□□□□□□□

□□

- HTML □ CSS
- □□□□
  - □ UI □□□ HTML □□
  - □□□□ API
  - □□□□ API □□□□□
- □□□□
  - □□□□
  - □□□□
    - Styles
    - Settings
  - From data to styles
    - Styles to CSS rules
    - □□□□ CSS □□
  - Global Styles API □□□□□
    - 1. □□□□□□□□ CSS □□□□□□□□□□□
    - 2. □□□□□□□□□□□□ HTML □□
    - 3. □□□□□□□□□□□
    - 4. □□□□□□□□□□□□□□□□□□□□□
- □□□□
  - □□□□□□
  - □□□□□□
  - □□□□□□□
  - □□□□□□□□□□□□□
    - □□□□□
  - □□□□□□□□□□□

□□□
□□□□□□

HTML 与 CSS

用户通过区块编辑器创建文章时，会生成多种产物：一份 HTML 文档以及若干 CSS 样式表，这些样式表可能内嵌于文档中，也可能是外部文件。

最终的 HTML 文档由以下几部分构成：

- 主题提供的 WordPress 模板，可通过 PHP（经典主题）或 HTML 模板（区块主题）实现（详细了解两者差异）
- 使用的区块和模式，它们带有预定义的结构（HTML 标记）
- 用户对内容的修改：添加内容、转换现有内容（将段落转换为标题）或修改内容（为区块添加类或内联样式）

前端加载的样式表包括：

- 区块样式：区块自带的样式表。在前端，您可能会看到包含 WordPress 定义的所有区块样式的单一样式表（`wp-block-library-*`），也可能看到每个使用中区块的独立样式表（如 `wp-block-group-*`、`wp-block-columns-*` 等）。完整细节请参阅此说明。
- 全局样式：这些样式通过 theme.json 文件的数据动态生成：请参阅说明、参考文档和操作指南。具体而言，它会合并来自 WordPress 的 theme.json、主题的 theme.json（如果存在）以及用户通过站点编辑器的全局样式侧边栏提供的数据。处理这些数据的结果是一个内嵌样式表，其 ID 为 `global-styles-inline-css`。
- 主题样式：传统上，主题会加载自己的样式表，其 ID 基于主题名称，例如 `twentytwentytwo-style-css`。除了拥有自己的样式表外，主题现在还可以声明一个 theme.json 文件，其中包含的样式将成为全局样式生成的样式表的一部分。
- 用户样式：用户在编辑器中的某些操作会生成样式内容，例如双色调、布局或链接颜色等功能。
- 其他样式：WordPress 和插件也可以加载样式表。

区块样式

自 WordPress 5.0 引入区块编辑器以来，一直为用户提供为特定区块"添加样式"的工具。通过使用这些工具，用户可以为区块附加新的类或内联样式，从而修改其视觉外观。

默认情况下，区块具有给定的 HTML 标记。以段落区块为例：

```
<p></p>
```

在最简单的形式中，任何针对 `p` 选择器的样式规则都会应用于此区块，无论这些规则来自区块、主题等。

用户可以通过应用不同的样式来更改此区块的状态：文本对齐方式、颜色、字体大小、行高等。这些状态通过 HTML 属性（主要是 `class` 或 `style` 属性，也可以是区块作者认为合适的任何其他属性）反映在区块的 HTML 标记中。

用户对区块进行一些修改后，初始标记可能会变成这样：

```
<p class="has-color has-green-color has-font-size has-small-font-size my-custom-class"
style="line-height: 1em"></p>
```

这就是我们所说的"用户提供的区块样式"，也称为"局部样式"或"序列化样式"。本质上，每个工具（字体大小、颜色等）最终都会向区块标记添加一些类和/或内联样式。这些类的 CSS 样式是区块、全局或主题样式表的一部分。

