботитесь о производительности, то **сделать React приложение очень быстрым довольно легко**. *Об этом — далее в статье.*

Измерение производительности React

Что я подразумеваю под "медленным"? Позвольте привести пример:

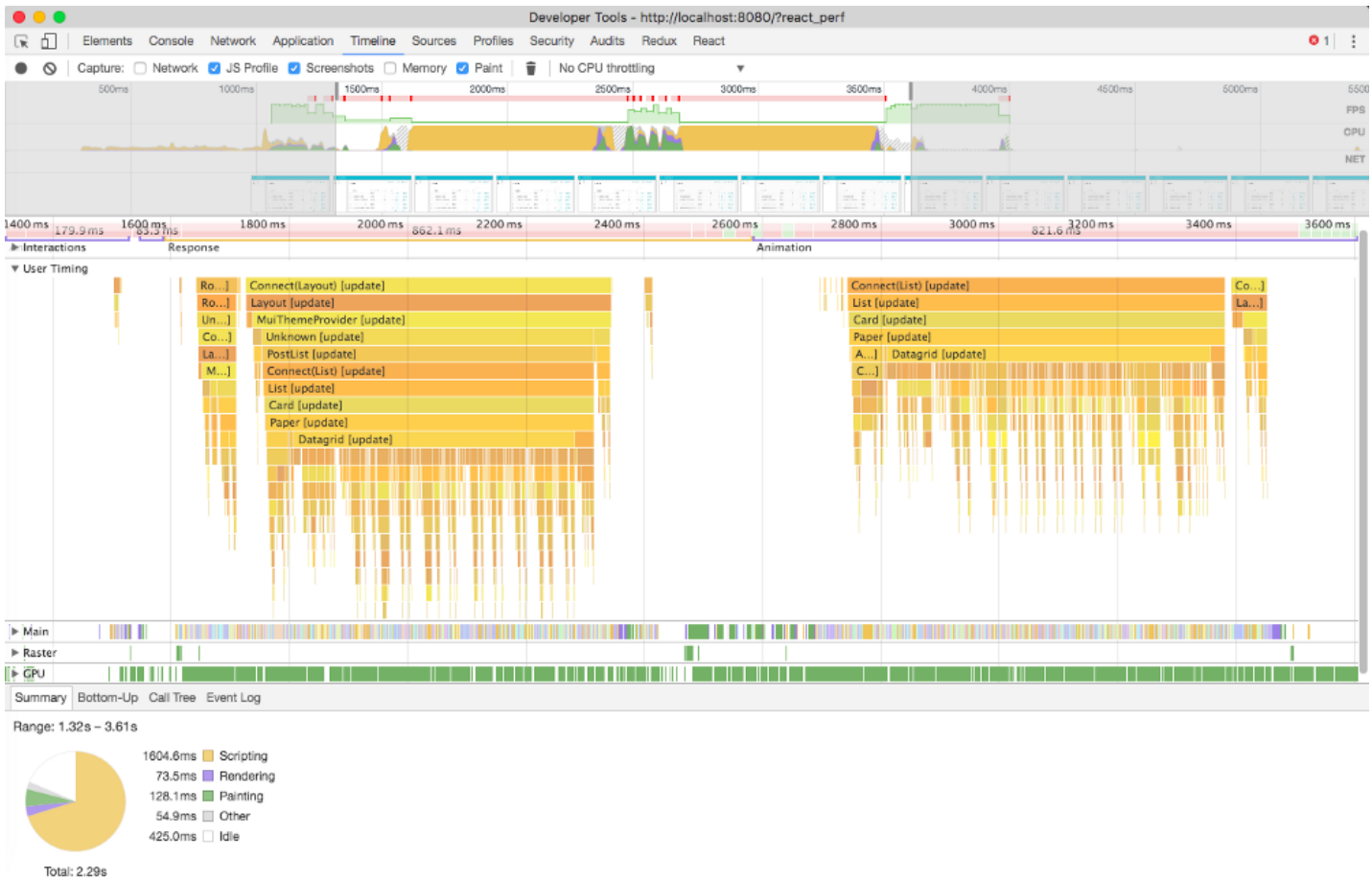
Я работаю над одним open-source проектом, который называется admin-on-rest, использующий material-ui и Redux для предоставления графического интерфейса (GUI) админ-панели для любого API. В этом приложении есть страница, отображающая список записей в виде таблицы. Когда пользователь изменяет порядок сортировки, или переходит на следующую страницу, или фильтрует вывод, интерфейс не так отзывчив, как хотелось бы.

