

РАЗРАБОТКА ВЕБ-САЙТОВ*, REACTJS*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

Новшества серверного рендеринга в React 16

ПЕРЕВОД

ru_vds 2 октября 2017 в 15:21 👁 19,6k

Оригинал: [Sasha Aickin](#)

Вышел React 16! Рассказывая об этом событии, можно упомянуть множество замечательных новостей (вроде архитектуры ядра Fibers), но лично меня больше всего восхищают улучшения серверного рендеринга. Предлагаю подробно всё это разобрать и сравнить с тем, что было раньше. Надеюсь, серверный рендеринг в React 16 понравится вам так же, как он понравился мне.



Как работает SSR в React 15

Для начала вспомним, как серверный рендеринг (Server-Side Rendering, SSR) выглядит в React 15. Для выполнения SSR обычно поддерживают сервер, основанный на Node,

использующий Express, Napi или Koa, и вызывают `renderToString` для преобразования корневого компонента в строку, которую затем записывают в ответ сервера:

```
// используем Express
import { renderToString } from "react-dom/server"
import MyPage from "./MyPage"
app.get("/", (req, res) => {
  res.write("<!DOCTYPE html><html><head><title>My Page</title>
</head><body>");
  res.write("<div id='content'>");
  res.write(renderToString(<MyPage/>));
  res.write("</div></body></html>");
  res.end();
});
```

Когда клиент получает ответ, клиентской подсистеме рендеринга, в коде шаблона, отдают команду восстановить HTML, сгенерированный на сервере, используя метод `render()`. Тот же метод используют и в приложениях, выполняющих рендеринг на клиенте без участия сервера:

```
import { render } from "react-dom"
import MyPage from "./MyPage"
render(<MyPage/>, document.getElementById("content"));
```

Если сделать всё правильно, клиентская система рендеринга может просто использовать HTML, сгенерированный на сервере, не обновляя DOM.

Как же SSR выглядит в React 16?

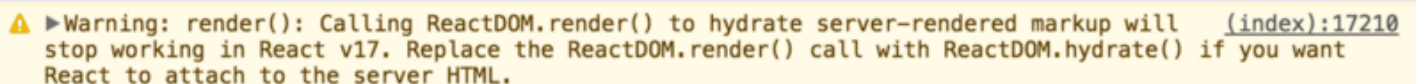
Обратная совместимость React 16

Команда разработчиков React показала чёткую ориентацию на обратную совместимость. Поэтому, если ваш код выполняется в React 15 без сообщений о применении устаревших конструкций, он должен просто работать в React 16 без дополнительных усилий с вашей стороны. Код, приведённый выше, например, нормально работает и в React 15, и в React 16.

Если случится так, что вы запустите своё приложение на React 16 и столкнётесь с ошибками, пожалуйста, [сообщите](#) о них! Это поможет команде разработчиков.

Метод `render()` становится методом `hydrate()`

Надо отметить, что переходя с React 15 на React 16, вы, возможно, столкнётесь со следующим предупреждением в браузере.

A screenshot of a browser warning message. It features a yellow background and a warning icon (a triangle with an exclamation mark). The text reads: "Warning: render(): Calling ReactDOM.render() to hydrate server-rendered markup will stop working in React v17. Replace the ReactDOM.render() call with ReactDOM.hydrate() if you want React to attach to the server HTML." On the right side, there is a link "(index):17210". Below the main text, there is a blue arrow pointing to the right.

