服务器端渲染（1）：基本套路

这一节，我们来介绍如何利用 React 做服务器端渲染，我们会先了解一下服务器端渲染的作用，然后在下一节我们会看一看业界公认最好的服务器端渲染框架 next.js 是怎么做的。

服务器端渲染

React 的功能就是把数据转化成用户界面，我们还是要祭出这个公式：

```
UI = f(data)
```

既然对 React 而言，“吃”进去的是 data，“吐”出来的是 UI，那么，这个转化过程在浏览器端可以做，当然在服务器端也可以做，不同的是浏览端的效果是直接操作 DOM，服务器端没有 DOM 可言，所以是直接吐出 HTML 字符串。

为什么要服务器端渲染

最近几年浏览器端框架很繁荣，以至于很多新入行的开发者只知道浏览器端渲染框架，都不知道存在服务器端渲染这回事，其实，网站应用最初全都是服务器端渲染，由服务器端用 PHP、Java 或者 Python 等其他语言产生 HTML 来给浏览器端解析。

相比于浏览器端渲染，服务器端渲染的好处是：

1. 可以缩短“第一有意义渲染时间”（First-Meaningful-Paint-Time）。

如果完全依赖于浏览器端渲染，那么服务器端返回的 HTML 就是一个空荡荡的框架和对 JavaScript 的应用，然后浏览器下载 JavaScript，再根据 JavaScript 中的 AJAX 调用获取服务器端数据，再渲染出 DOM 来填充网页内容，总共需要三个 HTTP 或 HTTPS 请求。

如果使用服务器端渲染，第一个 HTTP/HTTPS 请求返回的 HTML 里就包含可以渲染的内容了，这样用户第一时间就会感觉到“有东西画出来了”，这样的感知性能更好。

2. 更好的搜索引擎优化（Search-Engine-Optimization，SEO）。

大部分网站都希望自己能够出现在搜索引擎的搜索页前列，这个前提就是网页内容要能够被搜索引擎的爬虫正确抓取到。虽然 Google 这样的搜索引擎已经可以检索浏览器端渲染的网页，但毕竟不是全部搜索引擎都能做到，如果搜索引擎的爬虫只能拿到服务器端渲染的内容，完全浏览器端渲染就行不通了。

即使对于 Google，网页性能也是搜索排名的重要指标，如果通过服务器端渲染提高网页性能，网页的排名更可能靠前。

上面两点，就是服务器端渲染的主要意义。

React 对服务器端渲染的支持

因为 React 是声明式框架，所以，在渲染上对服务器端渲染非常友好。

假设我们我们要渲染一个以 `App` 为最根节点的组件树，浏览器端渲染的代码如下：

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(<App />, document.getElementById('root'));
```

现在我们想要在服务器端渲染 `App`，如果使用 React v16 之前的版本，代码是这样：

```
import React from 'react';
import ReactDOMServer from 'react-dom/server';

// 把产生html返回给浏览器端
const html = ReactDOMServer.renderToString(<Hello />);
```

从 React v16 开始，上面的服务器端代码依然可以使用，但是也可以把 `renderToString` 替换为 `renderToNodeStream`，代码如下：

```
import React from 'react';
import ReactDOMServer from 'react-dom/server';

// 把渲染内容以流的形式塞给response
ReactDOMServer.renderToNodeStream(<Hello />).pipe(response);
```

此外，浏览器端代码也有一点变化，`ReactDOM.render` 依然可以使用，但是官方建议替换为 `ReactDOM.hydrate`，原来的 `ReactDOM.render` 将来会被废弃掉。

`renderToString` 的功能是一口气同步产生最终 HTML，如果 React 组件树很庞大，这样一个同步过程可能比较耗时。假设渲染完整 HTML 需要 500 毫秒，那么一个 HTTP/HTTPS 请求过来，500 毫秒之后才返回 HTML，显得不大合适，这也是为什么 v16 提供了 `renderToNodeStream` 这个新 API 的原因。

`renderToNodeStream` 把渲染结果以“流”的形式塞给 response 对象（这里的 response 是 express 或者 koa 的概念），这意味着不用等到所有 HTML 都渲染出来了才给浏览器端返回结果，也许 10 毫秒内就渲染出来了网页头部，那就没必要等到 500 毫秒全部网页都出来了才推给浏览器，“流”的作用就是有多少内容给多少内容，这样用户只需要 10 毫秒多一点的延迟就可以看到网页内容，进一步改进了“第一有意义渲染时间”。

服务器端渲染的难点

看到这里，你可能觉得服务器端渲染也太简单了，的确，因为 React 组件可以不必关心自己是在哪个端渲染，可以做到代码一次编写，到处都可以执行。但是，真的这么简单吗？

为了简化问题，上面的代码示例有意忽略了一个事实，那就是，应用往往需要外部服务器获取数据啊！

除非你的网页应用根本没有动态内容，不然你必须要考虑在服务器端怎么给 React 组件获取数据。

比如，你现在看到的掘金小册，为了渲染你所看到的页面，需要调用掘金小册的服务器 API 来获取这篇文章的内容。对于浏览器端渲染，在 `componentDidMount` 里调用 AJAX 就好了；对于服务器端渲染，要想产生 HTML 的包含内容，必须事先把数据准备好，也就是说，代码要是这样才行：