На следующем анимированном скринкасте, замедленном в 5 раз, показано, как происходит обновление:



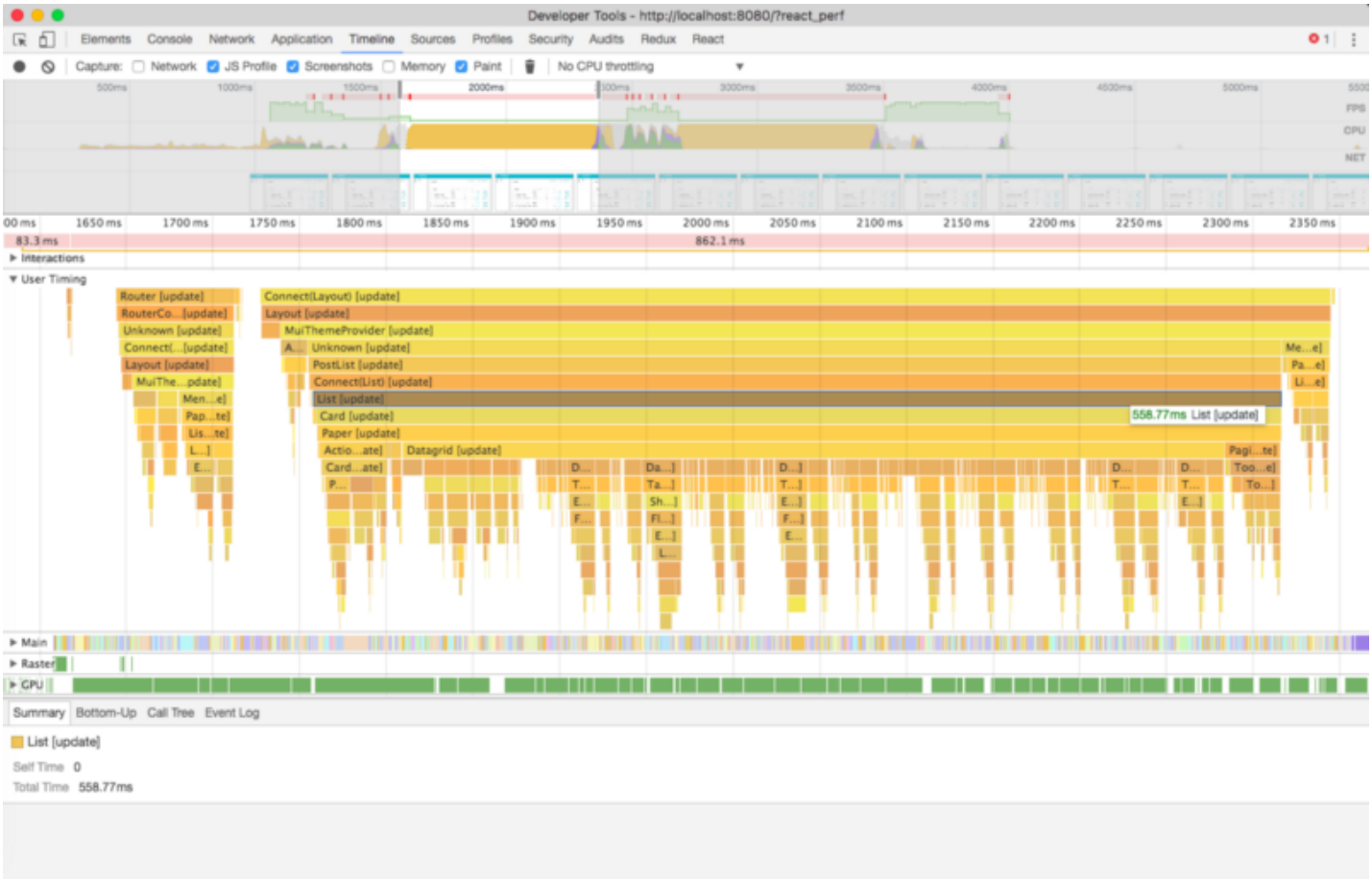
Чтобы понять, что происходит, я добавляю в конце URL `?react_perf`. Это активирует возможность профилирования компонентов, которая доступна с версии React 15.4. Сначала я жду начальной загрузки таблицы с данными. Далее, я открываю вкладку Timeline в инструментах разработчика в Chrome, кликаю на кнопку "Запись" и нажимаю на заголовок таблицы на странице, чтобы обновить порядок сортировки.