⚠ Warning: render(): Calling ReactDOM.render() to hydrate server-rendered markup will stop working in React v17. Replace the ReactDOM.render() call with ReactDOM.hydrate() if you want React to attach to the server HTML. [\(index\):17210](#)

Очередное полезное предупреждение React. Метод `render()` теперь называется `hydrate()`

Оказывается, в React 16 теперь есть два разных метода для рендеринга на клиентской стороне. Метод `render()` для ситуаций, когда рендеринг выполняются полностью на клиенте, и метод `hydrate()` для случаев, когда рендеринг на клиенте основан на результатах серверного рендеринга. Благодаря обратной совместимости новой версии React, `render()` будет работать и в том случае, если ему передать то, что пришло с

сервера. Однако, эти вызовы следует заменить вызовами `hydrate()` для того, чтобы система перестала выдавать предупреждения, и для того, чтобы подготовить код к React 17. При таком подходе код, показанный выше, изменился бы так:

```
import { hydrate } from "react-dom"
import MyPage from "../MyPage"
hydrate(<MyPage/>, document.getElementById("content"))
```

React 16 может работать с массивами, строками и числами

В React 15 метод компонента `render()` должен всегда возвращать единственный элемент React. Однако, в React 16 рендеринг на стороне клиента позволяет компонентам, кроме того, возвращать из метода `render()` строку, число, или массив элементов. Естественно, это касается и SSR.

Итак, теперь можно выполнять серверный рендеринг компонентов, который выглядит примерно так:

```
class MyArrayComponent extends React.Component {
  render() {
    return [
      <div key="1">first element</div>,
      <div key="2">second element</div>
    ];
  }
}

class MyStringComponent extends React.Component {
  render() {
    return "hey there";
  }
}
```

```
}  
class MyNumberComponent extends React.Component {  
  render() {  
    return 2;  
  }  
}
```

Можно даже передать строку, число или массив компонентов методу API верхнего уровня `renderToString`:

```
res.write(renderToString([  
  <div key="1">first element</div>,  
  <div key="2">second element</div>  
]));  
// Не вполне ясно, зачем так делать, но это работает!  
res.write(renderToString("hey there"));  
res.write(renderToString(2));
```

Это должно позволить вам избавиться от любых `div` и `span`, которые просто добавлялись к вашему дереву компонентов React, что ведёт к общему уменьшению размеров HTML-документов.

React 16 генерирует более эффективный HTML

Если говорить об уменьшении размеров HTML-документов, то React 16, кроме того, радикально снижает излишнюю нагрузку, создаваемую SSR при формировании HTML-кода. В React 15 каждый HTML-элемент в SSR-документе имеет атрибут `data-reactid`, значение которого представляет собой монотонно возрастающие ID, и текстовые узлы иногда окружены комментариями с `react-text` и ID. Для того, чтобы это увидеть, рассмотрим следующий фрагмент кода:

```
renderToString(  
  <div>  
    This is some <span>server-generated</span> <span>HTML.</span>  
  </div>  
>);
```

В React 15 этот фрагмент сгенерирует HTML-код, который выглядит так, как показано ниже (переводы строк добавлены для улучшения читаемости кода):

```
<div data-reactroot="" data-reactid="1"  
  data-react-checksum="122239856">  
  <!-- react-text: 2 -->This is some <!-- /react-text -->  
  <span data-reactid="3">server-generated</span>  
  <!-- react-text: 4--> <!-- /react-text -->  
  <span data-reactid="5">HTML.</span>  
</div>
```

В React 16, однако, все ID удалены из разметки, в результате HTML, полученный из такого же фрагмента кода, окажется значительно проще:

```
<div data-reactroot="">  
  This is some <span>server-generated</span> <span>HTML.</span>  
</div>
```

Такой подход, помимо улучшения читаемости кода, может значительно уменьшить размер HTML-документов. Это просто здорово!

React 16 поддерживает произвольные атрибуты DOM

В React 15 система рендеринга DOM была довольно сильно ограничена в плане атрибутов HTML-элементов. Она убирала нестандартные HTML-атрибуты. В React 16, однако, и клиентская, и серверная системы рендеринга теперь пропускают произвольные атрибуты, добавленные к HTML-элементам. Для того, чтобы узнать больше об этом новшестве, почитайте [пост Дэна Абрамова](#) в блоге React.