```
import React from 'react';
import ReactDOMServer from 'react-dom/server';

callAPI().then(result => {
  const props = result;
  ReactDOMServer.renderToNodeStream(<Hello {...props}/>).pipe(response);
});
```

最大的问题来了，如何给组件获取和提供数据呢？

解决了这个问题，才算真的解决了服务器端渲染的问题。

“脱水”和“注水”

React 有一个特点，就是把内容展示和动态功能集中在一个组件中。比如，一个 Counter 组件既负责怎么画出内容，也要负责怎么响应按键点击，这当然符合软件高内聚性的原则，但是也给服务器端渲染带来更多的工作。

设想一下，如果只使用服务器端渲染，那么产生的只有 HTML，虽然能够让浏览器端画出内容，但是，没有 JavaScript 的辅助是无法响应用户交互事件的。对应 Counter 的例子，一个 Counter 组件在浏览器中也就渲染出一个数字两个按钮，用户点击 `+` 按钮或者 `-` 按钮，什么都不会发生。

很显然我们必须要在浏览器端赋予 Counter 组件一些“神力”，让它能够响应事件。那么怎么赋予 Counter 组件“神力”呢？其实我们已经做过这件事了，Counter 组件里面已经有对按钮事件的处理，我们所要做的只是让 Counter 组件在浏览器端重新执行一遍，也就是 mount 一遍就可以了。

也就是说，**如果想要动态交互效果，使用 React 服务器端渲染，必须也配合使用浏览器端渲染。**

现在问题变得更加有趣了，在服务器端我们给 Counter 一个初始值（这个值可以不是缺省的 0），让 Counter 渲染产生 HTML，这些 HTML 要传递给浏览器端，为了让 Counter 的 HTML“活”起来点击相应事件，必须要在浏览器端重新渲染一遍 Counter 组件。在浏览器端渲染 Counter 之前，用户就可以看见 Counter 组件的内容，但是无法点击交互，要想点击交互，就必须要等到浏览器端也渲染一次 Counter 之后。

接下来的一个问题，如果服务器端塞给 Counter 的数据和浏览器端塞给 Counter 的数据不一样呢？

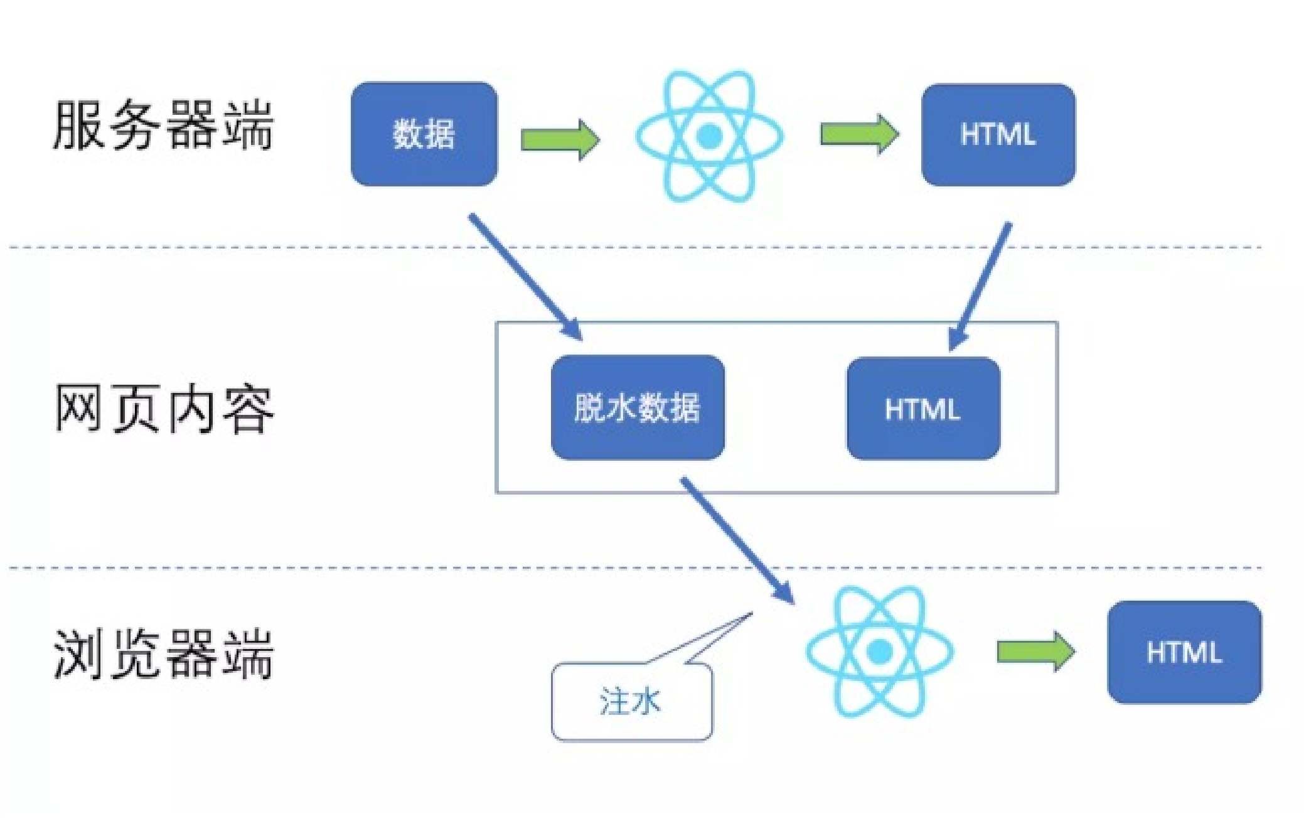