После обновления данных, я снова кликаю на кнопку записи, чтобы остановить её. Chrome отобразит жёлтый график под меткой "User Timing".



Если вы никогда не видели этого графика, он может показаться пугающим, но, на самом деле, им очень просто пользоваться. Этот график показывает время работы каждого из ваших компонентов. Он не показывает время внутренних компонентов React (вы всё равно не можете их оптимизировать), таким образом он позволяет вам сфокусироваться на оптимизации своего собственного кода.

Временная шкала выводит этапы записи работы приложения и позволяет приблизить тот момент, когда я кликнул на заголовке таблицы:



Похоже, что моё приложение перерисовывает компонент `<List>` сразу после клика на кнопку сортировки, и *перед* получением данных через REST. Это занимает более 500 мс. Приложение просто обновляет иконку сортировки в заголовке таблицы и отображает серый экран, обозначающий загрузку данных.

Иначе говоря, приложение занимает 500 мс, чтобы визуально отобразить ответную реакцию на клик. Полсекунды это значительный показатель — эксперты по UI говорят, что пользователи считают реакцию приложения мгновенной, только когда она меньше 100 мс. Реакция приложения более 100 мс — вот то, что я называю "медленным".

Почему ты обновился?

На графике выше можно увидеть много крошечных ямок. Это плохой знак, так как это означает, что множество компонентов перерисовались. На графике видно, что обновление `<Datagrid>` занимает больше всего времени. Почему приложение обновило всю таблицу с данными прежде, чем получило новые данные? *Давайте разбираться.*

Попытки понять причины перерисовки часто подразумевают добавление `console.log()` в `render()` функцию. Для функциональных компонентов вы можете использовать следующий компонент высшего порядка (HOC):

```
// src/log.js
const log = BaseComponent => props => {
 console.log(`Rendering ${BaseComponent.name}`);
 return <BaseComponent {…props} />;
}
export default log;

// src/MyComponent.js
import log from ‘./log’;
export default log(MyComponent);
```

> ***Совет:*** *стоит также отметить why-did-you-update — ещё один инструмент для эффективности React. Этот npm пакет заставляет React выводить в консоль предупреждения всякий раз, когда компонент перерисовывается с теми же props. Предупреждаю: вывод в консоли довольно подробный и он не работает с функциональными компонентами.*

В примере, когда пользователь кликает на заголовке столбца, приложение производит действие, изменяющее state: порядок сортировки списка (`currentSort`) обновлён. Это изменение state запускает перерисовку страницы `<List>`, которая в свою очередь перерисовывает весь компонент `<Datagrid>`. Мы хотим, чтобы заголовок таблицы немедленно отрисовал изменение иконки сортировки, как ответ на действия пользователя.

Обычно, React становится медленным не из-за одного медленного компонента (что будет отображено на графике, как одна большая яма). **В большинстве же случаев, React становится медленным из-за бесполезной перерисовки множества компонентов.**

Вы возможно читали, что VirtualDOM в React очень быстрый. Это правда, но в приложении среднего размера полная перерисовка может легко содержать в себе отрисовку сотни компонентов. Даже самый быстрый шаблонизатор VirtualDOM не может сделать это меньше, чем за 16 ms.

Оптимизация через разбиение на компоненты

Вот метод `render()` компонента `<Datagrid>`:

```
// Datagrid.js
render() {
    const {
        resource,
        children,
        ids,
        data,
        currentSort
```

```
    } = this.props;

    return (
        <table>
            <thead>
                <tr>
                    {Children.map(children, (field, index) =>
                        <DatagridHeaderCell
                            key={index}
                            field={field}
                            currentSort={currentSort}
                            updateSort={this.updateSort}
                        />
                    )}
                </tr>
            </thead>
            <tbody>
                {ids.map(id => (
                    <tr key={id}>
                        {Children.map(children, (field, index) =>
                            <DatagridCell
                                record={data[id]}
                                key={`${id}-${index}`}
                                field={field}
                                resource={resource}
                            />
                        )}
                    </tr>
                ))}
            </tbody>
        </table>
    );
}
```

Кажется, что это очень простая реализация табличных данных, но она крайне неэффективна. Каждый `<DatagridCell>` вызывает отрисовку как минимум двух или трёх компонентов. Как вы можете увидеть на анимированном скринкасте интерфейса в начале статьи, список содержит 7 столбцов и 11 строк, и это означает, что 7*11*3 = 231 компонент перерисовывается. И всё это пустая трата времени, так как изменению подвергается только `currentSort`.