修改区块状态的能力，加上区块可以存在于任何其他区块内（例如组内的段落），创造了大量潜在的状态和样式可能性。

从 UI 控件到 HTML 标记

如果您遵循区块教程，可以详细了解区块 API 的各个部分，并构建自己的区块。本文介绍区块如何让用户编辑其状态的基本概念。

要构建上述体验，区块作者需要以下几个部分：

1. **UI** 控件。它向用户呈现一些选择，例如能够更改区块的字体大小。控件负责从区块读取数据（此区块是否已分配字体大小？）及其所需的其他数据（用户在此区块中可以使用哪些字体大小？）。请参阅可用的组件库。
2. 区块属性。区块需要保存数据以了解对其应用了哪些修改：例如是否已为其指定字体大小。请参阅区块如何定义属性。
3. 访问样式数据。控件可能需要有关给定区块可用样式的外部信息：例如颜色列表或字体大小列表。这些被称为“样式预设”，因为它们通常是主题定义的样式预选，尽管 WordPress 提供了一些默认值。请查看主题可以提供给编辑器的数据列表，以及区块作者如何通过 useSetting 访问它。
4. 将用户样式序列化为 **HTML** 标记。在用户操作后，区块 HTML 标记需要相应更新（应用适当的类或内联样式）。此过程称为序列化，它是 edit、save 和 render_callback 函数的职责：这些函数获取区块数据并将其转换为 HTML。

本质上，这些是区块作者需要关心的基本机制，以便用户能够对其区块进行样式设置。虽然这可以完全手动完成，但有一个 API 可以自动化常见样式需求的过程：区块支持。

区块支持 API

区块支持 是一个允许区块声明其支持哪些功能的 API。通过在 block.json 文件 中添加一些信息，区块可以告知系统用户能对其执行哪些操作。

例如：

```
{
  "name": "core/paragraph",
  "...": "...",
  "supports": {
    "typography": {
      "fontSize": true
    }
  }
}
```

段落区块在其 `block.json` 中声明支持字体大小。这意味着该区块将显示一个 UI 控件供用户调整其字体大小，除非主题禁用了此功能（有关主题如何禁用 UI 控件的更多信息，请参阅 `theme.json` 参考文档）。系统还将负责设置 UI 控件数据（例如，如果区块已分配了字体大小，则显示该字体大小；显示可用字体大小列表），并在用户更改时将区块数据序列化为 HTML 标记（适当地附加类和内联样式）。

通过使用 `block.json` 中的区块支持机制，区块作者只需编写几行代码就能创建与以往相同的体验。请查看 区块支持 API 以了解如何为静态或动态区块添加区块支持。

除了用更少的工作量实现相同结果的好处外，还有其他一些优势：

- 区块的样式信息可供原生移动应用程序和服务器使用
- 区块将使用其他区块用于相同样式的 UI 控件，从而创建更一致的用户体验
- 区块使用的 UI 控件将随着其改进而自动更新，区块作者无需执行任何操作

区块支持 API 的当前限制

虽然区块支持 API 提供了价值，但也存在一些区块作者需要注意的限制。为了更好地理解这些限制，让我们以表格区块为例：

```
<table>
  <thead>
    <tr>
      <th>Header</th>
```

```
□□</tr>
□</thead>
□<tbody>
□□<tr>
□□□<th>First</th>
□□</tr>
□□<tr>
□□□<th>Second</th>
□□</tr>
□</tbody>
□<tfoot>
□□<tr>
□□□<th>Footer</th>
□□</tr>
□</tfoot>
</table>
```

1. 每个区块只能使用一种样式类型。

其中一个限制是，在所有 可用样式 中，区块只能使用其中任何一种的一个实例。以上述示例为例，表格区块只能有一个字体大小。如果区块作者希望拥有三种不同的字体大小（表头、主体和页脚），则无法使用当前的区块支持 API 实现。有关更详细的信息和前进方向，请参阅 此问题 。

2. 样式被序列化到区块最外层的 **HTML** 节点，即包装器。

区块支持 API 仅将字体大小值序列化到包装器，导致生成以下 HTML：`<table class="has-small-font-size">`。当前的区块支持 API 不会将此值序列化到其他节点，例如 `<tbody>`。

