[illegible]

[illegible]

- [illegible]
  - [illegible]
  - [illegible]
  - [illegible] bug [illegible]
  - [illegible]
- [illegible]
  - [illegible]
  - Mutability vs immutability
  - Reactive side effects
- [illegible]

Interactivity API [illegible] React[illegible] Vue[illegible] Svelte [illegible] Alpine [illegible] Interactivity API [illegible]
[illegible] Interactivity API [illegible]

[illegible]

[illegible] [illegible] [illegible]
[illegible] [illegible]
[illegible] [illegible] [illegible]

[illegible]

[illegible] Web [illegible] JavaScript [illegible] DOM [illegible]

[illegible]

- [illegible] /[illegible] [illegible] /[illegible] "[illegible] "[illegible]
- [illegible] /[illegible] [illegible] "[illegible] "[illegible] "[illegible] "[illegible]

```
<div id="my-interactive-plugin">
	<button
		id="show-hide-btn"
		aria-expanded="false"
		aria-controls="status-paragraph"
	>
		show
	</button>
	<button id="activate-btn" disabled>activate</button>
	<p id="status-paragraph" class="inactive" hidden>this is inactive</p>
</div>

<style>
	.active {
		color: green;
	}
	.inactive {
		color: red;
	}
</style>

<script>
	const showHideBtn = document.getElementById( 'show-hide-btn' );
	const activateBtn = document.getElementById( 'activate-btn' );
	const statusParagraph = document.getElementById( 'status-paragraph' );

	showHideBtn.addEventListener( 'click', () => {
		if ( statusParagraph.hasAttribute( 'hidden' ) ) {
			statusParagraph.removeAttribute( 'hidden' );
			showHideBtn.textContent = 'hide';
			showHideBtn.setAttribute( 'aria-expanded', 'true' );
			activateBtn.removeAttribute( 'disabled' );
		} else {
			if ( statusParagraph.classList.contains( 'active' ) ) {
				statusParagraph.textContent = 'this is inactive';
				statusParagraph.classList.remove( 'active' );
				activateBtn.textContent = 'activate';
			}
			statusParagraph.setAttribute( 'hidden', true );
			showHideBtn.textContent = 'show';
```

```
□□□showHideBtn.setAttribute( 'aria-expanded', 'false' );
□□□activateBtn.setAttribute( 'disabled', true );
□□}
□} );

□activateBtn.addEventListener( 'click', () => {
□□if ( activateBtn.textContent === 'activate' ) {
□□□statusParagraph.textContent = 'this is active';
□□□statusParagraph.classList.remove( 'inactive' );
□□□statusParagraph.classList.add( 'active' );
□□□activateBtn.textContent = 'deactivate';
□□} else {
□□□statusParagraph.textContent = 'this is inactive';
□□□statusParagraph.classList.remove( 'active' );
□□□statusParagraph.classList.add( 'inactive' );
□□□activateBtn.textContent = 'activate';
□□}
□} );
</script>
```

□□□□□□□□□□□□□□□□□□□□ JavaScript □□□ DOM □□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□

□□□□□□□□□ □□□□□□ □□□□□□ UI □□□□□□□□□□□□□□□□□□□□□□ API □□□□□□□□□□□

```
<div id="my-interactive-plugin" data-wp-interactive="myInteractivePlugin">
□<button
□□data-wp-on--click="actions.toggleVisibility"
□□data-wp-bind--aria-expanded="state.isVisible"
□□data-wp-text="state.visibilityText"
□□aria-controls="status-paragraph"
□>
□□show
□</button>
□<button
□□data-wp-on--click="actions.toggleActivation"
□□data-wp-bind--disabled="!state.isVisible"
```

```
□□data-wp-text="state.activationText"
□>
□□activate
□</button>
□<p
□□id="status-paragraph"
□□data-wp-bind--hidden="!state.isVisible"
□□data-wp-class--active="state.isActive"
□□data-wp-class--inactive="!state.isActive"
□□data-wp-text="state.paragraphText"
□>
□□this is inactive
□</p>
</div>

<style>
□.active {
□□color: green;
□}
□.inactive {
□□color: red;
□}
</style>
```