SSR в React 16 не поддерживает обработчики ошибок и порталы

В клиентской системе рендеринга React есть две новых возможности, которые, к сожалению, не поддерживаются в SSR. Это — обработчики ошибок (Error Boundaries) и порталы (Portals). Обработчикам ошибок посвящён [отличный пост Дэна Абрамова](#) в блоге React. Учитывайте однако, что (по крайней мере сейчас) обработчики не реагируют на серверные ошибки. Для порталов, насколько я знаю, пока даже нет пояснительной статьи, но Portal API требует наличия узла DOM, в результате, на сервере его использовать не удастся.

React 16 производит менее строгую проверку на стороне клиента

Когда вы восстанавливаете разметку на клиентской стороне в React 15, вызов `ReactDOM.render()` выполняет посимвольное сравнение с серверной разметкой. Если по какой-либо причине будет обнаружено несовпадение, React выдаёт предупреждение в

режиме разработки и заменяет всё дерево разметки, сгенерированной на сервере, на HTML, который был сгенерирован на клиенте.

В React 16, однако, клиентская система рендеринга использует другой алгоритм для проверки правильности разметки, которая пришла с сервера. Эта система, в сравнении с React 15, отличается большей гибкостью. Например, она не требует, чтобы разметка, созданная на сервере, содержала атрибуты в том же порядке, в котором они были бы расположены на клиентской стороне. И когда клиентская система рендеринга в React 16 обнаруживает расхождения, она лишь пытается изменить отличающееся поддереву HTML, вместо всего дерева HTML.

В целом, это изменение не должно особенно сильно повлиять на конечных пользователей, за исключением одного факта: React 16, при вызове `ReactDOM.render()` / `hydrate()`, не исправляет несовпадающие HTML-атрибуты, сгенерированные SSR. Эта оптимизация производительности означает, что вам понадобится внимательнее относиться к исправлению несовпадений разметки, приводящих к предупреждениям, которые вы видите в режиме `development`.

React 16 не нужно компилировать для улучшения производительности

В React 15, если вы используете SSR в таком виде, в каком он оказывается сразу после установки, производительность оказывается далеко не оптимальной, даже в режиме `production`.

Это так из-за того, что в React есть множество замечательных предупреждений и подсказок для разработчика. Каждое из этих предупреждений выглядит примерно так:

```
if (process.env.NODE_ENV !== "production") {  
  // что-то тут проверить и выдать полезное  
  // предупреждение для разработчика.  
}
```

К сожалению, [оказывается](#), что `process.env` — это не обычный объект JavaScript, и обращение к нему — операция затратная. В итоге, даже если значение `NODE_ENV` установлено в `production`, частая проверка переменной окружения ощутимо замедляет серверный рендеринг.

Для того, чтобы решить эту проблему в React 15, нужно было бы скомпилировать SSR-код для удаления ссылок на `process.env`, используя что-то вроде [Environment Plugin](#) в Webpack, или плагин [transform-inline-environment-variables](#) для Babel. По опыту знаю, что многие не компилируют свой серверный код, что, в результате, значительно ухудшает производительность SSR.