Несмотря на то, что React не обновляет реальный DOM (при условии, что VirtualDOM не изменился), то это всё равно занимает около 500 мс для обработки всех компонентов.

Чтобы избежать бесполезной перерисовки тела таблицы, для начала я должен *_извлечь_* его:

```
// Datagrid.js
render() {
    const {
        resource,
        children,
        ids,
        data,
        currentSort
    } = this.props;

    return (
        <table>
            <thead>
                <tr>
                    {React.Children.map(children, (field, index) =>
                        <DatagridHeaderCell
                            key={index}
                            field={field}
                            currentSort={currentSort}
                            updateSort={this.updateSort}
                        />
                    )}
                </tr>
            </thead>
            <DatagridBody resource={resource} ids={ids} data={data}>
                {children}
            </DatagridBody>
            </table>
        );
    );
}
```

Я создал новый `<DatagridBody>` компонент путём извлечения логики из тела таблицы:

```
// DatagridBody.js
import React, { Children } from 'react';
const DatagridBody = ({ resource, ids, data, children }) => (
    <tbody>
        {ids.map(id => (
            <tr key={id}>
                {Children.map(children, (field, index) =>
                    <DatagridCell
                        record={data[id]}
                        key={`${id}-${index}`}
                        field={field}
                        resource={resource}
                    />
                )}
            </tr>
        ))}
    </tbody>
);

export default DatagridBody;
```

Само по себе, извлечение тела таблицы никак не оказывает влияния на производительность, но оно открывает возможности для оптимизации. Большие компоненты общего назначения трудно оптимизировать. С маленькими же компонентами, отвечающими только за что-то одно, справиться легче.

shouldComponentUpdate

Документация React очень чётко описывает способ избежать бесполезной перерисовки путём использования `shouldComponentUpdate()`. По умолчанию, React *всегда отображает* компонент в VirtualDOM. Иными словами, ваша

работа как разработчика, проверять, не изменились ли props компонента, и если нет, то пропустить его перерисовку.

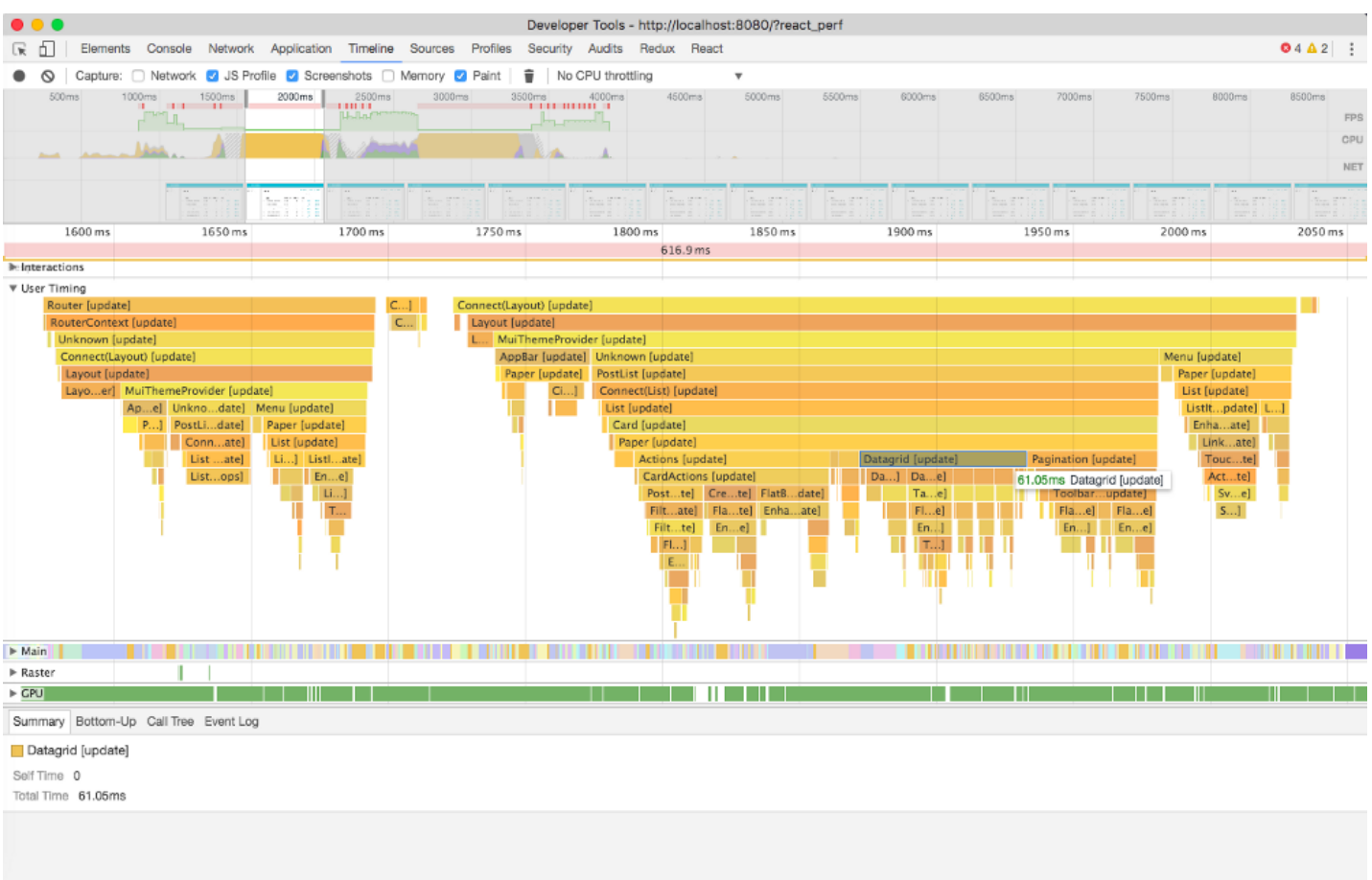
В случае с компонентом `<DatagridBody>`, в нём не должно быть перерисовки пока props не изменится.

