客户端导航

目录
客户端导航的工作原理
Interactivity Router 入门指南
添加路由依赖
确保脚本模块在导航期间加载
设置路由器区域
Where to place router regions
实现导航
实现预获取功能
完整示例：分页
更高级的用例
处理滚动和焦点
在导航时添加新区域
使用 data-wp-key 保留元素
Choosing good key values
Handling server state updates
Overriding router's internal in-memory cached pages Using custom HTML
管理浏览器历史记录
修改超时时间
Handling fetch errors
Disabling client-side navigation on certain pages
禁用导航反馈
订阅页面变更
深入理解交互式路由
The internal in-memory page cache
Router regions
CSS 处理
脚本模块处理
Server state and context

- [redacted]
  - [redacted]
  - [redacted]
  - [redacted]
- [redacted]

[redacted]
HTML [redacted] DOM
[redacted]

Interactivity API [redacted] `@wordpress/interactivity-router` [redacted] [redacted] [redacted] `data-wp-router-region` [redacted] URL [redacted] —— [redacted]

Interactivity API [redacted]

- [redacted] —— WordPress [redacted]
- [redacted] [redacted] —— [redacted] `<body>` [redacted]
  [redacted]

[redacted]

[redacted] `data-wp-on--click` [redacted] `actions.navigate()` [redacted]

1. [redacted] [redacted] URL [redacted] HTML[redacted]
2. [redacted] [redacted] HTML [redacted]
3. [redacted] **DOM**[redacted] “[redacted]”[redacted]
4. [redacted] [redacted]
5. [redacted] [redacted]
6. [redacted] [redacted]

[redacted]

- [redacted] [redacted] DOM [redacted]
- [redacted] [redacted]
- [redacted] [redacted]
- **SEO** [redacted] [redacted] HTML [redacted]

Interactivity Router 入门指南

`@wordpress/interactivity-router` 包自 WordPress 6.5 版本起已捆绑在核心中。如果您要开始一个新项目，最简单的设置方法是使用 `@wordpress/create-block-interactive-template` 脚手架工具，它会创建一个已配置好 Interactivity API 的区块：

```
npx @wordpress/create-block@latest my-interactive-block --template @wordpress/create-block-interactive-template
```

无论您使用的是区块还是经典主题，添加客户端导航都涉及相同的步骤：

1. 添加路由依赖：将 `@wordpress/interactivity-router` 添加为脚本模块的依赖项。
2. 确保脚本模块在导航期间加载：标记您的脚本模块，以便路由器知道在新页面上加载它。
3. 定义路由区域：使用 `data-wp-router-region` 属性标记应在导航期间更新的 HTML 元素。
4. 触发导航：使用路由器的 `actions.navigate()` 函数以编程方式进行导航。

步骤 1 和 2 根据您使用的是区块还是经典主题而有所不同，将在下方详细介绍。无论您的设置如何，步骤 3 和 4 都是相同的。

添加路由依赖

`@wordpress/interactivity-router` 模块应作为动态依赖添加，以便仅在需要时获取。

对于区块，这可以通过在 `view.js` 文件中动态导入包来实现。区块构建工具（`wp-scripts`）会检测到动态导入并自动注册 PHP 端的依赖：

```
const { actions } = yield import( '@wordpress/interactivity-router' );
yield actions.navigate( url );
```

对于经典主题，无需依赖区块的 `block.json`，而是手动在 PHP 中注册并加入脚本模块队列，将 `@wordpress/interactivity-router` 列为动态依赖。同时，直接在主题模板文件中添加交互性 API 指令，并使用 `wp_interactivity_process_directives()` 处理它们，如服务端渲染指南所述。

```
// functions.php
add_action( 'wp_enqueue_scripts', function () {
    wp_register_script_module(
        'my-theme/navigation',
        get_template_directory_uri() . '/assets/navigation.js',
        array(
□□□'@wordpress/interactivity',
```

```
□□□array(
□□□□'id'     => '@wordpress/interactivity-router',
□□□□'import' => 'dynamic',
□□□),
□□)
    );
    wp_enqueue_script_module( 'my-theme/navigation' );
} );
```

确保脚本模块在导航期间加载

在客户端导航期间，路由器需要知道新页面应加载哪些脚本模块。它通过查找 `<script>` 标签上的 `data-wp-router-options` 属性来识别它们，该属性需将 `loadOnClientNavigation` 设置为 `true`。如果没有此属性，路由器将不会在客户端导航期间加载脚本模块，并且区块的交互性将无法在新页面上工作。

对于 区块，当区块在其 `block.json` 中声明支持交互性时，此属性会自动添加。以下任一配置均可生效：

```
{
□"supports": {
□□"interactivity": true
□}
}
```

```
{
□"supports": {
□□"interactivity": {
□□□"clientNavigation": true
□□}
□}
}
```

如果你的区块 `block.json` 已包含其中一项，则无需额外设置——WordPress 会处理其余部分。

对于 经典主题 和 `block.json` 之外注册的其他脚本模块，该属性不会自动添加。你必须使用 `add_client_navigation_support_to_script_module()` 显式注册脚本模块以支持客户端导航：

```
wp_interactivity()->add_client_navigation_support_to_script_module(
    'my-theme/navigation'
);
```

如果没有此操作，当导航到需要该脚本模块的页面时，路由器将不会加载你的脚本模块。

设置路由器区域

路由器区域是页面中在客户端导航期间由路由器更新的部分。您可以通过在同一元素上同时添加 `data-wp-router-region` 和 `data-wp-interactive` 来定义一个路由器区域——目前这两个指令都是必需的。

`data-wp-router-region` 指令接受一个唯一的 ID 作为其值。当导航发生时，路由器会根据 ID 匹配当前页面和目标页面上的区域，并替换其内容——路由器区域之外的所有内容保持不变。每个区域在页面内必须是唯一的；如果两个区域共享相同的 ID，路由器将不知道更新哪一个。

以下是一个基本的路由器区域示例：

```
<div
    data-wp-interactive="myPlugin"
    data-wp-router-region="myPlugin/posts-list"
>
    <?php foreach ( $posts as $post ) : ?>
        <article>
            <h2><?php echo esc_html( $post->post_title ); ?></h2>
            <p><?php echo esc_html( $post->post_excerpt ); ?></p>
        </article>
    <?php endforeach; ?>
</div>
```

Where to place router regions

Router regions can be placed anywhere on the page. Their behavior depends on where they sit relative to other interactive elements and other router regions:

- **As a standalone element** — When a router region is not inside any existing `data-wp-interactive` element, it serves a dual role: it is the interactive boundary (since it also contains `data-wp-interactive`) *and* its content is updated during navigation:

  ```
  <div
  data-wp-interactive="myPlugin"
  data-wp-router-region="myPlugin/content"
  >
  <!-- Interactive boundary + navigable region -->
  <p data-wp-text="state.message">Hello</p>
  </div>
  ```

- **Inside an interactive element** — When a router region is nested inside an element that already has `data-wp-interactive`, the region becomes part of that element's interactivity. The parent interactive element stays untouched during navigation, but the region's content is updated:

```
<div data-wp-interactive="myPlugin">
  <h1>This heading is never updated during navigation</h1>

  <div
    data-wp-interactive="myPlugin"
    data-wp-router-region="myPlugin/posts"
  >
    <!-- This content is updated during navigation -->
  </div>
</div>
```

  Note that the router region still needs its own `data-wp-interactive` directive, even though it is already inside one.
- **Inside another router region** — When a router region is nested inside another router region, it becomes part of the parent region. The parent region is updated as a single unit during navigation; the nested region is not processed independently:

```
<div data-wp-interactive="myPlugin" data-wp-router-region="myPlugin/main">
  <!-- This inner region is part of "myPlugin/main" -->
  <div
    data-wp-interactive="myPlugin"
    data-wp-router-region="myPlugin/sidebar"
  >
    <!-- Updated together with the parent region -->
  </div>
</div>
```

□□□

□□□□□□□□□□□□ store □□□□ **action**□□□ Interactivity API □□□□□□□ DOM □□□ Actions □□ `store()` □□□□□□□□□□□□□□□□□□□□□□ `data-wp-on--click` □□□□□□□□□□□□□□□□□□□□ action □□□□

□□□□□□□□□□□□□□□□□□□□ HTML □□□□□□□□□□ `navigateTo` action□

```
<a data-wp-on--click="actions.navigateTo" href="/page-2/"> □□□□ 2 </a>
```

然后，在你的脚本模块中，定义 `navigateTo` action。它会阻止浏览器的默认整页导航，转而使用路由器的 `navigate()` 函数：

```
// view.js
import { store, withSyncEvent } from '@wordpress/interactivity';

store( 'myPlugin', {
  actions: {
    navigateTo: withSyncEvent( function* ( event ) {
      event.preventDefault();

      const { actions } = yield import(
        '@wordpress/interactivity-router'
      );
      yield actions.navigate( event.target.href );
    } ),
  },
} );
```

> 对于需要调用同步事件方法（如 `event.preventDefault()`）的 actions，需要使用 `withSyncEvent()` 包装器。详情请参阅 withSyncEvent() 文档。

实现预获取功能

路由器还提供了一个 `prefetch()` 函数，用于获取页面并将其存储在内部内存缓存中，而无需执行导航。通过在用户点击之前预获取页面，后续导航会感觉瞬间完成，因为内容已经可用。

一种常见模式是在用户悬停在链接上时预获取页面，并在点击时进行导航。您可以在同一元素上使用两个指令组合这两种行为——`data-wp-on--mouseenter` 用于预获取，`data-wp-on--click` 用于导航：

```
<a
  data-wp-on--mouseenter="actions.prefetchPage"
  data-wp-on--click="actions.navigateTo"
  href="/page-2/"
>
  悬停以预获取，点击以导航
</a>
```

脚本模块中的相应操作处理每个事件：

```
// view.js
import { store, withSyncEvent } from '@wordpress/interactivity';

store( 'myPlugin', {
□actions: {
□□prefetchPage: function* ( event ) {
□□□const { actions } = yield import(
□□□□'@wordpress/interactivity-router'
□□□);
□□□yield actions.prefetch( event.target.href );
□□},

□□navigateTo: withSyncEvent( function* ( event ) {
□□□event.preventDefault();

□□□const { actions } = yield import(
□□□□'@wordpress/interactivity-router'
□□□);
□□□yield actions.navigate( event.target.href );
□□} ),
□},
} );
```

完整示例：分页

此示例结合了路由器区域、导航和预取功能，为文章列表实现客户端分页。

PHP

模板查询当前页面的文章，并在路由器区域内渲染它们。底部的分页链接允许用户在页面间跳转。当用户悬停在 "上一页" 或 "下一页" 链接上时，目标页面会被预取。当用户点击链接时，路由器进行客户端导航——仅替换路由器区域内的内容，无需完全重新加载页面。导航完成后，页面会平滑滚动到顶部。

PHP:

```
<?php
$current_page = isset( $_GET['paged'] ) ? absint( $_GET['paged'] ) : 1;
$query = new WP_Query( array(
    'paged'          => $current_page,
    'posts_per_page' => 5,
) );
```

```
?>

<div
    data-wp-interactive="myPagination"
    data-wp-router-region="myPagination/posts"
>
    <ul class="posts-list">
        <?php while ( $query->have_posts() ) : $query->the_post(); ?>
            <li>
                <a href="<?php the_permalink(); ?>"><?php the_title(); ?></a>
            </li>
        <?php endwhile; wp_reset_postdata(); ?>
    </ul>

    <nav class="pagination">
        <?php if ( $current_page > 1 ) : ?>
            <a
                data-wp-on--mouseenter="actions.prefetch"
                data-wp-on--click="actions.navigate"
                href="?paged=<?php echo $current_page - 1; ?>"
            >
                &larr; □□□
            </a>
        <?php endif; ?>

        <span>□ <?php echo $current_page; ?> □</span>

        <?php if ( $query->max_num_pages > $current_page ) : ?>
            <a
                data-wp-on--mouseenter="actions.prefetch"
                data-wp-on--click="actions.navigate"
                href="?paged=<?php echo $current_page + 1; ?>"
            >
                □□□ &rarr;
            </a>
        <?php endif; ?>
    </nav>
</div>
```

JavaScript:

```
import { store, withSyncEvent } from '@wordpress/interactivity';

store( 'myPagination', {
  actions: {
    prefetch: function* ( event ) {
      const { actions } = yield import(
        '@wordpress/interactivity-router'
      );
      yield actions.prefetch( event.target.href );
    },

    navigate: withSyncEvent( function* ( event ) {
      event.preventDefault();

      const { actions } = yield import(
        '@wordpress/interactivity-router'
      );
      yield actions.navigate( event.target.href );

      // ページ遷移後にスクロールをトップに戻す
      window.scrollTo( { top: 0, behavior: 'smooth' } );
    } ),
  },
} );
```

[illegible]

[illegible]

[illegible] —— [illegible] `actions.navigate()`
[illegible]
[illegible]