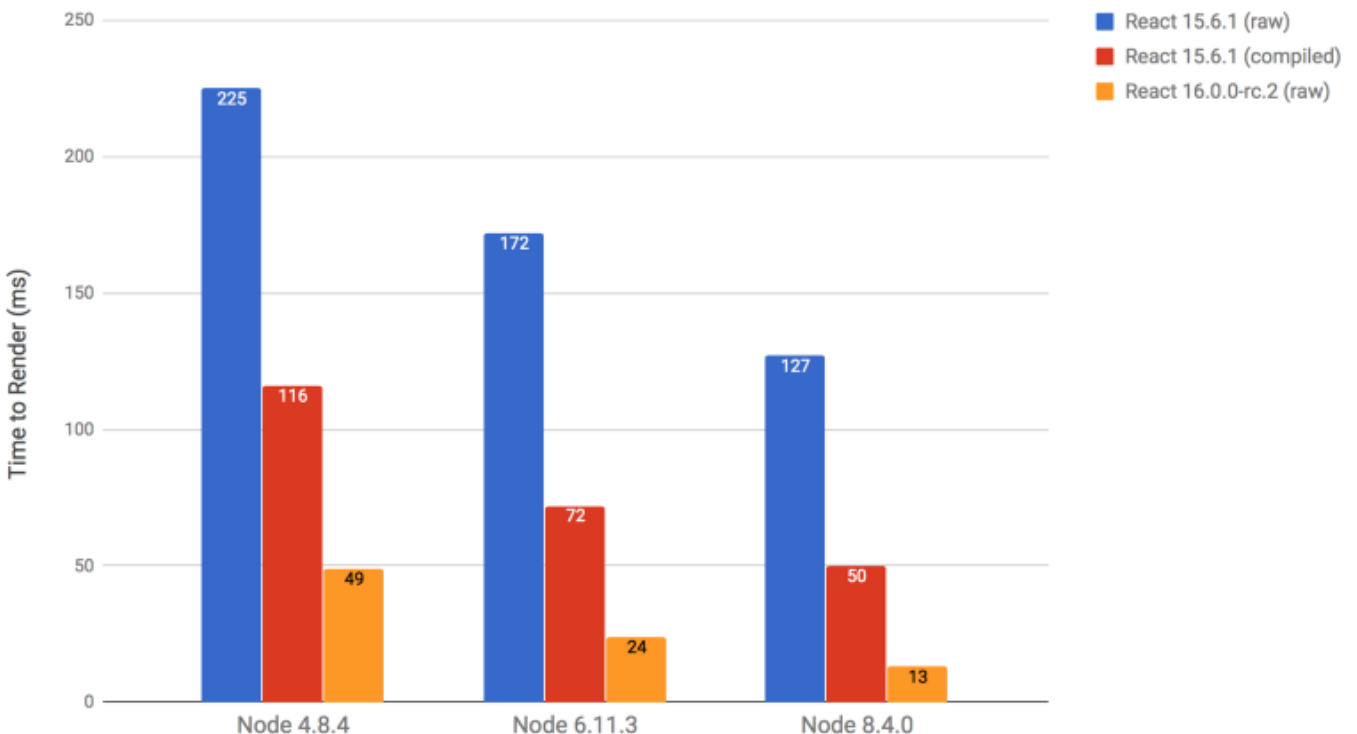
В React 16 эта проблема решена. Тут имеется лишь один вызов для проверки `process.env.NODE_ENV` в самом начале кода React 16, в итоге компилировать SSR-код для улучшения производительности больше не нужно. Сразу после установки, без дополнительных манипуляций, мы получаем отличную производительность.

React 16 отличается более высокой производительностью

Если продолжить разговор о производительности, можно сказать, что те, кто использовал серверный рендеринг React в продакшне, часто жаловались на то, что большие документы обрабатываются медленно, даже с применением всех рекомендаций по улучшению производительности. Тут хочется отметить, что рекомендуется всегда проверять, чтобы переменная `NODE_ENV` была установлена в значение `production`, когда вы используете SSR в продакшне.

С удовольствием сообщаю, что, проведя кое-какие [предварительные тесты](#), я обнаружил значительное увеличение производительности серверного рендеринга React 16 на различных версиях Node:

React 16 renders on the server faster than React 15 (smaller is better)



Рендеринг на сервере в React 16 быстрее, чем в React 15. Чем ниже столбик — тем результат лучше

При сравнении с React 16, даже с учётом того, что в React 15 обращения к `process.env` были устранены благодаря компиляции, наблюдается рост производительности примерно в 2.4 раза в Node 4, в 3 раза — в Node 6, и замечательный рост в 3.8 раза в Node 8.4. Если сравнить React 16 и React 15 без компиляции последнего, результаты на последней версии Node будут просто потрясающими.

Почему React 16 настолько быстрее, чем React 15? Итак, в React 15 серверные и клиентские подсистемы рендеринга представляли собой, в общих чертах, один и тот же код. Это означает потребность в поддержке виртуального DOM во время серверного рендеринга, даже учитывая то, что этот vDOM отбрасывался как только осуществлялся возврат из вызова `renderToString`. В результате, на сервере проводилось много ненужной работы.