Поэтому компонент должен выглядеть так:

```
import React, { Children, Component } from 'react';

class DatagridBody extends Component {
    shouldComponentUpdate(nextProps) {
        return (nextProps.ids !== this.props.ids
             || nextProps.data !== this.props.data);
    }

    render() {
        const { resource, ids, data, children } = this.props;
        return (
            <tbody>
                {ids.map(id => (
                    <tr key={id}>
                        {Children.map(children, (field, index) =>
                            <DatagridCell
                                record={data[id]}
                                key={`${id}-${index}`}
                                field={field}
                                resource={resource}
                             />
                        )}
                    </tr>
                ))}
            </tbody>
        );
    }
}

export default DatagridBody;
```

> ***Совет:*** *Вместо того, чтобы прописывать* ***shouldComponentUpdate()*** *вручную, я мог бы наследовать этот класс от* ***PureComponent*** *вместо* ***Component****.* ***PureComponent*** *будет сравнивать все* ***props*** *используя строгое сравнение (****===****) и перерисовывать, только если props изменились. Но я знаю, что* ***resource*** *и* ***children*** *не могут измениться в данном контексте, поэтому мне не нужно их сравнивать.*

Благодаря этой оптимизации перерисовка `<Datagrid>` после клика на заголовке таблицы пропускает её содержимое и все 231 компонент. Это уменьшило время обновления с 500 мс до 60 мс. Это чистое повышение производительности более чем на 400 мс!



> ***Совет:*** *Не обманывайтесь шириной графика, он приближен даже больше, чем на предыдущем графике. Это определённо лучше!*

Метод `shouldComponentUpdate` удалил множество ямок на графике и сократил общее время отрисовки. Я могу использовать этот же способ, чтобы избежать ещё больших перерисовок (например, не перерисовывать боковую панель, кнопки действий, не изменившиеся заголовки таблицы, пагинацию). Примерно, после часа возни со всем этим, вся страница отрисовывается всего за 100 мс после клика на заголовок столбца. Это достаточно быстро — даже если осталось ещё что оптимизировать.

Добавление `shouldComponentUpdate` метода может показаться громоздким, но если вы заботитесь о производительности, то большинство компонентов должны содержать его.

Но не вставляйте его везде, где только можно — выполнение `shouldComponentUpdate` в довольно простых компонентах может порой замедлить его отрисовку. Не делайте этого слишком рано в жизненном цикле приложения. Добавляйте этот метод лишь по мере роста приложения, когда вы сможете определить проблемы производительности в ваших компонентах.

Recompose

Я не особо доволен предыдущими изменениями в `<DatagridBody>`: из-за `shouldComponentUpdate` я должен был трансформировать простой, функциональный компонент в класс.

Это добавляет много строк кода, каждая из которых имеет свою цену — в виде написания, отладки и поддержки.