这是一个正在进行的工作领域，您可以在 跟踪问题 中关注。链接的提案正在探索一种不同的方式来序列化用户更改：不是每个区块支持序列化自己的数据（例如，像 `has-small-font-size`、`has-green-color` 这样的类），而是区块将获得一个单独的类（例如，`wp-style-UUID`），并且该类对应的 CSS 样式将由 WordPress 在服务器端生成。

虽然该提案的工作仍在继续，但有一个逃生舱口，区块作者可以使用一个实验性选项。任何区块支持都可以通过使用 `__experimentalSkipSerialization` 来跳过序列化到 HTML 标记。例如：

```
{
□"name": "core/paragraph",
□"...": "...",
□"supports": {
□□"typography": {
```

```
□□□"fontSize": true,
□□□"__experimentalSkipSerialization": true
□□}
□}
}
```

这意味着排版区块支持将执行所有操作（创建 UI 控件、将区块属性绑定到控件等），但不将用户值序列化到 HTML 标记中。类和内联样式将不会自动应用到包装器，区块作者有责任在 `edit`、`save` 和 `render_callback` 函数中实现这一点。有关如何为 WordPress 提供的一些区块完成此操作的示例，请参阅 此问题。

请注意，如果为某个组（排版、颜色、间距）启用了 `__experimentalSkipSerialization`，它将影响该组内的所有 区块支持。在上面的示例中，`typography` 组内的 所有 属性都将受到影响（例如 `fontSize`、`lineHeight`、`fontFamily` 等）。

若仅针对 单个 属性启用，可以使用数组来声明要跳过的属性。在下面的示例中，只有 `fontSize` 会跳过序列化，而 `typography` 组内的其他项目（例如 `lineHeight`、`fontFamily` 等）将不受影响。

```
{
□"name": "core/paragraph",
□"...": "...",
□"supports": {
□□"typography": {
□□□"fontSize": true,
□□□"lineHeight": true,
□□□"__experimentalSkipSerialization": [ "fontSize" ]
□□}
□}
}
```

此功能的支持已在 此 PR 中添加。

全局样式

全局样式是指生成全站样式的一种机制。与上一节描述的区块样式不同，这些样式不会被序列化到文章内容中，也不会附加到区块 HTML 上。相反，该系统的输出是一个新的样式表，其 ID 为 `global-styles-inline-css`。

该机制于 WordPress 5.8 中引入。当时，它仅从 WordPress 和当前主题获取数据。WordPress 5.9 扩展了该系统，使其也能从用户处获取样式数据。

以下是基本的数据流程：

全局样式的数据流程

生成样式表的过程本质上包含三个步骤：

1. 收集数据：WordPress 内置的 `theme.json` 文件、当前主题的 `theme.json` 文件（如果存在），以及用户通过站点编辑器中的全局样式界面提供的样式。
2. 整合数据：将来自不同来源（WordPress 默认值、主题和用户）的结构化信息进行规范化并合并为一个单一结构。
3. 将数据转换为样式表：将内部表示转换为 CSS 样式规则，并将其作为样式表加入队列。

收集数据

数据可来自三个不同来源：WordPress 默认设置、当前激活主题或用户。这三者均采用相同的 `theme.json` 格式。

来自 WordPress 和当前激活主题的数据从对应的 `theme.json` 文件中获取。用户数据则从数据库中提取，这些数据在用户通过站点编辑器的全局样式侧边栏保存更改后存储。

整合数据

此阶段的目标是构建一个统一的结构。

此阶段包含两个重要流程。首先，系统需要规范化所有输入数据，因为不同来源可能使用不同版本的 `theme.json` 格式。例如，某个主题可能使用 v1 版本，而 WordPress 核心则使用最新版本。其次，系统需要确定如何将输入数据合并为单一结构。这将是后续章节的重点内容。

Styles

Different parts of the incoming `theme.json` structure are treated differently. The data present in the `styles` section is blended together following this logic: user data overrides theme data, and theme data overrides WordPress data.