[illegible]

```
store( 'myPlugin', {
  actions: {
    navigateTo: withSyncEvent( function* ( event ) {
```

```
□□□event.preventDefault();

□□□const { actions } = yield import(
□□□□'@wordpress/interactivity-router'
□□□);
□□□yield actions.navigate( event.target.href );

□□□□□后滚动到顶部。
□□window.scrollTo( { top: 0, behavior: 'smooth' } );
□□} ),
□},
} );
```

为了可访问性，请考虑在导航后将焦点移动到有意义的元素上，例如主要内容区域或标题，以便键盘和屏幕阅读器用户知道他们在新页面上的位置。

在导航时添加新区域

有时你需要一些仅在特定页面显示的 UI 元素 —— 比如模态框、侧边栏或通知面板。使用常规的路由器区域时，当前页面必须已存在某个区域才能在导航期间更新它。 `attachTo` 选项通过允许你定义动态创建的区域来解决这个问题：当导航到包含这些区域的页面时，即使原始页面中不存在这些区域，它们也会被动态创建并插入到 DOM 中。

使用 **`attachTo`** 定义区域：

```
<div
□data-wp-interactive="myPlugin"
□data-wp-router-region='{ "id": "myPlugin/modal", "attachTo": "body" }'
>
□<div class="modal-overlay">
□□<div class="modal-content">
□□□<h2>□□□标题</h2>
□□□□□□内容在此...</p>
□□</div>
□</div>
</div>
```

`attachTo` 的值是一个 CSS 选择器。当从没有此区域的页面导航到此页面时，该区域将被创建并附加到与选择器匹配的元素上。

示例：导航时显示的模态框

没有模态框的页面 *(page-1.php)*：

```
<div
    data-wp-interactive="myPlugin"
    data-wp-router-region="myPlugin/content"
>
    <h1>页面 1</h1>
    <a
        data-wp-on--click="actions.navigate"
        href="/page-with-modal/"
    >
        打开包含模态框的页面
    </a>
</div>
```

包含模态框的页面 *(page-2.php)*：

```
<div
    data-wp-interactive="myPlugin"
    data-wp-router-region="myPlugin/content"
>
    <h1>页面 2</h1>
    <a
        data-wp-on--click="actions.navigate"
        href="/page-without-modal/"
    >
        关闭模态框
    </a>
</div>

<div
    data-wp-interactive="myPlugin"
    data-wp-router-region='{ "id": "myPlugin/modal", "attachTo": "body" }'
>
    <div class="modal-overlay">
        <div class="modal-content">
            <h2>我是一个模态框！</h2>
        </div>
    </div>
```

```
</div>
```

当从页面 1 导航到页面 2 时，模态框区域被创建并附加到 `<body>` 上。当导航回页面 1 时，模态框会自动被移除。

使用 `data-wp-key` 保留元素

在客户端导航期间，路由器使用 Preact 的协调算法来更新路由区域内的内容。该算法依赖启发式方法来高效匹配当前页面和目标页面之间的元素。这些启发式方法在大多数情况下效果良好，但在某些条件下可能会失效——例如，当两个元素在不同页面上具有相同类型和位置但使用不同指令时。在这种情况下，算法可能会错误地将它们视为同一元素，导致状态损坏或行为异常。

为了防止这种情况，您可以使用 `data-wp-key` 指令为元素提供稳定、明确的标识。当协调算法遇到带键元素时，它会通过键来匹配它们，而不是依赖启发式方法。具有匹配键的元素会在原地更新，保留其内部状态：焦点、滚动位置、CSS 动画、表单输入值以及对 DOM 节点的任何 JavaScript 引用。不匹配的元素会根据需要被干净地移除或创建。

键在以下两种场景中尤为重要：

1. 跨页面变化的列表——例如分页文章、筛选结果或排序表格。
2. 页面间结构不同的区域——例如，一个区域在一个页面上包含侧边栏而在另一个页面上没有，或者在相同区域渲染不同块的页面。

如果没有键，协调启发式方法可能会错误地匹配恰好具有相同类型和位置的不相关元素。在最好的情况下，这会导致不必要的 DOM 重新创建；在最坏的情况下，它可能损坏元素状态或产生错误的标记——例如，将一个元素的指令应用到完全不同的元素上。

使用基于稳定标识符的键，算法可以通过标识来匹配元素，而不是依赖启发式方法。这确保了每个元素在导航过程中都能被正确识别。

PHP:

```
<div
    data-wp-interactive="myPagination"
    data-wp-router-region="myPagination/posts"
>
    <ul>
        <?php while ( $query->have_posts() ) : $query->the_post(); ?>
            <li data-wp-key="post-<?php echo get_the_ID(); ?>">
                <a href="<?php the_permalink(); ?>">
                    <?php the_title(); ?>
```

```
                </a>
            </li>
        <?php endwhile; wp_reset_postdata(); ?>
    </ul>
</div>
```

每个 `<li>` 都通过文章 ID 设置键。如果用户从一个结果页面导航到另一个页面，并且一篇文章同时出现在两个页面上，路由器会为该文章重用现有的 DOM 节点，而不是销毁并重新创建它。

键对于非列表元素同样有用。如果一个路由区域在不同页面上渲染结构不同的内容，为顶级部分设置键有助于算法区分它们：

```
<div
    data-wp-interactive="myPlugin"
    data-wp-router-region="myPlugin/content"
>
    <?php if ( is_product_page() ) : ?>
        <section data-wp-key="product-detail">
            <!-- Product detail layout -->
        </section>
    <?php else : ?>
        <section data-wp-key="product-list">
            <!-- Product list layout -->
        </section>
    <?php endif; ?>
</div>
```

Without keys, navigating between these two pages would cause the algorithm to patch the product-detail `<section>` into the product-list `<section>` (or vice versa) by position, potentially corrupting their internal state. With distinct keys, the algorithm recognizes they are different elements and cleanly replaces one with the other.

Choosing good key values

A key should be:

- **Stable**: The same item should always produce the same key, regardless of its position in the list.
- **Unique among siblings**: No two sibling elements should share the same key. Keys only need to be unique within their parent, not globally.

Use data-derived identifiers whenever possible — post IDs, term IDs, or any value that uniquely identifies the item. Avoid using array indices as keys, because indices change when items are reordered, added, or removed, which defeats the purpose of keying.

```
<!-- Good: stable, data-derived key -->
<li data-wp-key="post-42">...</li>

<!-- Bad: index-based key (changes when items shift) -->
<li data-wp-key="item-0">...</li>
```

Handling server state updates

During client-side navigation, the client-side state persists while the server provides new state for the target page. In some cases, you may want parts of your client state to stay in sync with what the server provides for each page — for example, updating a product count that changes across pages, or resetting an "expanded" flag based on the new page's context.