В React 16, однако, команда разработчиков переписала серверный рендеринг с нуля, и теперь он совершенно не зависит от vDOM. Это и даёт значительный рост производительности.

Тут хотелось бы сделать одно предупреждение, касающееся ожидаемого роста производительности реальных проектов после перехода на React 16. Проведённые мной тесты заключались в создании огромного дерева из `` с одним очень простым рекурсивным компонентом React. Это означает, что мой бенчмарк относится к разряду синтетических и почти наверняка не отражает

сценарии реального использования React. Если в ваших компонентах имеется множество сложных методов `render`, обработка которых занимает много циклов процессора, React 16 ничего не сможет сделать для того, чтобы их ускорить. Поэтому, хотя я и ожидаю увидеть ускорение серверного рендеринга при переходе на React 16, я не жду, скажем, трёхкратного роста производительности в реальных приложениях. По непроверенным данным, при использовании React 16 в реальном проекте, удалось достичь **роста производительности примерно в 1.3 раза**. Лучший способ понять, как React 16 отразится на производительности вашего приложения — попробовать его самостоятельно.

React 16 поддерживает потоковую передачу данных

Последняя из новых возможностей React, о которой хочу рассказать, не менее интересна, чем остальные. Это — рендеринг непосредственно в потоки Node.

Потоковый рендеринг может уменьшить время до получения первого байта (TTFB, Time To First Byte). Начало документа попадает в браузер ещё до создания продолжения документа. В результате, все ведущие браузеры быстрее приступят к разбору и рендерингу документа.

Ещё одна отличная вещь, которую может получить от рендеринга в поток — это возможность реагировать на ситуацию, когда сервер выдаёт данные быстрее, чем сеть может их принять. На практике это означает, что если сеть перегружена и не может принимать данные, система рендеринга получит соответствующий сигнал и приостановит обработку данных до тех пор, пока нагрузка на сеть

не упадёт. В результате окажется, что сервер будет использовать меньше памяти и сможет быстрее реагировать на события ввода-вывода. И то и другое способно помочь серверу нормально работать в сложных условиях.

Для того, чтобы организовать потоковый рендеринг, нужно вызвать один из двух новых методов `react-dom/server`:

`renderToNodeStream` или `renderToStaticNodeStream`, которые соответствуют методам `renderToString` и `renderToStaticMarkup`. Вместо возврата строки эти методы возвращают объект [Readable](#). Такие объекты используются в модели работы с потоками Node для сущностей, генерирующих данные.

Когда вы получаете поток `Readable` из методов

`renderToNodeStream` или `renderToStaticNodeStream`, он находится в режиме приостановки, то есть, рендеринг в этот момент ещё не начинался. Рендеринг начнётся только в том случае, если вызвать [read](#), или, более вероятно, подключить поток `Readable` с помощью [pipe](#) к потоку [Writable](#). Большинство веб-фреймворков Node имеют объект ответа, который унаследован от `Writable`, поэтому обычно можно просто перенаправить `Readable` в ответ.

Скажем, вышеприведённый пример с Express можно было бы переписать для потокового рендеринга следующим образом:

```
// используем Express
import { renderToNodeStream } from "react-dom/server"
```

```
import MyPage from './MyPage'
app.get('/', (req, res) => {
  res.write("<!DOCTYPE html><html><head><title>My Page</title>
</head><body>");
  res.write("<div id='content'>");
  const stream = renderToNodeStream(<MyPage/>);
  stream.pipe(res, { end: false });
  stream.on('end', () => {
    res.write("</div></body></html>");
    res.end();
  });
});
```

Обратите внимание на то, что когда мы перенаправляем поток в объект ответа, нам необходимо использовать необязательный аргумент `{ end: false }` для того, чтобы сообщить потоку о том, что он не должен автоматически завершать ответ при завершении рендеринга. Это позволяет нам закончить оформление тела HTML-документа, и, как только поток будет полностью записан в ответ, завершить ответ самостоятельно.