For example, if we had the following three `theme.json` structures coming from WordPress, the theme, and the user respectively:

```
{
  "styles": {
    "color": {
      "background": "<WordPress value>"
```

```
□□},
□□"typography": {
□□□"fontSize": "<WordPress value>"
□□}
□}
}
```

```
{
□"styles": {
□□"typography": {
□□□"fontSize": "<theme value>",
□□□"lineHeight": "<theme value>"
□□}
□}
}
```

```
{
□"styles": {
□□"typography": {
□□□"lineHeight": "<user value>"
□□}
□}
}
```

The result after the consolidation would be:

```
{
□"styles": {
□□"color": {
□□□"background": "<WordPress value>"
□□},
□□"typography": {
□□□"fontSize": "<theme value>",
□□□"lineHeight": "<user value>"
□□}
□}
}
```

Settings

The `settings` section works differently than styles. Most of the settings are only used to configure the editor and have no effect on the global styles. Only a few of them are part of the resulting stylesheet: the presets.

Presets are the predefined styles that are shown to the user in different parts of the UI: the color palette or the font sizes, for example. They comprise the following settings: `color.duotone`, `color.gradients`, `color.palette`, `typography.fontFamilies`, `typography.fontSizes`. Unlike `styles`, presets from an origin don't override values from other origins. Instead, all of them are stored in the consolidated structure.

For example, if we have the following `theme.json` structures coming from WordPress, the theme, and the user respectively:

```
{
	"settings": {
		"color": {
			"palette": [ "<WordPress values>" ],
			"gradients": [ "<WordPress values>" ]
		}
	}
}
```

```
{
	"settings": {
		"color": {
			"palette": [ "<theme values>" ]
		},
		"typography": {
			"fontFamilies": [ "<theme values>" ]
		}
	}
}
```

```
{
	"settings": {
		"color": {
			"palette": [ "<user values>" ]
		}
	}
}
```

The result after the consolidation would be:

```
{
  "settings": {
    "color": {
      "palette": {
        "default": [ "<WordPress values>" ],
        "theme": [ "<theme values>" ],
        "user": [ "<user values>" ]
      },
      "gradients": {
        "default": [ "<WordPress values>" ]
      }
    },
    "typography": {
      "fontFamilies": {
        "theme": [ "<theme values>" ]
      }
    }
  }
}
```

From data to styles

The last phase of generating the stylesheet is converting the consolidated data into CSS style rules.

Styles to CSS rules

The `styles` section can be thought of as a structured representation of CSS rules, each chunk representing a CSS rule:

- A key/value in theme.json maps to a CSS declaration (`property: value`).
- The CSS selector for a given chunk is generated based on its semantics:
  - The top-level section uses the `body` selector.
  - The top-level elements use an ID selector matching the HTML element they represent (for example, `h1` or `a`).
  - Blocks use the default class name they generate (`core/group` becomes `.wp-block-group`) unless they explicitly set a different one using their `block.json` (`core/paragraph` becomes `p`). See the "Current limits" section for more about this.
  - Elements within a block use the concatenation of the block and element selector.

For example, the following `theme.json` structure:

```
{
  "styles": {
    "typography": {
      "fontSize": "<top-level value>"
    },
    "elements": {
      "h1": {
        "typography": {
          "fontSize": "<h1 value>"
        }
      }
    },
    "blocks": {
      "core/paragraph": {
        "color": {
          "text": "<paragraph value>"
        }
      },
      "core/group": {
        "color": {
          "text": "<group value>"
        },
        "elements": {
          "h1": {
            "color": {
              "text": "<h1 within group value>"
            }
          }
        }
      }
    }
  }
}
```

is converted to the following CSS:

```
body {
  font-size: <top-level value>;
}
h1 {
```

```
font-size: <h1 value>;
}
p {
color: <paragraph value>;
}
.wp-block-group {
color: <group value>;
}
.wp-block-group h1 {
color: <h1 within group value>;
}
```

□□□□ CSS □□

□ `settings` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ CSS □□□□□□ `--wp--preset--<category>-<slug>` □□□□□□□□□□□□□□□□□□□□□□□□