Use `getServerState()` and `getServerContext()` to react specifically to server-provided values and selectively update the client state in a callback:

```
import {
	store,
	getContext,
	getServerState,
	getServerContext,
} from '@wordpress/interactivity';

const { state } = store( 'myPlugin', {
	callbacks: {
		syncWithServer() {
			const serverState = getServerState();
			const serverContext = getServerContext();
			const context = getContext();

			// Keep the product count in sync with the server across navigations.
			if ( serverState.productCount !== undefined ) {
				state.productCount = serverState.productCount;
			}

			// Reset the expanded state based on the new page's context.
```

```
		if ( serverContext.isExpanded !== undefined ) {
			context.isExpanded = serverContext.isExpanded;
		}
	},
},
} );
```

For more details, see the Understanding global state, local context, and derived state guide.

Overriding router's internal in-memory cached pages

By default, once a page is stored in the router's internal in-memory cache, subsequent navigations use the cached version without making a new network request. Use the `force` option to bypass the router's internal in-memory cache and re-fetch the page from the server:

```
// Force re-fetch with navigate().
yield actions.navigate( '/products/', { force: true } );

// Force re-fetch with prefetch().
yield actions.prefetch( '/products/', { force: true } );
```

> If you're using `force: true` to refresh a page after a mutation (POST, PUT, DELETE request), make sure the mutation has completed before navigating:

```
store( 'myPlugin', {
	actions: {
		deleteAndRefresh: function* () {
			// Wait for the deletion to complete.
			yield fetch( '/wp-json/wp/v2/posts/123', { method: 'DELETE' } );

			// Now refresh the page to show updated data.
			const { actions } = yield import(
				'@wordpress/interactivity-router'
			);
			yield actions.navigate( window.location.href, { force: true } );
		},
	},
```

```
} );
```

Using custom HTML

Instead of fetching a page from a URL, you can provide HTML directly using the `html` option:

```
// Navigate with custom HTML.
yield actions.navigate( '/custom-page/', {
    html: `
        <div data-wp-interactive="myPlugin" data-wp-router-region="myPlugin/content">
            <h1>Custom Content</h1>
            <p>This HTML was provided directly, not fetched.</p>
        </div>
    `,
} );

// Prefetch with custom HTML.
yield actions.prefetch( '/custom-page/', {
    html: customHtmlString,
} );
```

This is useful when you need to control the `fetch` request yourself.

管理浏览器历史记录

默认情况下，`navigate()` 会使用 `pushState` 在浏览器的会话历史记录中添加一个新条目。使用 `replace` 选项可以替换当前的历史记录条目：

```
// 默认行为：添加新的历史记录条目（pushState）。
yield actions.navigate( '/page-2/' );

// 替换当前历史记录条目（replaceState）。
yield actions.navigate( '/page-2/', { replace: true } );
```

在以下情况下使用 `replace: true`：

- 更新用于过滤/排序的查询参数，且每次更改不应作为独立的历史记录条目。
- 实现无限滚动时更新 URL，但又不希望每个页面都成为独立的历史记录条目。

修改超时时间

如果导航耗时过长，路由器将回退到传统的整页加载方式。默认超时时间为 10 秒。可通过 `timeout` 选项进行调整：

```
// 设置较短超时以实现快速失败
yield actions.navigate( '/page/', { timeout: 5000 } );

// 为慢速连接设置较长超时
yield actions.navigate( '/page/', { timeout: 30000 } );
```

Handling fetch errors

When navigation fails (network error, timeout, or server error), the router automatically falls back to a full page reload. This means you cannot catch fetch errors from `navigate()` directly — the browser takes over before your code has a chance to handle them.

If you need custom error handling (for example, showing an error message instead of reloading), you can fetch the page manually, handle any errors yourself, and then pass the fetched HTML to `navigate()` using the `html` option:

```
store( 'myPlugin', {
	actions: {
		navigateWithCustomErrorHandling: withSyncEvent( function* ( event ) {
			event.preventDefault();
			const url = event.target.href;

			try {
				// Fetch the page manually.
				const response = yield fetch( url );

				if ( ! response.ok ) {
					// Handle HTTP errors.
					state.error = `Error: ${ response.status }`;
					return;
				}

				const html = yield response.text();
```

```
□□□□// Navigate using the fetched HTML.
□□□□const { actions } = yield import(
□□□□□'@wordpress/interactivity-router'
□□□□);
□□□□yield actions.navigate( url, { html } );
□□□} catch ( error ) {
□□□□state.error = 'Network error. Please check your connection.';
□□□}
□□} ),
□},
} );
```

Disabling client-side navigation on certain pages

Some pages may require a full page reload instead of client-side navigation. Use `wp_interactivity_config()` to disable client navigation:

```
// In your theme's functions.php or a plugin.
add_action( 'wp', function() {
    // Disable on specific page templates.
    if ( is_page_template( 'template-complex.php' ) ) {
        wp_interactivity_config(
            'core/router',
            array( 'clientNavigationDisabled' => true )
        );
    }
} );
```

When `clientNavigationDisabled` is `true`:

- `actions.navigate()` triggers a full page reload.
- `actions.prefetch()` does nothing.
- Navigating from another page to this page forces a reload.

□□□□□

Interactivity API □□□□□□□□□□□□□□

- 加载动画：导航期间出现在页面顶部的进度条。如果导航尚未完成，该进度条会在短暂延迟（400 毫秒）后出现。引入这 400 毫秒延迟是为了避免在页面已成功预取或连接速度极快时显示动画。
- 屏幕阅读器播报：用于导航进度的无障碍播报。

在某些情况下，您可能需要禁用这些功能：

```
// 禁用加载动画（用于实现即时感更新）。
yield actions.navigate( '/page/', { loadingAnimation: false } );

// 禁用屏幕阅读器播报（当提供自定义播报时）。
yield actions.navigate( '/page/', { screenReaderAnnouncement: false } );

// 同时禁用两者。
yield actions.navigate( '/page/', {
    loadingAnimation: false,
    screenReaderAnnouncement: false,
} );
```