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

const { state } = store( 'myInteractivePlugin', {
□state: {
□□isVisible: false,
□□isActive: false,
□□get visibilityText() {
□□□return state.isVisible ? 'hide' : 'show';
□□},
□□get activationText() {
□□□return state.isActive ? 'deactivate' : 'activate';
□□},
□□get paragraphText() {
□□□return state.isActive ? 'this is active' : 'this is inactive';
□□},
□},
```

```
actions: {
toggleVisibility() {
state.isVisible = ! state.isVisible;
if ( ! state.isVisible ) state.isActive = false;
},
toggleActivation() {
state.isActive = ! state.isActive;
},
},
} );
```

在这个声明式示例中，UI 会根据当前状态自动更新。作为开发者，您只需声明必要的状态、任何派生状态、修改状态的操作，以及 DOM 的哪些部分依赖于状态的哪些部分。框架会负责对 DOM 进行所有必要的更新，使其始终与当前状态保持同步。无论框架控制的元素数量有多少，逻辑都保持简单且易于维护。

你能发现这个 bug 吗?

在命令式示例中，为了教学目的故意引入了一个 bug。你能找到它吗？这并不容易！

查看答案!

如果先按下 Show 按钮，然后按下 Activate 按钮，最后按下 Hide 按钮，它不会使用 `statusParagraph.classList.add('inactive');` 添加 `inactive` 类。因此，当用户下次按下 Show 时，段落将不会显示为红色。

这类 bug 在命令式代码中非常常见，因为你需要手动控制所有条件。另一方面，它们在声明式代码中不存在，因为框架会负责更新 DOM，永远不会遗漏任何东西。

声明式方法的优势

如示例所示，命令式方法需要详细步骤并直接操作 DOM，随着交互复杂度的增加，这种方法会迅速变得复杂且难以维护。可能的状态和元素越多，需要添加的条件逻辑就越多，导致代码复杂度呈指数级增长。相比之下，声明式方法通过管理状态并让框架处理 DOM 更新来简化流程，从而产生更易读、易维护和可扩展的代码。

响应式

交互性 API

是一个声明式框架，这得益于其对响应式特性的运用。在响应式系统中，数据的变化会自动触发用户界面的更新，确保视图始终反映应用程序的当前状态。

响应式系统的工作原理

Interactivity API 采用细粒度的响应式系统。以下是其工作原理：

1. 响应式状态：在 Interactivity API 中，全局状态和局部上下文都是响应式的。这意味着当这些数据源发生变化时，依赖于它们的任何 UI 部分都会自动更新。
   - 全局状态：这是可以在整个交互块中访问的全局数据。
   - 局部上下文：这是特定于某个元素及其子元素的局部数据。
   - 派生状态：除了基本状态属性外，您还可以定义计算属性，这些属性会在其依赖项变化时自动更新。

   请访问 理解全局状态、局部上下文、派生状态和配置 指南，了解更多关于如何在 *Interactivity API* 中使用不同类型的响应式状态。
2. 操作：这些是通常由事件处理程序触发的函数，用于改变全局状态或局部上下文。
3. 响应式绑定：HTML 元素通过特殊属性（如 `data-wp-bind`、`data-wp-text` 或 `data-wp-class`）绑定到响应式状态值。
4. 自动更新：当操作改变全局状态或局部上下文时，Interactivity API 会自动更新依赖于该状态（直接或通过派生状态）的所有 DOM 部分。

让我们通过回顾前面的示例来分解这些概念：

