测试概览

目录
为何需要测试？
JavaScript 测试
文件夹结构
导入测试
Describing tests
设置与清理方法
模拟依赖项
依赖注入
Imported dependencies
测试全局对象
User interactions
区块 UI 集成测试
快照测试
太长不看版：快照测试失败时
什么是快照？
优势
劣势
使用场景
使用快照
痛点
最佳实践
故障排除
调试 Jest 单元测试
原生移动端测试
调试原生移动端单元测试
原生移动端到端测试
原生移动端集成测试
端到端测试

使用 wp-env
场景测试
核心区块测试
不稳定测试
PHP 测试
测试带前缀的函数
Writing Tests for Prefixed Functions & Classes
性能测试

Gutenberg 包含 PHP 和 JavaScript 代码，并鼓励对两者进行测试和代码风格检查。

为何需要测试？
除了测试将为你的生活带来乐趣之外，测试的重要性不仅在于它们有助于确保我们的应用程序按预期运行，还
因为它们提供了如何使用代码的简明示例。

测试也是我们代码库的一部分，这意味着我们对它们采用与所有应用程序代码相同的标准。

与所有代码一样，测试也需要维护。为了有测试而编写测试并非目标 —— 相反，我们应该努力在覆盖预期和意外行为、快速执行和代码维护之间找到适当的平衡。

编写测试时请考虑以下几点：

我们正在测试什么行为？
运行此代码时可能出现哪些错误？
测试是否确实测试了我们认为的内容？还是我们引入了误报 /漏报？
测试是否可读？其他贡献者能否通过查看相应的测试来理解我们的代码行为？

JavaScript 测试
JavaScript 测试使用 Jest作为测试运行器，并使用其 API 来处理 全局函数 （ describe、 test、 beforeEach
等）、 断言 、模拟 、监视 和模拟函数 。如果需要，你也可以使用 React Testing Library进行 React
组件测试。

需要注意的是，过去 React 组件曾使用 Enzyme进行单元测试。然而，现在所有现有和新的测试都改用
React Testing Library (RTL)。

假设你已经按照 说明 安装了 Node 和项目依赖项，可以通过命令行使用 NPM 运行测试：

```
npm test
```

代码检查是一种静态代码分析，用于强制执行编码标准并避免潜在错误。本项目使用 ESLint 和 TypeScript 的 JavaScript 类型检查 来捕获这些问题。虽然上述 `npm test` 会同时执行单元测试和代码检查，但也可以通过运行 `npm run lint` 独立验证代码检查。运行 `npm run lint:js:fix` 可以自动修复一些 JavaScript 问题。

为了改善你的开发工作流程，你应该设置编辑器代码检查集成。更多信息请参阅 入门文档 。

如果只想运行单元测试而不运行代码检查器，请改用 `npm run test:unit` 。

文件夹结构

请将测试文件存放在工作目录的 `test` 文件夹中。测试文件应与被测文件同名。

```
+-- test
|   +-- bar.js
+-- bar.js
```

只有测试文件（至少包含一个测试用例）应直接放在 `/test` 目录下。如需添加外部模拟或固定数据，请将其置于子文件夹中，例如：

- `test/mocks/[file-name].js`
- `test/fixtures/[file-name].js`

导入测试

根据之前的文件夹结构，在导入 被测试代码 时尽量使用相对路径，而不是项目路径。

推荐做法

```
import { bar } from '../bar';
```

不推荐做法

```
import { bar } from 'components/foo/bar';
```

这样当您决定将代码移动到应用程序目录的其他位置时，会让您的工作更轻松。

Describing tests

Use a `describe` block to group test cases. Each test case should ideally describe one behaviour only.

In test cases, try to describe in plain words the expected behaviour. For UI components, this might entail describing expected behaviour from a user perspective rather than explaining code internals.

Good

```
describe( 'CheckboxWithLabel', () => {
    test( 'checking checkbox should disable the form submit button', () => {
        ...
    } );
} );
```

Not so good

```
describe( 'CheckboxWithLabel', () => {
    test( 'checking checkbox should set this.state.disableButton to `true`', () => {
        ...
    } );
} );
```

设置与清理方法

Jest API 包含一些实用的 设置与清理方法，允许你在每个或所有测试之前 /之后执行任务，或在特定 `describe` 代码块内的测试前后执行操作。

这些方法可以处理异步代码，以便执行通常无法内联完成的设置操作。与 单个测试用例类似，你可以返回 Promise，Jest 会等待其解析：

```
// 为*所有*测试执行一次性设置
beforeAll( () =>
  someAsyncAction().then( ( resp ) => {
    window.someGlobal = resp;
  } )
);

// 为*所有*测试执行一次性清理
```

```
afterAll( () => {
  window.someGlobal = null;
} );
```

`afterEach` `afterAll`

" "


```
import VALID_VALUES_LIST from './constants';

function isValueValid( value ) {
  return VALID_VALUES_LIST.includes( value );
}
```

`VALID_VALUES_LIST`

`expect( isValueValid( VALID_VALUES_LIST[ 0 ] ) ).toBe( true );`

1) 2) `VALID_VALUES_LIST`

`VALID_VALUES_LIST` HTTP `isValueValid`

```
function isValueValid( value, validValuesList = [] ) {
  return validValuesList.includes( value );
}
```

`validValuesList`

`expect( isValueValid( 'hulk', [ 'batman', 'superman' ] ) ).toBe( false );`

`expect( isValueValid( 'hulk', null ) ).toBe( false );`

`expect( isValueValid( 'hulk', [] ) ).toBe( false );`

`expect( isValueValid( 'hulk', [ 'iron man', 'hulk' ] ) ).toBe( true );`

Imported dependencies

Often our code will use methods and properties from imported external and internal libraries in multiple places, which makes passing around arguments messy and impracticable. For these cases `jest.mock` offers a neat way to stub these dependencies.

For instance, lets assume we have `config` module to control a great deal of functionality via feature flags.

```
// bilbo.js
import config from 'config';
export const isBilboVisible = () =>
	config.isEnabled( 'the-ring' ) ? false : true;
```

To test the behaviour under each condition, we stub the config object and use a jest mocking function to control the return value of `isEnabled`.

```
// test/bilbo.js
import { isEnabled } from 'config';
import { isBilboVisible } from '../bilbo';

jest.mock( 'config', () => ( {
	// bilbo is visible by default
	isEnabled: jest.fn( () => false ),
} ) );

describe( 'The bilbo module', () => {
	test( 'bilbo should be visible by default', () => {
		expect( isBilboVisible() ).toBe( true );
	} );

	test( 'bilbo should be invisible when the `the-ring` config feature flag is enabled', () => {
		isEnabled.mockImplementationOnce( ( name ) => name === 'the-ring' );
		expect( isBilboVisible() ).toBe( false );
	} );
} );
```

□□□□□□

□□□□□□ Jest □□ □□□□□□□□□□□□□

```
import { myModuleFunctionThatOpensANewWindow } from '../my-module';

describe( 'my module', () => {
□beforeAll( () => {
□□jest.spyOn( global, 'open' ).mockImplementation( () => true );
□} );

□test( 'something', () => {
□□myModuleFunctionThatOpensANewWindow();
□□expect( global.open ).toHaveBeenCalled();
□} );
} );
```

User interactions

Simulating user interactions is a great way to **write tests from the user's perspective**, and therefore avoid testing implementation details.

When writing tests with Testing Library, there are two main alternatives for simulating user interactions:

1. The `fireEvent` API, a utility for firing DOM events part of the Testing Library core API.
2. The `user-event` library, a companion library to Testing Library that simulates user interactions by dispatching the events that would happen if the interaction took place in a browser.

The built-in `fireEvent` is a utility for dispatching DOM events. It dispatches exactly the events that are described in the test spec - even if those exact events never had been dispatched in a real interaction in a browser.

On the other hand, the `user-event` library exposes higher-level methods (e.g. `type`, `selectOptions`, `clear`, `doubleClick`...), that dispatch events like they would happen if a user interacted with the document, and take care of any react-specific quirks.

For the above reasons, **the `user-event` library is recommended when writing tests for user interactions**.

Not so good: using `fireEvent` to dispatch DOM events.

```
import { render, screen } from '@testing-library/react';

test( 'fires onChange when a new value is typed', () => {
  const spyOnChange = jest.fn();

  // A component with one `input` and one `select`.
  render( <MyComponent onChange={ spyOnChange } /> );

  const input = screen.getByRole( 'textbox' );
  input.focus();
  // No clicks, no key events.
  fireEvent.change( input, { target: { value: 62 } } );

  // The `onChange` callback gets called once with '62' as the argument.
  expect( spyOnChange ).toHaveBeenCalledTimes( 1 );

  const select = screen.getByRole( 'listbox' );
  select.focus();
  // No pointer events dispatched.
  fireEvent.change( select, { target: { value: 'optionValue' } } );

  // ...
```

Good: using `user-event` to simulate user events.

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';

test( 'fires onChange when a new value is typed', async () => {
  const user = userEvent.setup();

  const spyOnChange = jest.fn();

  // A component with one `input` and one `select`.
  render( <MyComponent onChange={ spyOnChange } /> );

  const input = screen.getByRole( 'textbox' );
  // Focus the element, select and delete all its contents.
  await user.clear( input );
  // Click the element, type each character separately (generating keydown,
```

```
	// keypress and keyup events).
	await user.type( input, '62' );

	// The `onChange` callback gets called 3 times with the following arguments:
	// - 1: clear ('')
	// - 2: '6'
	// - 3: '62'
	expect( spyOnChange ).toHaveBeenCalledTimes( 3 );

	const select = screen.getByRole( 'listbox' );
	//
	await user.selectOptions( select, [ 'optionValue' ] );

	// ...
} );
```

UI

" "
Jest
UI

```
import { initializeEditor } from 'test/integration/helpers/integration-test-editor';

async function setup( attributes ) {
	const testBlock = { name: 'core/cover', attributes };
	return initializeEditor( testBlock );
}
```

`initializeEditor` `@testing-library/react` `render`

集成测试编辑器模块还导出了一个 `selectBlock` 函数，可用于通过区块包装器上的 aria-label 选择要测试的区块，例如 "Block: Cover"。

快照测试

这是关于 [快照测试]的概述以及如何最好地利用快照测试。

太长不看版：快照测试失败时

当快照测试失败时，仅仅意味着组件的渲染结果发生了变化。如果这个变化是无意的，那么快照测试就防止了一个 bug □□

然而，如果这个变化是有意的，请按照以下步骤更新快照。运行以下命令来更新快照：

```
# --testPathPattern 是可选的，但通过只运行匹配的测试会快得多
npm run test:unit -- --updateSnapshot --testPathPattern path/to/tests


# 更新端到端测试的快照
npm run test:e2e -- --update-snapshots path/to/spec
```

1. 检查差异，确保更改是预期且有意为之的。
2. 提交更改。

什么是快照?

快照是由测试生成的某些数据结构的表示形式。快照存储在文件中，并与测试一同提交。当测试运行时，生成的数据结构会与文件中的快照进行比较。

创建快照非常简单：

```
test( 'foobar test', () => {
□const foobar = { foo: 'bar' };

□expect( foobar ).toMatchSnapshot();
} );
```

这是生成的快照：

```
exports[ `test foobar test 1` ] = `
  Object {
    "foo": "bar",
  }
```

```
`;
```

你永远不应直接创建或修改快照，它们由测试生成和更新。

优势

- 添加测试简单明了。
- 防止意外更改。
- 易于使用。
- 无需运行应用程序即可揭示内部结构。

劣势

- 表达性不强。
- 仅在引入更改时发现问题。
- 对于任何非确定性内容都有问题。

使用场景

快照主要针对组件测试。它们让我们能够感知组件结构的变化，这使其成为重构的 _理想 _

工具。如果快照在一系列提交过程中保持更新，快照差异就能记录组件结构的演变过程。相当酷 □□

```
import { render, screen } from '@testing-library/react';
import SolarSystem from 'solar-system';

describe( 'SolarSystem', () => {
□test( 'should render', () => {
□□const { container } = render( <SolarSystem /> );

□□expect( container ).toMatchSnapshot();
□} );

□test( 'should contain mars if planets is true', () => {
□□const { container } = render( <SolarSystem planets /> );

□□expect( container ).toMatchSnapshot();
□□expect( screen.getByText( /mars/i ) ).toBeInTheDocument();
□} );
} );
```

Reducer 测试也非常适合使用快照。它们通常是庞大复杂的数据结构，不应该发生意外变化 ——

这正是快照的专长所在！

使用快照

当快照不匹配导致 CI 测试失败时，你可能会措手不及。如果更改是预期的，你需要 [更新快照] 。快速但粗糙的解决方案是使用 `--updateSnapshot` 调用 Jest。可以按如下方式操作：

```
npm run test:unit -- --updateSnapshot --testPathPattern path/to/tests
```

`--testPathPattern` 不是必需的，但指定路径将通过运行测试子集来加快速度。

工作时在后台保持运行 `npm run test:unit:watch` 是个好主意。 Jest 将仅运行与更改文件相关的测试，当快照测试失败时，只需按 `u` 即可更新快照！

痛点

非确定性测试可能无法生成一致的快照，因此需要谨慎对待。当处理任何随机、基于时间或其他非确定性的内容时，快照可能会出现问题。

处理已连接的组件比较棘手。要对已连接的组件进行快照，你可能需要导出未连接的组件：

```
// my-component.js
export { MyComponent };
export default connect( mapStateToProps )( MyComponent );

// test/my-component.js
import { MyComponent } from '..';
// 运行这些 MyComponent 测试…
```

需要手动提供已连接的属性。这是一个审查已连接状态的好机会。

最佳实践

如果你要开始重构，快照非常有用，你可以将它们作为分支上的第一个提交，并观察其演变过程。

快照本身并不表达我们的预期。快照最好与其他描述我们期望的测试结合使用，就像上面的例子：

```
test( '如果 planets 为 true 则应包含 mars', () => {
  const { container } = render( <SolarSystem planets /> );

  //快照将捕获意外更改
  expect( container ).toMatchSnapshot();

  //这才是我们实际期望在测试中找到的内容
  expect( screen.getByText( /mars/i ) ).toBeInTheDocument();
```

```
} );
```

另一个好技巧是使用 `toMatchDiffSnapshot` 函数（由 `snapshot-diff` 包提供），它允许仅对 DOM 两个不同状态之间的差异进行快照。这种方法适用于测试属性更改对结果 DOM 的影响，同时生成更小的快照，例如：

```
test( '当 isShady 为 true 时应渲染更暗的背景', () => {
	const { container } = render( <CardBody>Body</CardBody> );
	const { container: containerShady } = render(
		<CardBody isShady>Body</CardBody>
	);
	expect( container ).toMatchDiffSnapshot( containerShady );
} );
```

类似地，`toMatchStyleDiffSnapshot` 函数允许仅对组件两个不同状态关联的样式之间的差异进行快照，例如：

```
test( '应渲染边距', () => {
	const { container: spacer } = render( <Spacer /> );
	const { container: spacerWithMargin } = render( <Spacer margin={ 5 } /> );
	expect( spacerWithMargin ).toMatchStyleDiffSnapshot( spacer );
} );
```

故障排除

有时我们需要为某些使用引用的故事模拟引用。请查阅以下文档了解更多信息：

- 为什么我们需要在 React 中使用 模拟引用进行快照测试。

在这种情况下，您可能会看到测试失败，以及 Jest 在尝试从 `ref.current` 访问属性时报告 `TypeError`。

调试 Jest 单元测试

运行 `npm run test:unit:debug` 将在调试模式下启动测试，以便 node 检查器客户端可以连接到进程并检查执行情况。有关使用 Google Chrome 或 Visual Studio Code 作为检查器客户端的说明，请参阅 wp-scripts 文档。

原生移动端测试

单元测试套件包含一组 Jest 测试，用于运行基于 React Native 开发的原生移动端代码路径。由于这些测试在
Node 环境中运行，您可以在本地开发机上启动它们，无需特定的原生 Android 或 iOS 开发工具或 SDK 。这也意味着可以使用典型的开发工具进行调试。请继续阅读以下调试说明。

调试原生移动端单元测试

要在本地以调试模式运行测试，请按照以下步骤操作：

0. 确保已运行 npm ci 以安装所有包
1. 在 Gutenberg 根目录下的 CLI 中运行 npm run test:native:debug。此时 Node
正在等待调试器连接。
2. 在 Chrome 中打开 chrome://inspect
3. 在 "Remote Target" 部分，找到 ../../node_modules/.bin/jest 目标并点击 "inspect"
链接。这将打开一个新窗口，其中 Chrome DevTools 调试器已附加到进程，并停在 jest.js
文件的开头。或者，如果目标不可见，请点击同一页面上的 Open dedicated DevTools for Node
链接。
4. 您可以在代码（包括测试代码）中设置断点或 debugger; 语句来暂停执行并进行检查
5. 点击 "Play" 按钮以继续执行
6. 享受调试原生移动端单元测试的过程！

原生移动端到端测试

Gutenberg 贡献者会注意到， PR 包含在 Android 和 iOS 上运行原生移动端 E2E 测试的持续集成 E2E 测试。如需排查失败的测试，请查阅我们在持续集成中运行 原生移动测试的指南

。有关在本地运行这些测试的更多信息，请参见 此处 。

原生移动端集成测试

目前正在为原生移动端项目添加集成测试，使用 react-native-testing-library
库。编写集成测试的指南可在此处找到： 此处 。

端到端测试

端到端测试使用 Playwright作为测试框架。最佳实践和详细说明请参阅专门的 端到端测试指南 。

使用 wp-env

如果你正在使用内置的 本地环境，你可以使用以下命令在本地运行 e2e 测试：

```
npm run test:e2e
```

或者以交互模式运行

```
npm run test:e2e -- --ui
```

场景测试

如果你发现端到端测试在本地运行时通过，但在 GitHub Actions 中失败，可以通过模拟慢速 CPU 或网络来隔离 CPU 或网络相关的竞态条件：

```
THROTTLE_CPU=4 npm run test:e2e
```

`THROTTLE_CPU` 是减速因子（在此示例中为 4 倍减速乘数）

参见 Chrome 文档：setCPUThrottlingRate

```
SLOW_NETWORK=true npm run test:e2e
```

`SLOW_NETWORK` 模拟相当于 Chrome 开发者工具中 “快速 3G”的网络速度。

参见 Chrome 文档：emulateNetworkConditions 和 NetworkManager.js

```
OFFLINE=true npm run test:e2e
```

`OFFLINE` 模拟网络断开连接。

参见 Chrome 文档：emulateNetworkConditions

核心区块测试

每个核心区块必须至少包含一组用于主保存功能的测试夹具文件，以及每组弃用功能对应的测试夹具。这些夹具用于测试区块的解析和序列化功能。更多信息和操作说明请参阅 集成测试夹具自述文件。

不稳定测试

当某个测试在多次重试尝试中无需代码更改即可通过或失败时，即被视为 不稳定 测试。我们在 CI 上最多自动重试失败测试 两次，以检测它们并通过 `report-flaky-tests` GitHub Action 自动报告到 GitHub issues 的 `[Type] Flaky Test` 标签下。请注意，连续失败三次的测试不计为不稳定测试，也不会被报告到 issue 中。

PHP 测试

PHP 测试使用 PHPUnit 作为测试框架。如果你使用内置的 本地环境，可以通过以下命令在本地运行 PHP 测试：

```
npm run test:php
```

要在文件更改时自动重新运行测试（类似于 Jest），请运行：

```
npm run test:php:watch
```

注意：*phpunit* 命令要求 `wp-env` 正在运行且 *composer* 依赖已安装。如果 *wp-env* 尚未运行，包脚本会为你启动它。

在其他环境中，请运行 `composer run test` 和 `composer run test:watch`。

PHP 代码风格使用 PHP_CodeSniffer 强制执行。建议你使用 Composer 安装 PHP_CodeSniffer 和 WordPress Coding Standards for PHP_CodeSniffer 规则集。安装 Composer 后，在项目目录中运行 `composer install` 以安装依赖项。上述 `npm run test:php` 将同时执行单元测试和代码检查。可以通过运行 `npm run lint:php` 独立验证代码检查。

要仅运行单元测试而不运行代码检查，请改用 `npm run test:unit:php`。

测试带前缀的函数

Gutenberg 的构建系统会自动为 PHP 函数添加 `gutenberg_` 前缀，以避免与 WordPress 核心发生冲突。在为区块函数编写测试时，必须测试函数的 构建（带前缀）版本，而非源代码版本。

如果测试被反向移植到 WordPress 核心，则必须更新它们以测试不带前缀的函数。

Writing Tests for Prefixed Functions & Classes

Always test the built (prefixed) function names and class names in your PHPUnit tests:

```
// phpunit/blocks/my-block-test.php
class My_Block_Test extends WP_UnitTestCase {
```

```
    public function test_my_function() {
        // Test the built function (with gutenberg_ prefix)
        $result = gutenberg_block_core_my_block_render_function( $args );
        $this->assertEquals( $expected, $result );
    }

    public function test_my_class() {
        // Test the built class (with _Gutenberg suffix)
        $handler = new WP_Example_Block_Handler_Gutenberg();
        $result = $handler->process( $input );
        $this->assertEquals( $expected, $result );
    }
}
```

For more detailed information about the build system and function prefixing, see the Build System: Function Prefixing and Block Loading documentation.

性能测试

为确保编辑器在添加功能时保持高性能，我们监控拉取请求和版本发布对以下关键指标的影响：

- 编辑器加载所需时间
- 输入时浏览器响应时间
- 选择区块所需时间

性能测试是运行编辑器并捕获这些指标的端到端测试。请确保已准备好端到端测试环境。

要设置端到端测试环境，请检出 Gutenberg 仓库并切换到要测试的分支。运行以下命令准备环境：

```
nvm use && npm install
npm run build
```

运行以下命令执行测试：

```
npm run test:performance
```

这将给出当前运行环境中对应分支 /代码的结果。

此外，您还可以通过运行命令 `./bin/plugin/cli.js perf [branches]` 来比较不同分支（或标签、提交）之间的指标，例如：

```
./bin/plugin/cli.js perf trunk v8.1.0 v8.0.0
```

最后，您可以传递额外的 `--tests-branch` 参数来指定要运行哪个分支的性能测试文件。这在修改 / 扩展性能测试时特别有用：

```
./bin/plugin/cli.js perf trunk v8.1.0 v8.0.0 --tests-branch add/perf-tests-coverage
```

注意

此命令执行基准测试可能需要较长时间。运行期间请尽量避免使用计算机或运行大量后台进程，以尽量减少可能影响跨分支结果的外部因素。

版本号 #2
由 壳壳虫 创建于 2026-03-27 01:08:43 UTC
由 壳壳虫 更新于 2026-03-27 01:08:43 UTC