禁用反馈的适用场景：

- 静默更新：不希望引起注意的后台刷新。
- 自定义加载界面：当您正在实现自己的加载指示器时。
- 自定义无障碍功能：当您正在提供自己的屏幕阅读器播报时。

订阅页面变更

`core/router` 存储库暴露了一个响应式的 `state.url` 属性，该属性会在每次客户端导航发生时更新。通过在 `data-wp-watch` 或 `watch` 回调中读取此值，您可以创建一个响应式订阅，每当 URL 变更时都会重新运行。

```
// view.js
import { watch, store } from '@wordpress/interactivity';

// 存储库范围的订阅。
watch( () => {
  const { state } = store( 'core/router' );
  sendAnalyticsPageView( state.url );
} );

// 基于元素的订阅：<div data-wp-watch="callbacks.sendPageView">
store( 'myPlugin', {
  callbacks: {
```

```
□□sendPageView() {
□□□const { state } = store( 'core/router' );
□□□sendAnalyticsPageView( state.url );
□□},
□},
} );
```

> `core/router` 存储库和 `state.url` 在页面加载时即可用且已填充数据，因此无需导入 `@wordpress/interactivity-router` 包来访问它们。

深入理解交互式路由

本节详细技术性地解释了客户端导航的内部工作原理。理解这些内部机制可以帮助您调试问题、优化性能，并为如何组织代码结构做出明智的决策。

The internal in-memory page cache

At the heart of the Interactivity API router is an internal in-memory page cache — a simple store that maps URLs to their processed page representations. When you call `prefetch()` or `navigate()`, the router first checks this cache to see if the target page has already been fetched and processed.

The cache uses a normalized version of the URL as its key. This normalization strips away the domain and any hash fragments, keeping only the pathname and query parameters. For example, `https://example.com/products/?category=shoes#details` becomes `/products/?category=shoes`. This ensures that navigations to the same logical page (regardless of how the URL was constructed) share the same cache entry.

Each entry in the cache stores not just the fetched HTML, but a fully processed page representation containing:

- **Virtual DOM trees** for each router region found in the page
- **Style sheet references** needed by the page
- **Script module information** for JavaScript that should be loaded
- **The page title** for updating the document title
- **Server state** that was embedded in the page by WordPress

An important detail is that the cache stores promises rather than resolved values. When a fetch begins, the router immediately stores the pending promise in the cache. This means that if multiple calls to `prefetch()` or `navigate()` target the same URL simultaneously (for example, if a user rapidly hovers over multiple links pointing to the same page), only one network request is made. All callers receive the same promise and wait for the same result.

Once a page is in the cache, it remains there for the duration of the browser session. Subsequent navigations to that URL will use the cached version instantly, without any network request. This is why client-side navigation feels so fast after the initial visit — the page is already prepared and ready to render.

If you need to force a fresh fetch (for example, after submitting a form that changes the page's content), you can use the `force: true` option with `navigate()` or `prefetch()`. This bypasses the cache check and fetches the page anew, replacing the existing cache entry with the fresh content.

```
// Force a fresh fetch after a form submission.
const { actions } = store( 'myPlugin', {
	actions: {
		*submitForm() {
			yield fetch( '/wp-json/my-plugin/v1/submit', {
				method: 'POST',
				body: JSON.stringify( {
					/* form data */
				} ),
			} );
			// Navigate to the same page, bypassing the cache
			// to reflect the updated content.
			yield navigate( window.location.href, { force: true } );
		},
	},
} );
```

Router regions

Router regions are the sections of your page that the router knows how to update during client-side navigation. They act as boundaries that tell the router "this is the content that should change when navigating between pages."

Defining router regions

You define a router region by adding the `data-wp-router-region` attribute to an element alongside `data-wp-interactive` (as described in Setting up router regions above).

The attribute value serves as a unique identifier for that region. You can specify it in two ways:

1. As a simple string:

```
<div
	data-wp-interactive="myPlugin"
	data-wp-router-region="myPlugin/main-content"
>
	<!-- Region content -->
</div>
```

2. As a JSON object (when you need to pass other options):

```
<div
	data-wp-interactive="myPlugin"
	data-wp-router-region='{ "id": "myPlugin/modal", "attachTo": "body" }'
>
	<!-- Region content -->
</div>
```

The region ID must be unique within a single page and consistent across pages that share the same region. For example, if both your "Products" page and "Product Detail" page have a sidebar, and you want that sidebar to update during navigation, both pages should define a region with the same ID (e.g., `"myPlugin/sidebar"`).

How regions are processed during page fetch

When the router fetches a new page (either through `prefetch()` or as part of `navigate()`), it processes the HTML to extract and prepare all router regions. This happens in several steps:

First, the router parses the fetched HTML into a document structure using the browser's built-in HTML parser. This gives it a complete DOM tree to work with, just as if the page had been loaded normally.

Next, the router scans this document for all elements that have both `data-wp-interactive` and `data-wp-router-region` attributes. For each region found, it extracts the region ID and checks whether the region is nested inside another region. Only top-level regions are processed directly; nested regions are handled as part of their parent's content.

For each top-level region, the router converts the HTML into a virtual DOM (vDOM) representation. The virtual DOM is a lightweight JavaScript object structure that mirrors the actual DOM but can be compared and manipulated much more efficiently by the Interactivity API. Importantly, the region element itself is included in this conversion — not just its children. This means that attributes on the region element, such as `data-wp-context`, will also be processed and updated during navigation.

Finally, each region's virtual DOM is stored in the page cache entry, indexed by its region ID. The cache entry now contains a map of region IDs to their corresponding virtual DOM trees.

除了处理区域外，路由器还会在此步骤中提取页面的 CSS 样式表和 JavaScript
脚本模块。尚未加载的新样式表会以禁用状态添加到文档中，以便浏览器可以开始下载它们而不应用它们。同
样，新的脚本模块会被识别，其依赖树会被解析并获取。这些资源会提前准备好，这样当导航实际渲染新内容

时，所有必要的样式和脚本都已就绪。样式和脚本模块的处理细节在下面的 CSS 处理 和 脚本模块处理部分中介绍。

导航期间区域的渲染方式

当调用 navigate()
且目标页面已成功获取（或已缓存）时，路由器需要更新当前页面以显示新内容。这个渲染过程经过精心编排
，以提高效率并避免视觉故障。

路由器首先检查当前页面和目标页面中存在哪些区域。基于此比较，可能会出现三种不同的情况：

情况 1：两个页面都存在区域（更新）

这是最常见的情况。当具有给定 ID
的区域同时存在于当前页面和目标页面上时，路由器会用新内容更新现有区域。

路由器不是简单地替换整个区域的 HTML（这会破坏任何内部状态并导致突兀的视觉过渡），而是使用虚拟
DOM 差异算法。该算法比较当前区域的虚拟 DOM 与新区域的虚拟 DOM
，并计算将一个转换为另一个所需的最小更改集。

