

Advanced 进阶

本节包含了 `@material-ui/core/styles` 的一些更多的进阶用法。

Note: `@mui/styles` is the **legacy** styling solution for MUI. It is deprecated in v5. It depends on [JSS](#) as a styling solution, which is not used in the `@mui/material` anymore. If you don't want to have both emotion & JSS in your bundle, please refer to the [@mui/system](#) documentation which is the recommended alternative. It is deprecated in v5. It depends on [JSS](#) as a styling solution, which is not used in the `@mui/material` anymore. If you don't want to have both emotion & JSS in your bundle, please refer to the [@mui/system](#) documentation which is the recommended alternative.

Theming 主题

若您想将主题传递到 React 组件树，请将添加 `ThemeProvider` 包装到应用程序的顶层。然后，您可以在样式函数中访问主题对象。

此示例为自定义组件创建了一个主题对象 (theme object)。If you intend to use some of the Material-UI's components you need to provide a richer theme structure using the `createTheme()` method. 请前往 [theming 部分](#) 学习如何构建自己的 Material-UI 主题。Head to the [theming section](#) to learn how to build your custom MUI theme.

```
import { ThemeProvider } from '@material-ui/core/styles';
import DeepChild from './my_components/DeepChild';

const theme = {
  background: 'linear-gradient(45deg, #FE6B8B 30%, #FF8E53 90%)',
};

function Theming() {
  return (
    <ThemeProvider theme={theme}>
      <DeepChild />
    </ThemeProvider>
  );
}
```

```
{{"demo": "Theming.js"}}
```

访问一个组件中的主题

您可能需要访问 React 组件中的主题变量。

`useTheme` hook

在函数组件 (function components) 中的使用：

```
import { useTheme } from '@material-ui/core/styles';

function DeepChild() {
  const theme = useTheme();
```

```
    return <span>{`spacing ${theme.spacing}`}</span>;
  }
```

{{"demo": "UseTheme.js"}}

withTheme HOC

在类 (class) 或函数 (function) 组件中的使用:

```
import { withTheme } from '@material-ui/core/styles';

function DeepChildRaw(props) {
  return <span>{`spacing ${props.theme.spacing}`}</span>;
}

const DeepChild = withTheme(DeepChildRaw);
```

{{"demo": "WithTheme.js"}}

覆盖主题

您可以嵌套多个主题提供者。当您在处理应用程序的不同区域时, 若需要应用的不同外观, 这个功能会让您得心应手。

```
<ThemeProvider theme={outerTheme}>
  <Child1 />
  <ThemeProvider theme={innerTheme}>
    <Child2 />
  </ThemeProvider>
</ThemeProvider>
```

{{"demo": "ThemeNesting.js"}}

内部主题将 **覆盖** 外部主题。 您可以提供一个函数来扩展外层主题:

```
<ThemeProvider theme={...} <ThemeProvider theme={...} >
  <Child1 />
  <ThemeProvider theme={outerTheme => ({ darkMode: true, ...outerTheme })}>
    <Child2 />
  </ThemeProvider>
</ThemeProvider>
```

覆盖样式 — classes 属性

通过 `makeStyles` (hook generator) 和 `withStyles` (HOC) 这两个 API, 用户可以为每个样式表创建多种样式规则。每个样式规则都有自己的类名。组件的 `classes` 变量会提供类名 (class names)。这在设置组件中嵌套元素的样式时特别有用。

```
// 样式表
const useStyles = makeStyles({
```

```

    root: {}, // 样式规则
    label: {}, // 嵌套的样式规则
  });

function Nested(props) {
  const classes = useStyles();
  return (
    <button className={classes.root}>
      {/* 'jss1' */}
      <span className={classes.label}>{/* 'jss2' 嵌套规则 */}</span>
    </button>
  );
}

function Parent() {
  return <Nested />;
}

```

但是，类名通常是不确定的。父级组件如何覆盖嵌套元素的样式呢？

withStyles

这是最简单的一种情况。封装的组件接受 `classes` 属性，它简单地合并了样式表提供的类名。

```

const Nested = withStyles({
  root: {}, // a style rule
  label: {}, // a nested style rule
})(({ classes }) => (
  <button className={classes.root}>
    <span className={classes.label}>{/* 'jss2 my-label' Nested*/}</span>
  </button>
));

function Parent() {
  return <Nested classes={{ label: 'my-label' }} />;
}