□□□□□ theme.json

```
{
"settings": {
"color": {
"palette": {
"default": [
{
"slug": "vivid-red",
"value": "#cf2e2e",
"name": "Vivid Red"
}
],
"theme": [
{
"slug": "foreground",
"value": "#000",
"name": "Foreground"
}
]
}
},
"blocks": {
"core/site-title": {
```

```
□□□□"color": {
□□□□□"palette": {
□□□□□□"theme": [
□□□□□□□{
□□□□□□□□"slug": "foreground",
□□□□□□□□"value": "#1a4548",
□□□□□□□□"name": "Foreground"
□□□□□□□}
□□□□□□]
□□□□□}
□□□□}
□□}
□}
}
```

□□□□□□□ CSS □□□□□

```
body {
  --wp--preset--color--vivid-red: #cf2e2e;
  --wp--preset--color--foreground: #000;
}

.wp-block-site-title {
  --wp--preset--color--foreground: #1a4548;
}
```

□□ CSS □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ CSS □□□□□□□□□□□□□□□ CSS □□

```
/* vivid-red */
.has-vivid-red-color { color: var(--wp--preset--color--vivid-red) !important; }
.has-vivid-red-background-color { background-color: var(--wp--preset--color--vivid-red)
!important; }
.has-vivid-red-border-color { border-color: var(--wp--preset--color--vivid-red) !important; }

/* foreground */
.has-foreground-color { color: var(--wp--preset--color--foreground) !important; }
.has-foreground-background-color { background-color: var(--wp--preset--color--foreground)
!important; }
```

```
.has-foreground-border-color { border-color: var(--wp--preset--color--foreground) !important;
}

/* 站点标题内的 foreground */
.wp-block-site-title .has-foreground-color { color: var(--wp--preset--color--foreground)
!important; }
.wp-block-site-title .has-foreground-background-color { background-color: var(--wp--preset--
color--foreground) !important; }
.wp-block-site-title .has-foreground-border-color { border-color: var(--wp--preset--color--
foreground) !important; }
```

Global Styles API 的当前限制

1. 为区块设置不同的 CSS 选择器需要服务器端注册

默认情况下，分配给区块的选择器是 `.wp-block-<block-name>`。然而，区块可以根据需要更改此设置。它们可以通过其 `block.json` 中的 `__experimentalSelector` 属性提供一个 CSS 选择器。

如果区块这样做，则需要在服务器端使用 `block.json` 进行注册，否则全局样式代码将无法访问该信息，并将使用区块的默认 CSS 选择器。

2. 无法为不同样式定位不同的 HTML 节点

每个样式块只能使用单个选择器。

如果区块使用 `__experimentalSkipSerialization` 将不同样式属性序列化到包装器之外的其他节点，这一点尤其重要。更多信息请参阅“区块支持的当前限制”。

3. 每个区块仅支持单一属性

与区块支持类似，任何样式在区块中只能有一个实例。例如，一个区块只能有一种字体大小。请参阅相关的“区块支持的当前限制”。

4. 仅使用区块支持的区块会显示在全局样式界面中

站点编辑器的全局样式界面有一个用于设置每个区块样式的屏幕。区块列表是使用区块 `block.json` 中的区块支持动态生成的。如果一个区块希望被列出，它需要使用区块支持机制。

布局样式

除了单个区块级别和全局样式外，还存在布局样式的概念，这些样式会同时输出给基于区块的主题和经典主题。

布局区块支持输出用于创建布局的区块之间共享的通用布局样式。布局样式对于为任何作为其他区块容器的区块提供通用样式非常有用。依赖这些布局样式的区块示例包括分组、行、列、按钮和社交图标。该功能通过区块 `block.json` 文件中 `supports` 下的 `layout` 设置在核心区块中启用。

布局样式主要在以下两个位置输出：

基础布局样式

基础布局样式是指所有选择特定布局类型的区块所共有的样式。常见的基础布局样式示例包括：为使用弹性布局类型的区块（如按钮和社交图标）设置 `display: flex`，以及为受限布局提供默认的最大宽度。

基础布局样式由处理全局样式的主要 PHP 类输出，并构成全局样式表的一部分。为了在经典主题中支持核心区块，无论主题是否提供自己的 `theme.json` 文件，这些样式始终会被输出。

