服务端渲染

服务器端呈现的最常见用例是在用户（或搜索引擎爬虫）首次请求您的应用时处理初次渲染。

当服务器收到请求时，它会将所需的组件呈现为 HTML 字符串，然后将其作为响应发送给客户端。 从那时起，客户端将接管渲染的职责。

在服务器端的 Material-UI

MUI was designed from the ground-up with the constraint of rendering on the server, but it's up to you to make sure it's correctly integrated. 为页面提供所需的 CSS 是至关重要的，否则页面只会渲染 HTML 而等待客户端注入 CSS，从而导致浏览器样式闪烁（FOUC）。 It's important to provide the page with the required CSS, otherwise the page will render with just the HTML then wait for the CSS to be injected by the client, causing it to flicker (FOUC). 若想将样式注入客户端，我们需要：

1. Create a fresh, new `emotion cache` instance on every request.
2. 用服务端收集器渲染 React 树组件。
3. 将 CSS 单独拿出。
4. 将 CSS 传递给客户端。

On the client-side, the CSS will be injected a second time before removing the server-side injected CSS.

Setting up

在下面的配置中，我们将了解如何设置服务器端的渲染。

主题

创建一个在客户端和服务端之间共享的主题：

`theme.js`

```
import { createTheme } from '@mui/material/styles';
import { red } from '@mui/material/colors';

// Create a theme instance.
const theme = createTheme({
  palette: {
    primary: {
      main: '#556cd6',
    },
    secondary: {
      main: '#19857b',
    },
    error: {
      main: red.A400,
    },
  },
});

export default theme;
import { createTheme } from '@material-ui/core/styles';
```

```
import red from '@material-ui/core/colors/red';

// 创建一个主题的实例。
```

服务器端

下面的大纲可以大致展现一下服务器端。 We are going to set up an Express middleware using app.use to handle all requests that come into the server. If you're unfamiliar with Express or middleware, know that the `handleRender` function will be called every time the server receives a request. If you're unfamiliar with Express or middleware, know that the `handleRender` function will be called every time the server receives a request. If you're unfamiliar with Express or middleware, know that the `handleRender` function will be called every time the server receives a request.

`server.js`

```
import express from 'express';

// 我们将在章节中填写这些需要遵守的内容。
function renderFullPage(html, css) {
  /* ... */
}

function handleRender(req, res) {
  /* ... */
}

const app = express();

// 每当服务器端接收到一个请求时，这个功能就会被触发。
app.use(handleRender);

const port = 3000;
app.listen(port);
```

Handling the request

The first thing that we need to do on every request is to create a new `emotion cache`.

When rendering, we will wrap `App`, the root component, inside a `CacheProvider` and `ThemeProvider` to make the style configuration and the `theme` available to all components in the component tree.

The key step in server-side rendering is to render the initial HTML of the component **before** we send it to the client-side. 我们用 ReactDOMServer.renderToString() 来实现此操作。 我们用 ReactDOMServer.renderToString() 来实现此操作。

MUI is using emotion as its default styled engine. We need to extract the styles from the emotion instance. The client-side is straightforward. All we need to do is use the same cache configuration as the server-side. 让我们来看看客户端的文件： We need to extract the styles from the emotion instance. The client-side is straightforward. All we need to do is use the same cache configuration as the server-side. 让我们来看看客户端的文件：

`getCache.js`

```
import createCache from '@emotion/cache';

export default function getCache() {
  const cache = createCache({ key: 'css' });
  cache.compat = true;
  return cache;
}
```

With this we are creating new emotion cache instance and using this to extract the critical styles for the html as well.