Подводные камни потокового рендеринга

Потоковый рендеринг способен улучшить многие сценарии SSR, однако, существуют некоторые шаблоны, которым потоковая передача данных на пользу не пойдёт.

В целом, любой шаблон, в котором на основе разметки, созданной в ходе серверного рендеринга, формируются данные, которые надо добавить в документ до этой разметки, окажется фундаментально несовместимым с потоковой передачей данных. Среди примеров подобного — фреймворки, которые динамически определяют, какие CSS-правила надо добавить на страницу в

предшествующем сгенерированной разметке теге `<style>`, или фреймворки, которые добавляют элементы в тег `<head>` документа в процессе рендеринга тела документа. Если вы используете подобные фреймворки, вам, вероятно, придётся применять обычный рендеринг.

Ещё один шаблон, который ещё не работает в React 16 — это встроенные вызовы `renderToNodeStream` в деревьях компонента. Обычное дело в React 15 — использовать `renderToStaticMarkup` для создания шаблона страницы и встраивать вызовы `renderToString` для формирования динамического содержимого. Например, это может выглядеть так:

```
res.write("<!DOCTYPE html>");
res.write(renderToStaticMarkup(
  <html>
    <head>
      <title>My Page</title>
    </head>
    <body>
      <div id="content">
        { renderToString(<MyPage/>) }
      </div>
    </body>
  </html>));
```

Однако, если заменить эти вызовы подсистемы рендеринга на их потоковые аналоги, код перестанет работать. Потки `Readable` (которые возвращаются из `renderToNodeStream`) пока невозможно встраивать в компоненты как элементы. Надеюсь, такая возможность ещё будет добавлена в React.

Итак, выше мы рассмотрели основные новшества серверного рендеринга в React 16. Надеюсь, вам они понравились так же, как и мне. В заключение хочу сказать огромное спасибо всем, кто участвовал в разработке React 16.

Продолжаете читать? Вообще-то, пора бы уже с этим завязывать и попробовать что-нибудь отрендерить.

Уважаемые читатели! Вы ещё здесь? Похоже, серверный рендеринг в React 16 вы уже испытали. Если так — просим поделиться впечатлениями.

Проголосовать:



+33



Поделиться:



Сохранить:



Комментарии (18)

Похожие публикации

ПЕРЕВОД

ru_vds • 10 октября 2017 в 14:58

Добротный риалтайм на React и Socket.io

ПЕРЕВОД

ru_vds • 18 июля 2017 в 16:12

8

React или Vue? Выбираем библиотеку для фронтенд-разработки

ПЕРЕВОД

ru_vds • 3 марта 2017 в 13:59

104

Популярное за сутки

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

BarakAdama • вчера в 10:52

69

Почему следует игнорировать истории основателей успешных стартапов

ПЕРЕВОД

m1rko • вчера в 10:44

20

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

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

24

Java и Project Reactor

zealot_and_frenzy • вчера в 10:56

10

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle

erogov • вчера в 12:46

6

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке

NAGru • вчера в 10:10

31

Энтузиаст сделал новую материнскую плату для ThinkPad X200s

alizar • вчера в 15:32

49

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

Pochtoycom • вчера в 13:06

85

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

marks • вчера в 14:19

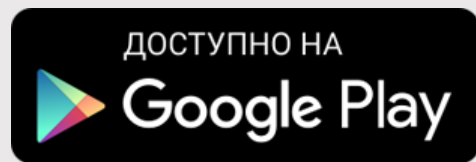
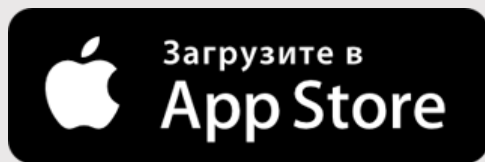
140

Дела шпионские (часть 1)

TashaFridrih • вчера в 13:16

16

Мобильное приложение



Полная версия

2006 – 2018 © TM