```
const { state } = store( 'myInteractivePlugin', {
  state: {
    isVisible: false,
    isActive: false,
    get visibilityText() {
      return state.isVisible ? 'hide' : 'show';
    },
    // ...其他派生状态
  },
  actions: {
    toggleVisibility() {
      state.isVisible = ! state.isVisible;
    },
    // ...其他操作
```

```
},
} );
```

□□□□□□□

- `isVisible` □ `isActive` □□□□□□□□
- `visibilityText` □□□□□□□ `isVisible` □□□□□□□□□
- `toggleVisibility` □□□□□□□□□

HTML □□□□□□□

```
<button
□data-wp-on--click="actions.toggleVisibility"
□data-wp-text="state.visibilityText"
□data-wp-bind--aria-expanded="state.isVisible"
>
□show
</button>
```

□□□□□□□□□□□□□□□□□□□

1. □□□□□□□□□□□□ `toggleVisibility` □□□
2. □□□□□ `state.isVisible` □
3. Interactivity API □□□□□□□□□□
   - □□□□□□□□□□□□ `data-wp-text="state.visibilityText"` □□
   - □□ `aria-expanded` □□□□□ `data-wp-bind--aria-expanded="state.isVisible"` □□
   - □□□□□ `isVisible` □ `visibilityText` □□□□□ DOM □□□

Mutability vs immutability

Unlike many other reactive frameworks, **the Interactivity API does not require the use of immutability** when updating the global state or the local context. You can directly mutate objects and arrays, and the reactivity system will still work as expected. This can lead to more intuitive and straightforward code in many cases.

For example, you can push a new item to an array like this:

```
const { state } = store( 'myArrayPlugin', {
□state: {
□□list: [ 'item 1', 'item 2' ],
□},
□actions: {
```

```
		addItem() {
			// Right:
			state.list.push( 'new item' );

			// Wrong:
			state.list = [ ...state.list, 'new item' ]; // Don't do this!
		},
	},
} );
```

There's no need to create a new array or use the spread operator as you might in other frameworks. The Interactivity API will detect this change and update any parts of the UI that depend on `state.list`.

Reactive side effects

In addition to automatically updating the UI, the Interactivity API allows you to perform side effects when reactive data changes using directives like `data-wp-watch`. Side effects are useful for tasks like logging, making API calls, or updating other parts of your application that aren't directly tied to the UI.

Here's an example of how you might use `data-wp-watch`:

```
<div
	data-wp-interactive="myCounterPlugin"
	data-wp-context='{ "counter": 0 }'
	data-wp-watch="callbacks.logCounter"
>
	<p>Counter: <span data-wp-text="context.counter"></span></p>
	<button data-wp-on--click="actions.increment">Increment</button>
</div>
```

```
store( 'myCounterPlugin', {
	actions: {
		increment() {
			const context = getContext();
			context.counter += 1;
		},
	},
	callbacks: {
```

```
□□logCounter: () => {
□□□const context = getContext();
□□□console.log( `The counter is now: ${ context.counter }` );
□□},
□},
} );
```

In this example:

1. The `data-wp-context` directive adds a local context with a property `counter` whose value is `0`.
2. The `data-wp-watch` directive is set to `callbacks.logCounter`.
3. Every time `context.counter` changes, the `logCounter` callback will be executed.
4. The `logCounter` callback logs the current counter to the console.

This allows you to create declarative side effects that automatically run in response to data changes. Some other use cases for `data-wp-watch` might include:

- Saving data to `localStorage` when the data changes.
- Sending analytics events.
- Changing the focus for accessibility purposes.
- Updating the page title, meta tags, or `<body>` attributes.
- Triggering animations.

总结

当你继续使用交互性 API 时，请记住从状态、操作和副作用的角度思考。定义你的数据，描述它应如何变化，然后让交互性 API 处理其余部分。这种思维转变可能需要一些时间，特别是如果你习惯了更命令式的编程风格，但通过接受它，你将释放交互性 API 的全部潜力，创造出真正动态且交互式的 WordPress 区块，让你的用户感到愉悦。

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