例如，如果产品列表区域在当前页面上包含 10 个产品，在新页面上包含 10
个不同的产品，差异算法可能会确定它只需要更新现有列表项元素内的文本内容和图像源 ——
而不是从头开始销毁和重新创建所有 10 个项目。这保留了 DOM
状态（如区域内的滚动位置或焦点状态）并产生更平滑的视觉过渡。

当元素共享相同类型和位置但代表不同事物时，协调算法所依赖的启发式方法可能会失败。您可以使用 data-

wp-key 指令为元素提供稳定的标识，确保它们在导航过程中正确匹配。详情请参阅 使用 data-wp-key

保留元素 。

情况 2：目标页面上存在带有 attachTo 的区域（创建）

有时页面包含一个在当前页面上不存在的区域 ——例如，仅在某些页面上出现的模态对话框。如果此区域指定了
attachTo 属性，路由器将动态创建它。

The attachTo property contains a CSS selector that identifies where in the current page the new region should be appended. When the router encounters such a region, it:

1. Finds the element matching the attachTo selector in the current page
2. Creates new DOM elements for the region
3. Appends these elements to the matched parent
4. Renders the region's virtual DOM into the newly created elements

This allows content that exists on one page but not another to appear smoothly during navigation, without requiring the target element to exist in advance.

Scenario 3: Region exists only on the current page (remove)

When a region exists on the current page but not on the target page, it means that content is no longer needed. The router handles this by setting the region's content to empty, effectively clearing it from the display.

If the region was dynamically created via `attachTo` during a previous navigation, the entire region element is removed from the DOM. If it was part of the original page structure, the element remains but its content is cleared.

What happens to HTML outside router regions?

An important detail to understand is that HTML outside of router regions remains completely untouched during client-side navigation. The router only modifies the content inside the regions it manages — everything else in the DOM stays exactly as it was.

This means that if you have static elements like a site header, footer, or navigation menu that aren't wrapped in a router region, they won't change when the user navigates between pages. This can be intentional (for elements that truly are the same across all pages) or it can be a source of confusion if you expect those elements to update.

However, there's an important exception: **interactive elements outside router regions can still react to global state changes**. If you have an interactive element outside any router region, with directives that use `getServerState()` to read global state, these directives will automatically re-evaluate when navigation brings in new server state.

For example, consider a shopping cart icon in the header that displays the number of items:

```
<!-- This header is NOT inside a router region -->
<header data-wp-interactive="myShop">
  <div class="cart-icon">
    <span data-wp-text="state.cartCount"></span> items
  </div>
</header>

<!-- This is the router region that updates during navigation -->
<main data-wp-interactive="myShop" data-wp-router-region="myShop/content">
  <!-- Page content -->
</main>
```

If `state.cartCount` comes from the regular client-side state, the cart icon will not update during navigation — even if the new page has a different cart count in its server state. The header, while

being interactive, is outside any router region, so it's not re-rendered.

But if you use `getServerState()` instead:

```
const { state } = store( 'myShop', {
□state: {
□□get cartCount() {
□□□// This reacts to server state changes during navigation.
□□□return getServerState().cartCount;
□□},
□},
} );
```

Now the cart icon will update whenever navigation brings in a new `cartCount` value from the server, even though the header itself is outside any router region. This is because `getServerState()` creates a reactive subscription to server-provided state, which is updated during every navigation.

This pattern is useful for global UI elements that need to stay synchronized with server data across navigations, without requiring them to be inside a router region. However, `getServerState()` can also be used to synchronize the `state` of interactive elements inside router regions, as described in the Handling server state updates section.

CSS 处理

客户端导航中较为棘手的方面之一是管理 CSS
样式表。不同页面可能需要不同的样式，路由器必须确保每个页面激活正确的样式 ——
同时避免出现无样式内容闪烁或破坏 CSS 层叠顺序。

CSS 层叠顺序的挑战

CSS 规则按特定顺序应用，当两条规则具有相同特异性时，文档中靠后的规则 "胜出"。这意味着 HTML 中 `<link>` 和 `<style>`
元素的顺序至关重要。如果路由器简单地将新样式表追加到文档末尾，可能会无意中改变规则的优先级，导致
视觉错误。

考虑这个例子：页面 A 有样式表 `base.css` 和 `theme.css`，页面 B 有 `base.css`、`components.css` 和
`theme.css`。如果用户从 A 导航到 B，路由器需要在 `base.css` 和 `theme.css` 之间插入 `components.css` ——
而不是在末尾。否则，`theme.css` 中本应覆盖 `components.css` 的任何规则将停止工作。

样式提取与准备过程

当路由器获取页面时，它会提取所有与样式相关的元素：包括 `<link rel="stylesheet">` 标签和内联 `<style>`
块。每个样式元素通过其属性组合（对于 `<link>` 标签主要是 `href`）或其内容哈希值（对于内联 `<style>`
块）进行标识。

然后路由器将提取的样式与当前页面文档中已存在的样式进行比较。样式分为三类：

1. 已存在 ：样式表已在当前页面加载。准备阶段无需操作。
2. 新增 ：样式表在当前页面中不存在。需要添加。
3. 不再需要 ：样式表在当前页面中但不在目标页面中。将在导航期间禁用。

预加载新样式但不应用

对于新样式表，路由器面临一个困境：它需要确保在显示新页面内容前样式已完全加载（防止无样式内容闪烁
），但又不想立即应用它们（因为用户仍在查看当前页面）。

解决方案是添加新的 <link> 元素，并将其 media 属性设置为阻止应用的值。路由器使用 media="preload"
，这告诉浏览器 "此样式表不适用于任何媒体类型 "——在允许浏览器下载和解析的同时有效禁用它。

当以这种方式添加 <link> 元素时，浏览器会立即开始下载 CSS 文件。路由器通过监听 load
事件来跟踪每个样式表何时完成加载。这使得它能够等待所有新样式准备就绪后再继续导航。

使用最短公共超序列算法维护层叠顺序

在插入新样式表时，路由器必须保持正确的层叠顺序。它通过一种基于寻找两个序列的最短公共超序列（ SCS ）的算法来实现这一目标。

给定当前页面的样式表（序列 X）和目标页面的样式表（序列 Y）， SCS 算法会找到包含 X 和 Y
作为子序列的最短序列，同时保持它们内部的顺序。这能明确告知路由器新元素应插入的位置以及哪些现有元
素需要保留。

例如：

当前页面样式（ X）： [A, C, D]
目标页面样式（ Y）： [A, B, C, E]
最短公共超序列： [A, B, C, D, E]