К счастью, вы можете реализовать логику `shouldComponentUpdate` в компоненте высшего порядка (HOC), благодаря recompose. Это функциональный инструментарий для React, предоставляющий, например, HOC функцию `pure()`:

```
// DatagridBody.js
import React, { Children } from 'react';
import pure from 'recompose/pure';

const DatagridBody = ({ resource, ids, data, children }) => (
    <tbody>
        {ids.map(id => (
            <tr key={id}>
                {Children.map(children, (field, index) =>
                    <DatagridCell
                        record={data[id]}
                        key={`${id}-${index}`}
                        field={field}
                        resource={resource}
                    />
                )}
            </tr>
        ))}
    </tbody>
);

export default pure(DatagridBody);
```

Единственное отличие между этим кодом и начальной реализацией — в последней строчке: я экспортирую `pure(DatagridBody)` вместо `DatagridBody`. `pure` похожа на `PureComponent`, но без лишнего бойлерплейта.

Я даже могу быть более конкретным и ориентироваться только на те props, о которых я точно знаю, что они могут измениться, используя `shouldUpdate()` вместо `pure()`:

```
// DatagridBody.js
import React, { Children } from 'react';
import shouldUpdate from ‘recompose/shouldUpdate’;

const DatagridBody = ({ resource, ids, data, children }) => (
    ...
);

const checkPropsChange = (props, nextProps) =>
 (nextProps.ids !== props.ids ||
  nextProps.data !== props.data);

export default shouldUpdate(checkPropsChange)(DatagridBody);
```

`checkPropsChange` — это чистая функция, и я даже могу экспортировать её для unit-тестирования.

Библиотека recompose предлагает более эффективные HOC, такие как `onlyUpdateForKeys()`, который совершает ту же проверку, что я делал в своей `checkPropsChange`:

```
// DatagridBody.js
import React, { Children } from 'react';
import onlyUpdateForKeys from ‘recompose/onlyUpdateForKeys’;

const DatagridBody = ({ resource, ids, data, children }) => (
    ...
);

export default onlyUpdateForKeys([‘ids’, ‘data’])(DatagridBody);
```

Я горячо рекомендую `recompose`. Помимо оптимизации производительности, она помогает вам извлекать логику выборки данных, составлять HOC и работать с props в функциональном и тестируемом стиле.

Redux

Если для управления состоянием приложения вы используете Redux (*который я также рекомендую*), тогда подключенные к нему компоненты уже чистые. Нет нужды в каком-либо другом HOC.

Просто запомните, если изменилось всего одно свойство, то подключенный компонент перерисуется, — и все его потомки тоже. Поэтому, если вы используете Redux для компонентов страницы, вам следует использовать `pure()` или `shouldUpdate()` для нижележащих по дереву компонентов.

Но также помните, что Redux использует строгое сравнение для props. Поскольку Redux связывает state с props компонента, то если вы будете изменять объект в state, то Redux просто проигнорирует это. И вот по этой причине вы должны использовать **иммутабельность** в ваших reducers.

К примеру, в admin-on-rest, клик по заголовку таблицы диспатчит `SET_SORT` action. Reducer, который слушает этот action, должен заменить объект в state, а не *обновить* его:

```
// listReducer.js
export const SORT_ASC = 'ASC';
export const SORT_DESC = 'DESC';

const initialState = {
    sort: 'id',
    order: SORT_DESC,
    page: 1,
    perPage: 25,
    filter: {},
};

export default (previousState = initialState, { type, payload }) =>
{
    switch (type) {
    case SET_SORT:
        if (payload === previousState.sort) {
            // обратим порядок сортировки
            return {
                ...previousState,
                order: oppositeOrder(previousState.order),
                page: 1,
            };
        }
        // заменим поле sort
        return {
            ...previousState,
            sort: payload,
            order: SORT_ASC,
            page: 1,
        };
    // ...
    default:
        return previousState;
    }
};
```

Следуя коду этого reducer-а, когда Redux проверяет state на изменения, используя тройное сравнение, он обнаруживает, что объект state изменился и перерисовывает таблицу с данными. Но если бы мы мутировали state, то Redux бы пропустил это изменение и соответственно ничего бы не перерисовал:

```
// не повторяйте это в домашних условиях
export default (previousState = initialState, { type, payload }) =>
{
    switch (type) {
    case SET_SORT:
        if (payload === previousState.sort) {
            // никогда так не делайте
            previousState.order= oppositeOrder(previousState.order);
            return previousState;
        }
        // и так тоже не делайте
        previousState.sort = payload;
        previousState.order = SORT_ASC;
        previousState.page = 1;
        return previousState;
    // ...
    default:
        return previousState;
    }
};
```

Чтобы писать иммутабельные reducers, некоторые разработчики используют библиотеку immutable.js, которая также от Facebook. Но с тех пор, как деструктуризация ES6 упростила выборочную замену в свойствах компонента, то я не считаю, что эта библиотека необходима. Кроме того, она тяжеловесная (60 kB), поэтому подумайте дважды, прежде чем добавить её в зависимости своего проекта.