在 React v16 之前，React 在浏览器端渲染之后，会把内容和服务器端给的 HTML 做一个比对。如果完全一样，那最好，接着用服务器端 HTML 就好了；如果有一丁点不一样，就会立刻丢掉服务器端的 HTML，重新渲染浏览器端产生的内容，结果就是用户可以看到界面闪烁。因为 React 抛弃的是整个服务器端渲染内容，组件树越大，这个闪烁效果越明显。

React 在 v16 之后，做了一些改进，不再要求整个组件树两端渲染结果分毫不差，但是如果发生不一致，依然会抛弃局部服务器端渲染结果。

总之，**如果用服务器端渲染，一定要让服务器端塞给 React 组件的数据和浏览器端一致。**

为了达到这一目的，必须把传给 React 组件的数据给保留住，随着 HTML 一起传递给浏览器网页，这个过程，叫做“脱水”（Dehydrate）；在浏览器端，就直接拿这个“脱水”数据来初始化 React 组件，这个过程叫“注水”（Hydrate）。

前面提到过 React v16 之后用 `React.hydrate` 替换 `React.render`，这个 `hydrate` 就是“注水”。



总之，为了实现React的服务器端渲染，必须要处理好这两个问题：

1. 脱水
2. 注水

Facebook 未使用服务器端渲染

值得一提的是，虽然 React 从最初版本就支持“服务器端渲染”，并且 React 的创建者 Facebook 也全力在自己的网站产品中使用 React，但他们自己却没有使用 React 的服务器端渲染功能。理由是，Facebook 已经在 PHP 上投入了很多资源，不打算放弃这些投入。

这里我当然不是批评 Facebook，实际上，Facebook 对 React 的支持是真心的，它在自己的网站上大范围使用 React，而不只是做出来后让外部使用者当小白鼠，这种全力投入也给了 React 使用者很大信心。但另一方面，因为 Facebook 自己不用 React 的服务器端渲染，如何利用这个功能，就缺乏一个官方参考标准了。

也许就是因为缺乏 Facebook 的官方标准，业界对服务器端渲染的解决方法层出不穷，不过，到目前看来，`next.js` 还是最佳方案。

小结

本小节介绍了 React 的服务器端渲染功能，读者应该理解下列要点：

1. 服务器端渲染的作用；
2. React 的服务器端渲染支持方法；
3. 什么叫“脱水”和“注水”。

留言

评论将在后台进行审核，审核通过后对所有人可见

zhangyanling77 前端开发 @ 成都
那么服务端渲染之后，前端构建工具这块的插件使用和一些优化岂不是就要牺牲掉了
▲ 0　收起评论　1月前

ITSheng
服务端渲染的代码也是经过各种构建工具优化过的
29天前

评论审核通过后显示　评论