算法随后确定：保持 A 和 C 在原位，在 A 和 C 之间插入 B，在 C 之后保留 D，并在末尾插入 E。

这种方法确保了：

同时出现在两个页面中的样式表保持正确的相对顺序
新样式表被插入到适当位置以维持层叠顺序的正确性
执行最少数量的 DOM 操作

在导航期间激活和停用样式

当 navigate() 实际渲染新页面时，路由器会切换样式表的启用和停用状态：

激活样式 ：对于属于目标页面的每个样式表，路由器移除 media="preload"
覆盖（或恢复原始指定的 media 属性）。这会使浏览器应用这些样式。
停用样式 ：对于存在于当前页面但不在目标页面中的每个样式表，路由器将其 media 属性设置为
"preload"。这会在不将元素从 DOM 中移除的情况下禁用样式。

通过将已停用的样式元素保留在 DOM 中（而不是移除它们），路由器可以在用户导航返回时快速重新激活它们。样式已经加载并解析完毕；只需重新启用即可。

脚本模块处理

交互性 API 使用 脚本模块 来实现交互行为。路由器必须确保在导航到新页面时，所需的脚本模块被加载并执行。

识别客户端导航所需的脚本模块

并非所有脚本模块都应在客户端导航期间加载。有些模块可能用于管理功能，或仅适用于初始页面加载的特性。如 入门指南 部分所述，WordPress 使用 `data-wp-router-options` 属性来标记哪些脚本模块应在导航期间加载：

```
<script
type="module"
src="/wp-content/plugins/my-plugin/view.js"
data-wp-router-options='{"loadOnClientNavigation": true}'
></script>
```

当路由器获取页面时，它会扫描所有具有此属性且 `loadOnClientNavigation` 设置为 `true` 的 `<script type="module">` 元素。这些就是它将预加载和执行的模块。

处理导入映射

现代 JavaScript 使用导入映射将裸模块说明符（如 `@wordpress/interactivity`）解析为实际 URL。WordPress 会生成一个导入映射，告诉浏览器在哪里可以找到每个模块：

```
<script type="importmap">
{
"imports": {
"@wordpress/interactivity": "/wp-includes/js/dist/interactivity.min.js",
"@wordpress/interactivity-router": "/wp-includes/js/dist/interactivity-router.min.js"
}
}
</script>
```

当路由器获取新页面时，它会从该页面提取导入映射，并将任何新的映射与当前页面的导入映射合并。这确保了即使在具有不同脚本集的页面之间导航时，脚本模块也能正确解析其依赖关系。

预加载脚本模块及其依赖项

预加载脚本模块需要解析其完整的依赖树，因为单个入口点模块可能依赖于数十个其他脚本模块，而这些模块又可能依赖于更多模块。

为了处理这个问题，路由器执行递归依赖解析：

1. 它获取每个入口点脚本模块的源代码
2. 它解析源代码以查找所有 `import` 语句
3. 对于每个导入，它使用导入映射解析脚本模块说明符
4. 它递归地获取并解析每个依赖项
5. 此过程持续进行，直到依赖树中的所有脚本模块都被获取

路由器会智能地避免冗余工作。如果一个脚本模块已经被初始页面加载（它出现在初始导入映射中），路由器不会再次获取它——浏览器已经将其缓存。

处理导入时机

一个重要的细节是，脚本模块代码不应在实际导航发生前执行。路由器需要准备好脚本模块代码（以避免导航期间的延迟），但不应在用户仍在查看当前页面时运行该代码。

路由器通过转换获取的脚本模块来实现这一点。它重写源代码以使用 blob URL（直接嵌入 URL 中的数据），并缓存这些转换后的脚本模块。当导航发生时，路由器使用动态 `import()` 来执行缓存的脚本模块。

由于浏览器的模块系统按 URL 缓存脚本模块，多次导入相同的 blob URL 会返回相同的模块实例。这确保了每个脚本模块只执行一次，即使多个代码路径尝试导入它。

导航期间的脚本模块执行

当 `navigate()` 渲染新页面时，它会导入为该页面预加载的所有脚本模块：

```
// 简化的概念视图，展示实际发生的过程。
for ( const moduleInfo of page.scriptModules ) {
await import( moduleInfo.blobUrl );
}
```

每个脚本模块的顶层代码都会运行，这通常包括调用 `store()` 来注册操作、回调和状态。由于交互性 API 的存储是全局且可累加的，这些注册会与初始页面加载时已有的存储定义合并。

Server state and context

Interactive elements often need data from the server — configuration values, content from the database, user preferences, and more. The Interactivity API provides three mechanisms for this: global state, local context and config.

During client-side navigation, this server-provided data needs to be extracted from the new page and made available to the client-side code.

How server data is embedded in pages

When WordPress renders a page with interactive elements, it embeds server-provided data in special `<script>` tags:

```
<!-- Global state -->
<script
type="application/json"
id="wp-script-module-data-@wordpress/interactivity"
>
{
  "state": {
    "myPlugin": {
      "cartItemCount": 3,
    }
  }
  "config": {
    "myPlugin": {
      "userLoggedIn": true
    }
  }
}
</script>
```

Local context is embedded directly in the `data-wp-context` attribute of elements:

```
<div
data-wp-interactive="myPlugin"
data-wp-context='{ "productId": 42, "inStock": true }'
>
<!-- Content -->
</div>
```

Extracting state, context and config during fetch

When the router fetches a new page, it extracts these types of server data:

1. **Global state**: The router finds the `<script type="application/json">` element with ID `wp-script-module-data-@wordpress/interactivity` and parses its JSON content to extrat its `state` property. This state comes from `wp_interactivity_state` is stored in the internal in-

memory page cache entry.
2. **Local context**: Context values are embedded in the virtual DOM representation of each router region. When a region's HTML is converted to vDOM, the `data-wp-context` attributes are preserved and will be processed during rendering.
3. **Config**: The router finds the `<script type="application/json">` element with ID `wp-script-module-data-@wordpress/interactivity` and parses its JSON content to extrat its `config` property. This configuration comes from `wp_interactivity_config` and is stored in the internal in-memory page cache entry.

Merging server data during navigation

When navigation renders the new page, the server-provided data may need to merge with the existing client-side state, depending on the use case. The key principle here is that **client-side state is never automatically overwritten by the server**. This design ensures that any changes your JavaScript code has made to the state (such as user preferences, UI toggles, or form input) are preserved across navigations.