常见的布局定义存储在核心布局区块支持文件中。

独立布局样式

当启用了布局支持的区块被渲染时，会通过 `layout.php` 处理并添加两项内容到输出中：

- 语义化类名被添加到区块标记中，以指示正在使用的布局设置。例如，`is-layout-flow` 用于使用默认/流式布局的区块（如群组），而 `is-content-justification-right` 则会在用户将区块设置为右对齐时添加。
- 为正在渲染的单个区块上设置的非默认布局值生成独立样式。这些样式通过容器类名附加到区块上，类名格式为 `wp-container-$id`，其中 `$id` 是一个唯一数字。

可用的布局类型

目前有四种布局类型在使用：

- 默认/流式：项目垂直堆叠。父容器块的显示值未指定，因此可以使用该 HTML 元素的默认值。对于大多数元素，这通常是 `block`。子元素之间的间距通过垂直边距处理。
- 约束式：项目垂直堆叠，使用与流式布局相同的间距逻辑。为子内容提供约束宽度，输出标准内容尺寸和宽尺寸的宽度。默认使用在 `theme.json` 的 `settings.layout` 中设置的全局 `contentSize` 和 `wideSize` 值。
- 弹性：项目使用弹性盒布局显示。默认为水平方向。子元素之间的间距通过 `gap` CSS 属性处理。
- 网格：项目使用网格布局显示。默认为 `auto-fill` 方式生成列，但也可以设置为固定列数。子元素之间的间距通过 `gap` CSS 属性处理。

关于控制块之间的间距以及启用块间距控制，请参阅：什么是 blockGap 以及如何使用它？。

从主题中定位布局或容器区块

布局区块支持旨在通过区块编辑器和站点编辑器控制布局功能。在可能的情况下，尽量使用区块的功能来确定特定的布局需求，而不是依赖额外的样式表。

对于希望定位容器区块以添加或调整特定样式的主题，区块的类名通常是最佳选择。诸如 `wp-block-group` 或 `wp-block-columns` 等类名通常是定位特定区块的可靠选择。除了区块和布局类名外，还有一个由区块和布局组合而成的类名：例如，对于具有约束布局的群组区块，它将为 `wp-block-group-is-layout-constrained`。

对于定位使用特定布局类型的区块，请避免定位 `wp-container-`，因为容器类可能不会始终出现在渲染的标记中。

语义化类名

目前正在扩展布局块支持输出的稳定语义化类名。相关工作正在此议题中讨论。

当前可通过布局块支持输出的语义化类名包括：

- `is-layout-flow`：使用默认/流式布局类型的块。
- `is-layout-constrained`：使用约束布局类型的块。
- `is-layout-flex`：使用弹性布局类型的块。
- `is-layout-grid`：使用网格布局类型的块。
- `wp-container-$id`：其中 `$id` 为半随机数。仅当块包含非默认布局值时存在的容器类。此类不应直接用于任何 CSS 目标选择，因为它可能存在也可能不存在。
- `is-horizontal`：当块显式将 `orientation` 设置为 `horizontal` 时。
- `is-vertical`：当块显式将 `orientation` 设置为 `vertical` 时。
- `is-content-justification-left`：当块显式将 `justifyContent` 设置为 `left` 时。
- `is-content-justification-center`：当块显式将 `justifyContent` 设置为 `center` 时。
- `is-content-justification-right`：当块显式将 `justifyContent` 设置为 `right` 时。
- `is-content-justification-space-between`：当块显式将 `justifyContent` 设置为 `space-between` 时。
- `is-nowrap`：当块显式将 `flexWrap` 设置为 `nowrap` 时。

选择退出生成的布局样式

默认情况下会输出布局样式，因为核心结构块需要这些样式。但主题可以通过使用 `disable-layout-styles` 块支持来退出生成的块布局样式，同时保留语义类名输出。此类主题将负责提供自己的所有布局样式。请参阅主题支持下的条目。

□□□ #2
□ □□□ □□□ 2026-03-26 17:08:43 UTC
□ □□□ □□□ 2026-03-26 17:08:43 UTC