Reselect

Для предотвращения ненужной отрисовки подключенных к Redux компонентов, вы также должны убедиться, что функция `mapStateToProps` не возвращает новые объекты при каждом её вызове.

Возьмём, к примеру, компонент `<List>` в admin-on-rest. Он берёт из state список записей для текущего ресурса (например, посты, комментарии, и т.д.) с помощью следующего кода:

```
// List.js
import React from 'react';
import { connect } from 'react-redux';

const List = (props) => ...
const mapStateToProps = (state, props) => {
    const resourceState = state.admin[props.resource];
    return {
        ids: resourceState.list.ids,
        data: Object.keys(resourceState.data)
            .filter(id => resourceState.list.ids.includes(id))
            .map(id => resourceState.data[id])
            .reduce((data, record) => {
                data[record.id] = record;
                return data;
            }, {}),
    };
};

export default connect(mapStateToProps)(List);
```

State содержит массив всех ранее загруженных записей, проиндексированных ресурсом. Например, `state.admin.posts.data` содержит список постов:

```
{
   23: { id: 23, title: “Hello, World”, /* … */ },
   45: { id: 45, title: “Lorem Ipsum”, /* … */ },
   67: { id: 67, title: “Sic dolor amet”, /* … */ },
}
```

Функция `mapStateToProps` фильтрует объект state и возвращает только те записи, которые фактически отображаются в списке. Что-то вроде этого:

```
{
    23: { id: 23, title: “Hello, World”, /* … */ },
    67: { id: 67, title: “Sic dolor amet”, /* … */ },\
}
```

Проблема в том, что при каждом запуске функции `mapStateToProps`, она возвращает новый объект, даже если внутренние объекты не изменились. Как следствие, компонент `<List>` каждый раз перерисовывается, когда в state что-то меняется — в то время, как, при изменении date или id, изменяться должны только id.

Reselect решает это проблему с помощью мемоизации. Вместо вычисления props напрямую в `mapStateToProps`, вы используете *selector* из reselect, который возвращает тот же объект, если никаких изменений с ним не производилось.

```
import React from 'react';
import { connect } from 'react-redux';
import { createSelector } from 'reselect'

const List = (props) => ...

const idsSelector = (state, props) =>
    state.admin[props.resource].ids
const dataSelector = (state, props) =>
    state.admin[props.resource].data

const filteredDataSelector = createSelector(
  idsSelector,
  dataSelector
```

```
  (ids, data) => Object.keys(data)
      .filter(id => ids.includes(id))
      .map(id => data[id])
      .reduce((data, record) => {
          data[record.id] = record;
          return data;
      }, {})
)

const mapStateToProps = (state, props) => {
    const resourceState = state.admin[props.resource];
    return {
        ids: idsSelector(state, props),
        data: filteredDataSelector(state, props),
    };
};

export default connect(mapStateToProps)(List);
```

Теперь компонент `<List>` будет перерисовываться только при изменении подмножества state.

Что касается recompose, selectors — это чистые функции, лёгкие для тестирования и компоновки. Написание своего selector для подключенных к Redux компонентов — это хорошая практика.

Остерегайтесь объектных литералов в JSX

Однажды ваш компонент станет ещё более "чистым", и вы можете обнаружить у себя в коде плохие паттерны, приводящие к бесполезным перерисовкам. Наиболее общим примером этого является использование объектных литералов в JSX, которые мне нравится называть "**Печально известные {{**". Позвольте привести пример:

```
import React from 'react';
import MyTableComponent from './MyTableComponent';

const Datagrid = (props) => (
    <MyTableComponent style={{ marginTop: 10 }}>
        ...
    </MyTableComponent>
)
```

Свойство `style` компонента `<MyTableComponent>` получает новое значение каждый раз, когда компонент `<Datagrid>` отрисовывается. Таким образом, даже если `<MyTableComponent>` чистый, он всё равно будет перерисовываться при перерисовке `<Datagrid>`. По факту, каждый раз вы передаёте объектный литерал как свойство в дочерний компонент, вы нарушаете чистоту. Решение простое:

```
import React from 'react';
import MyTableComponent from './MyTableComponent';

const tableStyle = { marginTop: 10 };
const Datagrid = (props) => (
    <MyTableComponent style={tableStyle}>
        ...
    </MyTableComponent>
)
```

Это выглядит весьма просто, но я столько раз видел эту ошибку, что у меня развилось чувство по обнаружению "печально известных `{{`" в JSX. Я регулярно заменяю их константами.