```

makeStyles

想使用 hook API 的话需要一些额外的工作。你必须把父级属性作为第一个参数传递给 hook。

```

const useStyles = makeStyles({
  root: {}, // 样式规则
  label: {}, // 嵌套的样式规则
});

function Nested(props) {
  const classes = useStyles(props);
  return (
    <button className={classes.root}>
      <span className={classes.label}>{/* 'jss2 my-label' 嵌套规则 */}</span>
    </button>
  );
}

```

```

    </button>
  );
}

function Parent() {
  return <Nested classes={{ label: 'my-label' }} />;
}

```

JSS 插件

JSS 使用插件来扩展其核心，您可以挑选所需的功能，并且只需承担您正在使用的内容性能的开销。

Not all the plugins are available in MUI by default. 以下（一个 [jss-preset-default 的子集](#)）被包含在内： The following (which is a subset of [jss-preset-default](#)) are included:

- [jss-plugin-rule-value-function](#)
- [jss-plugin-global](#)
- [jss-plugin-nested](#)
- [jss-plugin-camel-case](#)
- [jss-plugin-default-unit](#)
- [jss-plugin-vendor-prefixer](#)
- [jss-plugin-props-sort](#)

当然，你也可以随意使用额外的插件。我们有一个使用 [jss-rtl](#) 插件的例子。

```

import { create } from 'jss';
import { StylesProvider, jssPreset } from '@material-ui/styles';
import rtl from 'jss-rtl';

const jss = create({
  plugins: [...jssPreset().plugins, rtl()],
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}

```

字符串模板

如果相比 JSS 您更喜欢 CSS 的语法，则可以使用 [jss-plugin-template](#) 插件。

```

const useStyles = makeStyles({
  root: `
    background: linear-gradient(45deg, #fe6b8b 30%, #ff8e53 90%);
    border-radius: 3px;
    font-size: 16px;
    border: 0;
    color: white;
    height: 48px;
    padding: 0 30px;
    box-shadow: 0 3px 5px 2px rgba(255, 105, 135, 0.3);
  `;
});

```

```
`;  
});
```

请注意，此插件不支持选择器或嵌套规则。

```
{{"demo": "StringTemplates.js"}}
```

CSS 注入顺序

了解浏览器如何计算 CSS 优先级是**非常重要的**，因为它是您在覆盖样式时需要了解的重点之一。我们推荐您阅读 MDN 上的这段内容：[如何计算优先级？](#)

默认情况下，注入的 style 标签会被插入到页面 <head> 元素的最后。它们的优先级高于您页面上的任何其他样式标签，如 CSS 模块、styled components。



injectFirst

StylesProvider 组件有一个 injectFirst 属性，用于将样式标签**首先**从页头（优先级较低）注入：

```
import { StylesProvider } from '@material-ui/styles';  
  
<StylesProvider injectFirst>  
  { /* 你的组件树。  
    Styled components can override MUI's styles. */ }  
</StylesProvider>;
```

makeStyles / withStyles / styled

使用 makeStyles / withStyles / styled 的注入顺序于调用顺序**相同**。例如，在这种情况下，字体最终是红色：

```
import clsx from 'clsx';  
import { makeStyles } from '@material-ui/styles';  
  
const useStylesBase = makeStyles({  
  root: {  
    color: 'blue', //   
  },  
});  
  
const useStyles = makeStyles({  
  root: {  
    color: 'red', //   
  },  
});  
  
export default function MyComponent() {  
  // Order doesn't matter  
  const classes = useStyles();  
  const classesBase = useStylesBase();
```

```
// Order doesn't matter
const className = clsx(classes.root, classesBase.root);

// color: 红色 📌 胜出。
return <div className={className} />;
}
```

Hook 调用顺序和类名顺序**不影响**注入属性权重。

insertionPoint

JSS [提供了一种机制](#) 来控制这种情况。By adding an `insertionPoint` within the HTML you can [control the order](#) that the CSS rules are applied to your components.

HTML 注释

最简单的方法是在 `<head>` 中添加一个 HTML 注释，来决定 JSS 注入样式的位置：

```
<head>
<!-- jss-insertion-point --> <link href="..." />
</head> />
</head>
```

```
insertionPoint: 'jss-insertion-point',
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}

import { create } from 'jss';
import { StylesProvider, jssPreset } from '@material-ui/styles';

const jss = create({
  ...jssPreset(),
  // 当将样式注入到 DOM 中时，定义了一个自定义插入点以供 JSS 查询。
  insertionPoint: 'jss-insertion-point',
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}
```