我们将看到在 `renderFullPage` 函数中，是如何传递这些信息的。

```
import express from 'express';
import * as React from 'react';
import ReactDOMServer from 'react-dom/server';
import CssBaseline from '@mui/material/CssBaseline';
import { ThemeProvider } from '@mui/material/styles';
import { CacheProvider } from '@emotion/react';
import createEmotionServer from '@emotion/server/create-instance';
import App from './App';
import theme from './theme';
import createEmotionCache from './createEmotionCache';

function handleRender(req, res) {
  const cache = createEmotionCache();
  const { extractCriticalToChunks, constructStyleTagsFromChunks } =
    createEmotionServer(cache);

  // Render the component to a string.
  const html = ReactDOMServer.renderToString(
    <CacheProvider value={cache}>
      <ThemeProvider theme={theme}>
        {/* CssBaseline kickstart an elegant, consistent, and simple baseline to
build upon. */}
        <CssBaseline />
        <App />
      </ThemeProvider>
    </CacheProvider>,
  );

  // Grab the CSS from emotion
  const emotionChunks = extractCriticalToChunks(html);
  const emotionCss = constructStyleTagsFromChunks(emotionChunks);

  // Send the rendered page back to the client.
  res.send(renderFullPage(html, emotionCss));
}

const app = express();

app.use('/build', express.static('build'));
```

```
// This is fired every time the server-side receives a request.
app.use(handleRender);

const port = 3000;
app.listen(port); import express from 'express';
import * as React from 'react';
import ReactDOMServer from 'react-dom/server';
import CssBaseline from '@material-ui/core/CssBaseline';
import { ThemeProvider } from '@material-ui/core/styles';
import createEmotionServer from '@emotion/server/create-instance';
import App from './App';
import theme from './theme';
import getCache from './getCache';

function handleRender(req, res) {
  const cache = getCache();
  const { extractCriticalToChunks, constructStyleTagsFromChunks } =
    createEmotionServer(cache);

  // 将组件渲染成字符串。 */}
        <CssBaseline />
        <App />
      </ThemeProvider>
    </CacheProvider>,
  );

  // Grab the CSS from emotion
  const emotionChunks = extractCriticalToChunks(html);
  const emotionCss = constructStyleTagsFromChunks(emotionChunks);

  // Send the rendered page back to the client.
  res.send(renderFullPage(html, emotionCss));
}

const app = express();

app.use('/build', express.static('build'));

// This is fired every time the server-side receives a request.
app.use(handleRender);

const port = 3000;
app.listen(port);
```

Inject initial component HTML and CSS

The final step on the server-side is to inject the initial component HTML and CSS into a template to be rendered on the client-side.

```
function renderFullPage(html, css) {
  return `
    <!DOCTYPE html>
    <html>
      <head>
        <title>My page</title>
        ${css}
        <meta name="viewport" content="initial-scale=1, width=device-width" />
        <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Roboto:300,400,500,700&display=swap" />
      </head>
      <body>
        <div id="root">${html}</div>
      </body>
    </html>
  `;
}
```

The client-side

The client-side is straightforward. All we need to do is use the same cache configuration as the server-side. Let's take a look at the client file:

`client.js`

```
import * as React from 'react';
import ReactDOM from 'react-dom';
import CssBaseline from '@material-ui/core/CssBaseline';
import { ThemeProvider } from '@material-ui/core/styles';
import { CacheProvider } from '@emotion/react';
import App from './App';
import theme from './theme';
import getCache from './getCache';

function Main() {
  return (
    <CacheProvider value={getCache}>
      <ThemeProvider theme={theme}>
        {/* CssBaseline kickstart an elegant, consistent, and simple baseline to
build upon. */}
        <CssBaseline />
        <App />
      </ThemeProvider>
    </CacheProvider>
  );
}

ReactDOM.hydrate(<Main />, document.querySelector('#root'));
  const html = ReactDOMServer.renderToString(
    <CacheProvider value={cache}>
      <ThemeProvider theme={theme}>
```

```
        {/* CssBaseline kickstart an elegant, consistent, and simple baseline to
build upon. */}
        <CssBaseline />
        <App />
      </ThemeProvider>
    </CacheProvider>,
  );

  // Grab the CSS from emotion
  const emotionChunks = extractCriticalToChunks(html);
  const emotionCss = constructStyleTagsFromChunks(emotionChunks);

  // Send the rendered page back to the client.
  res.send(renderFullPage(html, emotionCss));
}

const app = express();

app.use('/build', express.static('build'));

// This is fired every time the server-side receives a request.
app.use(handleRender);

const port = 3000;
app.listen(port); */}
        <CssBaseline />
        <App />
      </ThemeProvider>
    </CacheProvider>
  );
}

ReactDOM.hydrate(<Main />, document.querySelector('#root'));
```

参考实现

We host different reference implementations which you can find in the GitHub repository under the `/examples` folder:

- 本教程的参考实现
- Gatsby
- Next.js (TypeScript version)

故障排除（Troubleshooting）

查看常见问题解答：我的应用程序在服务端上不能正确渲染。