对于 全局状态
，合并机制如下：客户端已存在的属性保持不变，仅从服务器数据中添加新属性（即客户端尚未存在的属性）
。若需要在导航期间让客户端状态反映服务器变更，请使用 `getServerState()`
订阅服务器提供的值并自行更新客户端状态。

初始页面的服务器状态：

```
{ "totalResults": 120, "isFiltersOpen": false }
```

导航前（用户打开筛选器，修改客户端状态）：

```
getServerState()  → { "totalResults": 120, "isFiltersOpen": false }
state             → { "totalResults": 120, "isFiltersOpen": true }
```

新页面的服务器状态：

```
{ "totalResults": 85, "sortOrder": "date" }
```

导航后：

```
getServerState()  → { "totalResults": 85, "sortOrder": "date" }
state             → { "totalResults": 120, "isFiltersOpen": true, "sortOrder": "date" }
```

`state` 中的 `totalResults` 保持为 `120` 是因为该属性已存在于客户端。 `isFiltersOpen` 同样被保留。
`sortOrder` 被添加是因为客户端原先不存在该属性。与此同时， `getServerState()`
始终精确反映服务器为新页面发送的数据。

对于 局部上下文 ，其行为遵循相同原则。交互性 API
会分别追踪服务器上下文和客户端上下文。导航期间，服务器上下文会更新为新页面的值，但客户端上下文保
持不变。使用 `getServerContext()` 读取服务器提供的值，使用 `getContext()`
读取客户端值，根据具体使用场景选择合适的方法。

初始页面的服务器上下文：

```
{ "isAvailable": true, "isLiked": false }
```

导航前（用户已点赞项目，修改客户端上下文）：

```
getServerContext() → { "isAvailable": true, "isLiked": false }
getContext()       → { "isAvailable": true, "isLiked": true }
```

新页面的服务器上下文：

```
{ "isAvailable": false, "discount": 15 }
```

导航后：

```
getServerContext() → { "isAvailable": false, "discount": 15 }
getContext()       → { "isAvailable": true, "isLiked": true, "discount": 15 }
```

`getContext()` 中的 `isAvailable` 保持为 `true` 是因为该属性已存在于客户端。 `isLiked` 同样被保留。 `discount` 被添加是因为客户端原先不存在该属性。与此同时， `getServerContext()` 始终精确反映服务器为新页面发送的数据。

订阅服务器数据变更

交互性 API 提供两个函数用于访问导航期间更新的服务器数据：

- `getServerState()`：返回服务器为当前页面提供的全局状态
- `getServerContext()`：返回服务器为当前元素提供的局部上下文

这些函数是响应式的。当在回调函数或派生状态获取器内部使用时，它们会自动建立订阅。当导航发生且新的服务器数据到达时，任何使用这些函数的代码都会使用新值重新运行。

这与常规的 `state` 和 `getContext()` 不同，后者返回的是客户端状态和上下文。如上所述，现有客户端值在导航期间不会被覆盖，因此 `state` 和 `getContext()` 会持续反映客户端在导航之前的状态。当你需要对服务器为新页面发送的值做出响应时，请使用 `getServerState()` 和 `getServerContext()`。

更多详细信息，请参阅 理解全局状态、本地上下文和派生状态 指南。

整合应用：导航流程

现在我们已经了解了各个组件，让我们追踪一次完整的导航过程，看看它们是如何协同工作的。

第一阶段：预获取（可选但推荐）

当调用 `prefetch()` 时（例如，在链接悬停时）：

1. 路由器规范化 URL 并检查页面缓存。
2. 如果未缓存，则开始获取 HTML。
3. 获取到的 HTML 被解析成文档。
4. 提取路由器区域并转换为虚拟 DOM。
5. 样式表与先前加载的进行比较；新的样式表会以 media="preload" 的方式添加。
6. 识别脚本模块并与先前加载的进行比较；新的模块会解析其依赖项并获取源代码。
7. 提取服务器状态。
8. 完全处理后的页面被存储在缓存中。
9. 函数返回（页面现在已准备好进行即时导航）。

第二阶段：导航

当调用 navigate() 时（例如，点击链接后）：

1. 路由器检查客户端导航是否被禁用；如果是，则回退到完整页面加载。
2. 如果尚未预取，则立即运行第一阶段中的获取过程。
3. 路由器等待页面准备就绪（获取完成，样式加载）。
4. 如果等待时间超过阈值（ 400ms），可能会显示加载指示器。
5. 渲染阶段开始（为提高效率而进行批量处理）：
执行新页面的脚本模块。
服务器状态与客户端状态合并。
每个路由器区域使用其新的虚拟 DOM 进行更新。
创建并附加带有 attachTo 但尚不存在的区域。
根据需要激活 /停用样式。
更新文档标题。
6. 更新浏览器历史记录（ pushState 或 replaceState）。
7. 为无障碍访问进行屏幕阅读器播报。
8. 如果 URL 包含哈希，页面将滚动到该元素。
9. 导航完成。

竞态条件防护

一个微妙但重要的细节：用户并不总是等待导航完成就点击另一个链接。路由器会优雅地处理这种情况。

当调用 navigate() 时，路由器会记住目标 URL。如果在第一次导航完成前又调用了另一个 navigate() ，路由器会更新其目标并放弃第一次导航。当第一次导航的 fetch 完成时，它会检查其 URL 是否仍是当前目标 — — 如果不是，则直接返回而不渲染。

这确保了快速点击多个链接不会导致视觉故障或渲染过时内容。只有最近的导航会完成。

全页面客户端导航（实验性功能）
全页面客户端导航是一项实验性功能，它扩展了本指南中描述的基于区域的方法。与要求您定义独立的路由器
区域不同，全页面导航将整个 <body> 元素视为单个区域 ——在导航过程中有效地替换所有页面内容。

此功能仅在 Gutenberg 插件中可用，且必须手动启用。要激活它，请前往 **WP** 管理后台 **> Gutenberg >** 实验功能，并勾选 **"Interactivity API:** 全页面客户端导航 **"** 选项。

启用后，此模式会自动拦截页面上的所有链接点击和悬停事件，触发客户端导航和预获取，而无需您编写任何自定义操作处理程序。它通过路由器包中的一个独立入口点提供：

```
import '@wordpress/interactivity-router/full-page';
```

全页面客户端导航本质上是基于区域导航的一个特例，其中只有一个覆盖整个页面的区域。由于它会替换所有内容，页面上的每个交互元素都必须使用 **Interactivity API**（而非 jQuery 或其他库），客户端导航才能正常工作。

> 此功能为实验性功能，仍在积极开发中。它可能无法在所有场景下正常工作。如果您尝试使用，请在 Gutenberg GitHub 仓库__ 中报告遇到的任何问题。也欢迎贡献代码！

版本号 #2
由 壳壳虫 创建于 2026-03-26 17:08:43 UTC
由 壳壳虫 更新于 2026-03-26 17:08:43 UTC