其他 HTML 元素

在构建生产环境时，[Create React App](#) 会移除 HTML 注释。为了解决这个问题，您可以提供一个 DOM 元素（而不是一条注释）作为 JSS 插入点，譬如一个 `<noscript>` 元素。

```
<head>
  <noscript id="jss-insertion-point" />
  <link href="..." />
```

```
</head> />
</head>
```

```
import { create } from 'jss';
import { StylesProvider, jssPreset } from '@mui/styles';

const jss = create({
  ...jssPreset(),
  // Define a custom insertion point that JSS will look for when injecting the
  styles into the DOM.
  insertionPoint: document.getElementById('jss-insertion-point'),
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}

import { create } from 'jss';
import { StylesProvider, jssPreset } from '@material-ui/styles';

const jss = create({
  ...jssPreset(),
  // 当将样式注入到 DOM 中时，定义了一个自定义插入点以供 JSS 查询。
});
```

JS createComment

codesandbox.io 阻止访问 `<head>` 元素。要解决这个问题，您可以使用 JavaScript 中的 `document.createComment()` API。

```
import { create } from 'jss';
import { StylesProvider, jssPreset } from '@mui/styles';

const styleNode = document.createComment('jss-insertion-point');
document.head.insertBefore(styleNode, document.head.firstChild);

const jss = create({
  ...jssPreset(),
  // Define a custom insertion point that JSS will look for when injecting the
  styles into the DOM.
  insertionPoint: 'jss-insertion-point',
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}

import { create } from 'jss';
import { StylesProvider, jssPreset } from '@material-ui/styles';

const styleNode = document.createComment('jss-insertion-point');
document.head.insertBefore(styleNode, document.head.firstChild);
```

```
const jss = create({
  ...jssPreset(),
  // 我们定义了一个自定义插入点，JSS在DOM中注入样式时会查找该插入点。
```

服务端渲染

This example returns a string of HTML and inlines the critical CSS required, right before it's used:

```
import ReactDOMServer from 'react-dom/server';
import { ServerStyleSheets } from '@material-ui/styles';

function render() {
  const sheets = new ServerStyleSheets();

  const html = ReactDOMServer.renderToString(sheets.collect(<App />));
  const css = sheets.toString();

  return `
<!DOCTYPE html>
<html>
  <head>
    <style id="jss-server-side">${css}</style>
  </head>
  <body>
    <div id="root">${html}</div>
  </body>
</html>
`;
}
```

您可以[根据这篇服务端渲染指南](#)来获取更多详细的例子，或者您也可以阅读 [ServerStyleSheets](#) 的 API 文档。

Gatsby

这个 [官方的 Gatsby 插件](#)，可以利用它来实现 `@material-ui/style` 的服务器端渲染。请参考插件页面的设置和使用说明。

Refer to [this example Gatsby project](#) for an up-to-date usage example.

Next.js

您需要有一个自定义的 `pages/_document.js`，然后复制 [此逻辑](#) 以注入服务器侧渲染的样式到 `<head>` 元素中。> Refer to [this example project](#) for an up-to-date usage example.

类名 (Class names)

类名 (class names) 由 [类名生成器](#) 生成。

默认值

默认情况下，`@material-ui/core/styles` 生成的类名 **不是固定值**；所以你不能指望它们保持不变。让我们以下面的样式 (style) 作为示例：


```
const useStyles = makeStyles({
  root: {
    opacity: 1,
  },
});
```

上述例子将生成一个类似于 `makeStyles-root-123` 这样的类名。

您必须使用组件的 `classes` 属性来覆盖样式。类名的不确定性使样式隔离成为可能。

- 在开发环境中，类名为： `.makeStyles-root-123`，它遵循以下逻辑：

```
const sheetName = 'makeStyles';
const ruleName = 'root';
const identifier = 123;

const className = `${sheetName}-${ruleName}-${identifier}`;
```

- 在生产环境中，类名为： `.jss123`，它遵循以下逻辑：

```
const productionPrefix = 'jss';
const identifier = 123;

const className = `${productionPrefix}-${identifier}`;
```

当满足以下条件时，类名是 **确定的**：

- 仅使用一个主题提供程序（**无主题嵌套**）。
- 样式表的名称以 `Mui` 开头（包含所有 Material-UI 组件）。
- [类名生成器](#)的 `disableGlobal` 选项为 `false`（默认值）。

全局 CSS

`jss-plugin-global`

[jss-plugin-global](#) 插件安装在默认的预设中。您可以使用它来定义全局类名称。

```
{{"demo": "GlobalCss.js"}}
```

混合

您也可以将 JSS 生成的类名称与全局名称结合起来。

```
{{"demo": "HybridGlobalCss.js"}}
```

CSS 前缀 (prefixes)