Следующий подозреваемый в краже чистоты компонента это `React.CloneElement()`. Если вы передаёте свойство в качестве

значения второго параметра, то склонированный элемент будет получать новые props при каждой отрисовке.

```
// плохо
const MyComponent = (props) =>
    <div>{React.cloneElement(Foo, { bar: 1 })}</div>;

// хорошо
const additionalProps = { bar: 1 };
const MyComponent = (props) =>
    <div>{React.cloneElement(Foo, additionalProps)}</div>;
```

Я обжёгся на этом пару раз с material-ui на примере следующего кода:

```
import { CardActions } from 'material-ui/Card';
import { CreateButton, RefreshButton } from 'admin-on-rest';

const Toolbar = ({ basePath, refresh }) => (
    <CardActions>
        <CreateButton basePath={basePath} />
        <RefreshButton refresh={refresh} />
    </CardActions>
);

export default Toolbar;
```

Хотя компонент `<CreateButton>` чистый, он отрисовывается каждый раз, когда отрисовывается `<Toolbar>`. Это всё потому, что компонент `<CardAction>` из material-ui добавляет специальный стиль первому потомку для размещения на полях — и делает он это с помощью объектного литерала. Поэтому `<CreateButton>` получает разный объект `style` каждый раз. Я смог решить это с помощью HOC функции `onlyUpdateForKeys()` из recompose.

```
// Toolbar.js
import onlyUpdateForKeys from 'recompose/onlyUpdateForKeys';

const Toolbar = ({ basePath, refresh }) => (
    ...
);

export default onlyUpdateForKeys(['basePath', 'refresh'])(Toolbar);
```

Заключение

Есть ещё много вещей, которые нужно сделать, чтобы поддерживать приложение на React быстрым (использовать ключи, ленивую загрузку тяжёлых маршрутов, пакет `react-addons-perf`, ServiceWorkers для кэширования состояния приложения, добавить изоморфности, и т.д.), но корректная реализация `shouldComponentUpdate` — это первый и самый действенный шаг.

Сам по себе, React — не быстрый, но он предлагает все инструменты, чтобы сделать быстрым приложение любого размера.

Это выглядит нелогичным, особенно, когда множество фреймворков предлагают альтернативы React, утверждая, что они быстрее его в N раз. Но React ставит во главу угла удобство и опыт разработчика, а не производительность. Это та причина, по которой разработка больших приложений с React это приятный опыт, без плохих сюрпризов и со стабильным темпом реализации.

Не забывайте время от времени профилировать ваше приложение и посвящать некоторое время на добавление `pure()` вызовов при необходимости. Но не делайте этого в самом начале, и не тратьте слишком много времени на оптимизацию каждого компонента — за исключением, если вы не делаете это под мобильные устройства. И не забывайте тестировать на разных устройствах, чтобы получать хорошие впечатления об отзывчивости вашего приложения с точки зрения пользователя.

Если вы хотите узнать больше об оптимизации производительности React, то вот список отличных статей по этой теме:

- React Rally — Animated — React Performance Toolbox — Потрясающий набор слайдов от Christopher Chedeau (Vjeux), одного из разработчиков React Native. Кстати, тоже француз.
- Progressive Web Apps with React.js — Part 2 — Page Load Performance — статья от Addy Osmany, который работает в Google и пишет много статей о производительности.
- Optimizing the Performance of Your React Application — статья, сфокусированная на пакете `react-addons-perf` для более точного профилирования React приложения.
- React Higher Order Components in depth — интересное введение в Render Hijacking.
- A Deep Dive into React Perf Debugging — статья, шаг за шагом описывающая отладку сессий с помощью Chrome Dev Tools.
- Making React reactive- the pursuit of high performing, easily maintainable React apps — статья о том, как избежать перерисовки, применяя Observables.

Проголосовать:

+34

Поделиться:

Сохранить:

Комментарии (59)

Похожие публикации

Оптимизируем redux хранилище для более производительных изменений

5

lavrton • 23 декабря 2016 в 00:14

RxConnect — когда React встречает RxJS

25

ПЕРЕВОД

bsideup • 5 сентября 2016 в 14:58

Текстовый туториал по React.js и Redux на русском

22

ИЗ ПЕСОЧНИЦЫ

maxfarseer • 14 марта 2016 в 16:54

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