JSS 使用特征探测来应用正确的前缀。如果您看不到最新版本 Chrome 中显示一个特定前缀，[请不要感到惊讶](#)。您的浏览器可能不需要它。

TypeScript usage

在 TypeScript 中使用 `withStyles` 可能有点棘手，但有一些实用程序可以帮助提高使用感受。

使用 `createStyles` 来杜绝类型扩展

A frequent source of confusion is TypeScript's [type widening](#), which causes this example not to work as expected:

```
const styles = {
  root: {
    display: 'flex',
    flexDirection: 'column',
  }
};

withStyles(styles);
//      ^^^^^^
//      属性 'flexDirection' 的类型是不兼容的。
//      'string' 类型不能赋予给这些类型: '"-moz-initial" | "inherit" | "initial"
//      | "revert" | "unset" | "column" | "column-reverse" | "row"...'。
```

然而，这是不是很 [DRY](#)，因为它需要你在两个不同的地方保持类名（`'root'`，`'paper'`，`'button'`，...）。我们提供了一个类型操作符 `WithStyles` 来帮助解决这个问题，因此您可以直接写入：

```
withStyles({
  root: {
    display: 'flex',
    flexDirection: 'column',
  },
});
```

然而，如果您尝试让样式随主题而变化，类型扩展会再次显示其不怎么雅观的部分：

```
withStyles(({ palette, spacing }) => ({
  root: {
    display: 'flex',
    flexDirection: 'column',
    padding: spacing.unit,
    backgroundColor: palette.background.default,
    color: palette.primary.main,
  },
}));
```

这是因为 TypeScript [扩展了函数表达式](#) 的返回类型。

因此，我们建议使用我们的 `createStyles` 帮助函数来构造样式规则对象：

```
const useStyles = makeStyles(
{
  root: {
```

```

        /* ... */
    },
    label: {
        /* ... */
    },
    outlined: {
        /* ... */
        '&$disabled': {
            /* ... */
        },
    },
    outlinedPrimary: {
        /* ... */
        '&:hover': {
            /* ... */
        },
    },
    disabled: {},
},
{ name: 'MuiButton' },
);

```

`createStyles` 只是身份函数；它不会在运行时“做任何事情”，只是在编译时指导类型推断。

Media queries (媒体查询)

`withStyles` 允许样式对象具有顶级媒体查询的权限，如下所示：

```

const styles = createStyles({
  root: {
    minHeight: '100vh',
  },
  '@media (min-width: 960px)': {
    root: {
      display: 'flex',
    },
  },
});

```

To allow these styles to pass TypeScript however, the definitions have to be unambiguous concerning the names for CSS classes and actual CSS property names. Due to this, class names that are equal to CSS properties should be avoided. Due to this, class names that are equal to CSS properties should be avoided.

```

// 这样是错误的，因为 TypeScript 认为 `@media (min-width: 960px)` 是一个类名
// 并且认为 `content` 是 css 属性
const ambiguousStyles = createStyles({
  content: {
    minHeight: '100vh',
  },
  '@media (min-width: 960px)': {
    content: {

```

```

        display: 'flex',
      },
    },
  });

// 这样定义就可以
const ambiguousStyles = createStyles({
  contentClass: {
    minHeight: '100vh',
  },
  '@media (min-width: 960px)': {
    contentClass: {
      display: 'flex',
    },
  },
});

```

使用 `WithStyles` 来扩充你的属性

由于用 `withStyles(styles)` 装饰的组件被注入了一个特殊的 `classes` 属性，您需要相应地定义其属性：

```

const styles = (theme: Theme) =>
  createStyles({
    root: {
      /* ... */
    },
    paper: {
      /* ... */
    },
    button: {
      /* ... */
    },
  });

interface Props {
  // 未被注入样式的属性
  foo: number;
  bar: boolean;
  // 已被注入样式的属性
  classes: {
    root: string;
    paper: string;
    button: string;
  };
}

```

However this isn't very [DRY](#) because it requires you to maintain the class names (`'root'` , `'paper'` , `'button'` , ...) in two different places. We provide a type operator `WithStyles` to help with this, so that you can just write:

这是对 `@material-ui/core/Button` 组件样式表的简化。

装饰组件

组件样式表的简化。

```
const DecoratedSFC = withStyles(styles)(({ text, type, color, classes }: Props)
=> (
  <Typography variant={type} color={color} classes={classes}>
    {text}
  </Typography>
));

const DecoratedClass = withStyles(styles)(
  class extends React.Component<Props> {
    render() {
      const { text, type, color, classes } = this.props;
      return (
        <Typography variant={type} color={color} classes={classes}>
          {text}
        </Typography>
      );
    }
  },
);
```

将 `withStyles(styles)` 作为函数来如期使用: