

VITEST Docs - 简体中文



Table of contents

• Cn - README	3
• Api - Index	4
• Config - Index	56
• Guide - Cli	74
• Guide - Comparisons	77
• Guide - Coverage	79
• Guide - Debugging	81
• Guide - Environment	83
• Guide - Extending matchers	85
• Guide - Features	87
• Guide - Filtering	91
• Guide - Ide	93
• Guide - In source	97
• Guide - Index	99
• Guide - Migration	103
• Guide - Mocking	106
• Guide - Snapshot	115
• Guide - Test context	119
• Guide - Testing types	121
• Guide - Ui	123
• Guide - Using plugins	125
• Guide - Why	126



Vitest

由 Vite 提供支持的极速单元测试框架

[中文文档](#) | [英文文档](#)

[代码仓库](#)

感谢

[@elonehoo](#) - 发起与维护该文档翻译

[Go to TOC](#)

API 索引

下面的变量中使用了以下类型签名

```
type Awaitable<T> = T | PromiseLike<T>;
type TestFunction = () => Awaitable<void>;

interface TestOptions {
  timeout?: number;
  retry?: number;
}
```

当一个测试函数返回一个 promise 时, Vitest 将等待直到它被解决以收集异步的期望值。如果 promise 被拒绝, 测试将失败。

提示 在 Jest 中, `TestFunction` 也可以是 `(done: DoneCallback) => void` 类型。如果使用此选项, 则在调用 `done` 之前测试不会结束。你可以使用 `async` 函数实现相同的目的, 请参阅迁移指南中的[回调]([../guide/migration#done-callback](#))部分。

test

- 类型:** `(name: string, fn: TestFunction, timeout?: number | TestOptions) => void`
- 别名:** `it`

`test` 定义了一组关于测试期望的方法。它接收测试名称和一个含有测试期望的函数。

同时, 可以提供一个超时时限 (以毫秒为单位) 用于指定等待多长时间后终止测试, 默认为 5 秒。你也可以通过 `testTimeout` 选项进行全局配置。

```
import { expect, test } from "vitest";

test("should work as expected", () => {
  expect(Math.sqrt(4)).toBe(2);
});
```

test.skip

- 类型:** `(name: string, fn: TestFunction, timeout?: number | TestOptions) => void`
- 别名:** `it.skip`

如果你想跳过运行某些测试, 但由于一些原因不想删除代码, 你可以使用 `test.skip` 来避免运行它们。

```
import { assert, test } from "vitest";

test.skip("skipped test", () => {
  // 跳过测试, 没有错误
  assert.equal(Math.sqrt(4), 3);
});
```

test.skipIf

- **类型:** `(condition: any) => Test`
- **别名:** `it.skipIf`

在某些情况下，你可能会在不同的环境中多次运行测试，并且某些测试可能基于特定环境下运行。只要条件成立，你就可以使用 `test.skipIf` 跳过测试，而不是用 `if` 包裹测试代码。

```
import { assert, test } from "vitest";

const isDev = process.env.NODE_ENV === "development";

test.skipIf(isDev)("prod only test", () => {
  // 只在生产环境下进行测试
});
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

test.runIf

- **类型:** `(condition: any) => Test`
- **别名:** `it.runIf`

`test.skipIf` 的相反面。

```
import { assert, test } from "vitest";

const isDev = process.env.NODE_ENV === "development";

test.runIf(isDev)("dev only test", () => {
  // 只在开发环境下进行测试
});
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

test.only

- **类型:** `(name: string, fn: TestFunction, timeout?: number) => void`
- **别名:** `it.only`

使用 `test.only` 只允许特定条件的测试套件。这在调试模式下是很有用的。

同时，可以提供一个超时时限（以毫秒为单位）用于指定等待多长时间后终止测试，默认为 5 秒。你也可以通过 `testTimeout` 选项进行全局配置。

```
import { assert, test } from "vitest";

test.only("test", () => {
```

```
// 仅运行此测试（以及仅标记有的其他测试）
assert.equal(Math.sqrt(4), 2);
});
```

有时候只运行一个特定文件中的 `only` 测试用例是很有用的，这可以忽略整个测试套件中的所有其他测试用例，避免污染输出。

为了做到这一点，运行 `vitest` 命令并带上包含有问题的测试的特定文件。

```
# vitest interesting.test.ts
```

test.concurrent

- 类型:** `(name: string, fn: TestFunction, timeout?: number) => void`
- 别名:** `it.concurrent`

`test.concurrent` 将连续测试标记为并行运行。它接收测试名称、带有要收集的测试的异步函数以及可选的超时时间（以毫秒为单位）。

```
import { describe, test } from "vitest";

// 标有并发的两个测试将并发运行
describe("suite", () => {
  test("serial test", async () => {
    /* ... */
  });
  test.concurrent("concurrent test 1", async () => {
    /* ... */
  });
  test.concurrent("concurrent test 2", async () => {
    /* ... */
  });
});
```

`test.skip`、`test.only` 和 `test.todo` 适用于并发测试。以下所有组合均有效：

```
test.concurrent(/* ... */);
test.skip.concurrent(/* ... */); // or test.concurrent.skip(/* ... */)
test.only.concurrent(/* ... */); // or test.concurrent.only(/* ... */)
test.todo.concurrent(/* ... */); // or test.concurrent.todo(/* ... */)
```

在异步并发测试中使用快照时，由于 JavaScript 的限制，你需要使用 [测试环境](#) 中的 `expect` 来确保检测到正确的测试。

```
test.concurrent("test 1", async ({ expect }) => {
  expect(foo).toMatchSnapshot();
});
test.concurrent("test 2", async ({ expect }) => {
  expect(foo).toMatchSnapshot();
});
```

You cannot use this syntax, when using Vitest as [type checker] (/guide/testing-types).

test.todo

- **类型:** `(name: string) => void`
- **别名:** `it.todo`

使用 `test.todo` 来存根测试，以便稍后实现。测试报告中将显示一个条目，以便告知你还有多少测试未实现。

```
// 测试的报告中将显示一个记录
test.todo("unimplemented test");
```

testfails

- **类型:** `(name: string, fn: TestFunction, timeout?: number) => void`
- **别名:** `it.fail`

使用 `test.fail` 来指示测试断言将显式失败。

```
import { expect, test } from "vitest";
const myAsyncFunc = () => new Promise((resolve) => resolve(1));
test.fail("fail test", async () => {
  await expect(myAsyncFunc()).rejects.toBe(1);
});
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

test.each

- **类型:** `(cases: ReadonlyArray<T>, ...args: any[]) => void`
- **别名:** `it.each`

当你需要使用不同的变量运行相同的测试时，请使用 `test.each`。你可以按照测试参数的顺序，在测试名称插入符合 `printf` 格式的参数。

- `%s`：字符串
- `%d`：数值
- `%i`：整数
- `%f`：小数
- `%j`：json 格式
- `%o`：对象
- `%#`：对应的测试参数下标
- `%%`：单个百分比符号 ('%)

```
test.each([
  [1, 1, 2],
  [1, 2, 3],
  [2, 1, 3],
])("add(%i, %i) -> %i", (a, b, expected) => {
  expect(a + b).toBe(expected);
});
```

```
// 它将返回如下内容
// ✓ add(1, 1) -> 2
// ✓ add(1, 2) -> 3
// ✓ add(2, 1) -> 3
```

You can also access object properties with `$` prefix, if you are using objects as arguments:

```
test.each([
  { a: 1, b: 1, expected: 2 },
  { a: 1, b: 2, expected: 3 },
  { a: 2, b: 1, expected: 3 },
])('add($a, $b) -> $expected', ({ a, b, expected }) => {
  expect(a + b).toBe(expected)
})

// this will return
// ✓ add(1, 1) -> 2
// ✓ add(1, 2) -> 3
// ✓ add(2, 1) -> 3
```

从 Vitest 0.25.3 开始，你可以使用模板字符串表。

- 第一行应该是列名，使用 `|` 分隔；
- 使用 `${value}` 语法作为模板文本表达式，为一个或多个后续数据行提供数据。

```
test.each`

| a      | b   | expected         |
|--------|-----|------------------|
| 1      | 1   | 2                |
| 'a'    | 'b' | 'ab'             |
| []     | 'b' | 'b'              |
| {}     | 'b' | [Object Object]b |
| asd: 1 | 'b' | [Object Object]b |

`('returns $expected when $a is added $b', ({ a, b, expected }) => {
  expect(a + b).toBe(expected)
})
```

如果你想访问 `TestContext`，请在单个测试中使用 `describe.each`。

当使用 Vitest 作为 [type checker](/guide/testing-types) 时，不能使用此语法。

bench

- 类型:** `(name: string, fn: BenchFunction, options?: BenchOptions) => void`

`bench` 定义了一个基准。在 Vitest 术语中，基准是定义一系列操作的函数。Vitest 多次运行此函数以显示不同的性能结果。

Vitest 在底层使用 `tinybench` 库，继承其所有可用选项作为第三个参数。

```
import { bench } from "vitest";

bench(
  "normal sorting",
  () => {
    const x = [1, 5, 4, 2, 3];
```

```
x.sort((a, b) => {
  return a - b;
});
{
  time: 1000
});
```

```
export interface Options {
  /**
   * time needed for running a benchmark task (milliseconds)
   * @default 500
   */
  time?: number;

  /**
   * number of times that a task should run if even the time option is finished
   * @default 10
   */
  iterations?: number;

  /**
   * function to get the current timestamp in milliseconds
   */
  now?: () => number;

  /**
   * An AbortSignal for aborting the benchmark
   */
  signal?: AbortSignal;

  /**
   * warmup time (milliseconds)
   * @default 100ms
   */
  warmupTime?: number;

  /**
   * warmup iterations
   * @default 5
   */
  warmupIterations?: number;

  /**
   * setup function to run before each benchmark task (cycle)
   */
  setup?: Hook;

  /**
   * teardown function to run after each benchmark task (cycle)
   */
  teardown?: Hook;
}
```

bench.skip

- **类型:** (name: string, fn: BenchFunction, options?: BenchOptions) => void

你可以使用 `bench.skip` 语法跳过运行某些基准测试。

```
import { bench } from "vitest";

bench.skip("normal sorting", () => {
  const x = [1, 5, 4, 2, 3];
  x.sort((a, b) => {
    return a - b;
  });
});
```

bench.only

- **类型:** `(name: string, fn: BenchFunction, options?: BenchOptions) => void`

使用 `bench.only` 仅在指定测试套件中运行某些基准测试。这在调试时很有用。

```
import { bench } from "vitest";

bench.only("normal sorting", () => {
  const x = [1, 5, 4, 2, 3];
  x.sort((a, b) => {
    return a - b;
  });
});
```

bench.todo

- **类型:** `(name: string) => void`

使用 `bench.todo` 来存根基准测试，以便稍后实现。

```
import { bench } from "vitest";

bench.todo("unimplemented test");
```

describe

当你在文件的顶层使用 `test` 或 `bench` 时，它们会被收集为它的隐式套件的一部分。使用 `describe` 你可以在当前上下文中定义一个新套件，作为一组相关的测试或基准以及其他嵌套套件。测试套件可让你组织测试和基准测试，从而使报告更加清晰。

```
// basic.spec.ts
// organizing tests

import { describe, expect, test } from "vitest";

const person = {
  isActive: true,
  age: 32,
};

describe("person", () => {
  test("person is defined", () => {
    expect(person).toBeDefined();
  });

  test("is active", () => {
    expect(person.isActive).toBeTruthy();
  });
});
```

```

});  

  test("age limit", () => {  

    expect(person.age).toBeLessThanOrEqual(32);  

  });  

});

```

如果你需要有测试层次结构，你还可以嵌套描述块：

```

// basic.bench.ts
// organizing benchmarks

import { bench, describe } from "vitest";

describe("sort", () => {
  bench("normal", () => {
    const x = [1, 5, 4, 2, 3];
    x.sort((a, b) => {
      return a - b;
    });
  });

  bench("reverse", () => {
    const x = [1, 5, 4, 2, 3];
    x.reverse().sort((a, b) => {
      return a - b;
    });
  });
});

```

如果你有测试或基准的层次结构，你还可以嵌套描述块：

```

import { describe, expect, test } from "vitest";

const numberToCurrency = (value) => {
  if (typeof value !== "number") throw new Error("Value must be a number");

  return value
    .toFixed(2)
    .toString()
    .replace(/\B(?=(\d{3})+)(?!\d))/g, ",");
};

describe("numberToCurrency", () => {
  describe("given an invalid number", () => {
    test("composed of non-numbers to throw error", () => {
      expect(() => numberToCurrency("abc")).toThrowError();
    });
  });

  describe("given a valid number", () => {
    test("returns the correct currency format", () => {
      expect(numberToCurrency(10000)).toBe("10,000.00");
    });
  });
});

```

describe.skip

- **类型:** `(name: string, fn: TestFunction, options?: number | TestOptions) => void`

在测试套件中使用 `describe.skip` 避免运行特定的描述块。

```
import { assert, describe, test } from "vitest";

describe.skip("skipped suite", () => {
  test("sqrt", () => {
    // 跳过测试套件, 不会有错误
    assert.equal(Math.sqrt(4), 3);
  });
});
```

describe.only

- **类型:** `(name: string, fn: TestFunction, options?: number | TestOptions) => void`

使用 `describe.only` 仅运行指定的测试套件。

```
// 仅运行此测试套件 (以及仅标有的其他测试套件)
describe.only("suite", () => {
  test("sqrt", () => {
    assert.equal(Math.sqrt(4), 3);
  });
});

describe("other suite", () => {
  // ... 测试套件将会被跳过
});
```

有时候只运行一个特定文件中的 `only` 测试用例是很有用的，这可以忽略整个测试套件中的所有其他测试用例，避免污染输出。

为了做到这一点，运行 `vitest` 命令并带上包含有问题的测试的特定文件。

```
# vitest interesting.test.ts
```

describe.concurrent

- **类型:** `(name: string, fn: TestFunction, options?: number | TestOptions) => void`

使用 `describe.concurrent` 在测试套件中将每个测试标记为并发。

```
// 该测试套件中的所有测试都将并行运行
describe.concurrent("suite", () => {
  test("concurrent test 1", async () => {
    /* ... */
  });
  test("concurrent test 2", async () => {
    /* ... */
  });
  test.concurrent("concurrent test 3", async () => {
```

```
    /* ... */
});  
});
```

`.skip`, `.only` 和 `.todo` 可以与并发测试套件一起使用。以下所有组合均有效：

```
describe.concurrent(/* ... */);  
describe.skip.concurrent(/* ... */); // or describe.concurrent.skip(/* ... */)  
describe.only.concurrent(/* ... */); // or describe.concurrent.only(/* ... */)  
describe.todo.concurrent(/* ... */); // or describe.concurrent.todo(/* ... */)
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

describe.shuffle

- **类型:** `(name: string, fn: TestFunction, options?: number | TestOptions) => void`

Vitest 提供了一种通过 CLI 标志 `--sequence.shuffle` 或配置选项 `sequence.shuffle` 来随机运行所有测试的方法，但是如果你只想让测试套件的一部分以随机顺序运行测试，可以使用此标志对其进行标记。

```
describe.shuffle("suite", () => {  
  test("random test 1", async () => {  
    /* ... */  
  });  
  test("random test 2", async () => {  
    /* ... */  
  });  
  test("random test 3", async () => {  
    /* ... */  
  });  
});  
// 执行顺序依赖于 sequence.seed 的配置选项(默认为 Date.now())
```

`.skip`, `.only` 和 `.todo` 可以与并发测试套件一起使用。以下所有组合均有效：

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

describe.todo

- **类型:** `(name: string) => void`

使用 `describe.todo` 将稍后实现的测试套件进行存档。测试报告中将显示一个记录，以便你知道还多少条未实现的测试。

```
// 测试套件的报告中将显示一个记录  
describe.todo("unimplemented suite");
```

describe.each

- **类型:** `(cases: ReadonlyArray<T>, ...args: any[]): (name: string, fn: (...args: T[]) => void, options?: number | TestOptions) => void`

如果你有多个测试依赖相同的数据，可以使用 `describe.each`。

```
describe.each([
  { a: 1, b: 1, expected: 2 },
  { a: 1, b: 2, expected: 3 },
  { a: 2, b: 1, expected: 3 },
])("describe object add($a, $b)", ({ a, b, expected }) => {
  test(`returns ${expected}`, () => {
    expect(a + b).toBe(expected);
  });

  test(`returned value not be greater than ${expected}`, () => {
    expect(a + b).not.toBeGreaterThan(expected);
  });

  test(`returned value not be less than ${expected}`, () => {
    expect(a + b).not.toBeLessThan(expected);
  });
});
```

从 Vitest 0.25.3 开始，你可以使用模板字符串表。

- 第一行应该是列名，使用 `|` 分隔；
- 使用 `${value}` 语法作为模板文本表达式，为一个或多个后续数据行提供数据。

```
describe.each`

| a                                         | b                  | expected                                                                                            |
|-------------------------------------------|--------------------|-----------------------------------------------------------------------------------------------------|
| 1                                         | 1                  | 2                                                                                                   |
| 'a'                                       | 'b'                | 'ab'                                                                                                |
| []                                        | 'b'                | 'b'                                                                                                 |
| {}<br>asd: 1                              | 'b'                | [object Object]b                                                                                    |
| 'describe template string add(\$a, \$b)', | { a, b, expected } | => {       test(`returns \${expected}`, () => {         expect(a + b).toBe(expected)       })     } |

`
```

当使用 Vitest 作为 [type checker](/guide/testing-types) 时，不能使用此语法。

expect

- 类型:** `ExpectStatic & (actual: any) => Assertions`

`expect` 用来创建断言。在当前上下文中，可以使用 `assertions` 方法来断言一个语句。Vitest 默认提供 `chai` 进行断言，同时基于 `chai` 实现兼容 `Jest` 的断言语句。

例如，这里会断言 `input` 的值是否等于 `2`，如果它们不相等，断言则会抛出错误，并且测试失败。

```
import { expect } from "vitest";

const input = Math.sqrt(4);

expect(input).to.equal(2); // chai API
expect(input).toBe(2); // jest API
```

从技术上来说，这里并没有使用 `test` 方法，所以我们在控制台会看到 Nodejs 的报错，而不是 Vitest 的报错。想要了解更多关于 `test` 的信息，请参阅 [test 章节](#)。

此外，`expect` 可用于静态访问匹配器功能，这个后面会介绍。

```
'expect` has no effect on testing types, if expression doesn't have a type error. If you want to use
Vitest as [type checker](/guide/testing-types), use [ `expectTypeOf` ](#expecttypeof) or
[ `assertType` ](#asserttype).
```

not

使用 `not` 将会否定断言。举例，此代码断言 `input` 的值不等于 `2`。如果它们相等，断言则会抛出错误，并且测试失败。

```
import { expect, test } from "vitest";

const input = Math.sqrt(16);

expect(input).not.to.equal(2); // chai API
expect(input).not.toBe(2); // jest API
```

toBe

- **类型:** `(value: any) => Awaitable<void>`

`toBe` 可用于断言基础对象是否相等，或者对象是否共享相同的引用。它相当于调用了 `expect(Object.is(3, 3)).toBe(true)`。如果对象不相同，但你想检查它们的结构是否相同，则可以使用 `toEqual`。

例如，下面的测试将会检查 `stock` 是否有 13 个苹果。

```
import { expect, test } from "vitest";

const stock = {
  type: "apples",
  count: 13,
};

test("stock has 13 apples", () => {
  expect(stock.type).toBe("apples");
  expect(stock.count).toBe(13);
});

test("stocks are the same", () => {
  const refStock = stock; // 相同的引用

  expect(stock).toBe(refStock);
});
```

尽量不要将 `toBe` 与浮点数一起使用。由于 JavaScript 会对它们进行四舍五入，例如 `0.1 + 0.2` 的结果严格来说并不是 `0.3`。如果需要可靠地断言浮点数，请使用 `toBeCloseTo` 进行断言。

toBeCloseTo

- **类型:** `(value: number, numDigits?: number) => Awaitable<void>`

使用 `toBeCloseTo` 进行浮点数的比较。可以选择使用 `numDigits` 参数限制小数点后的检查位数。例如：

```
import { expect, test } from "vitest";

test("decimals are not equal in javascript", () => {
  expect(0.2 + 0.1).toBe(0.3); // 0.2 + 0.1 is 0.30000000000000004
});

test("decimals are rounded to 5 after the point", () => {
  // 0.2 + 0.1 is 0.30000 | "00000000004" removed
  expect(0.2 + 0.1).toBeCloseTo(0.3, 5);
  // nothing from 0.30000000000000004 is removed
  expect(0.2 + 0.1).not.toBeCloseTo(0.3, 50);
});
```

toBeDefined

- **类型:** `() => Awaitable<void>`

`toBeDefined` 断言检查值是否不等于 `undefined`。在检查函数是否有返回值时非常有用。

```
import { expect, test } from "vitest";

const getApples = () => 3;

test("function returned something", () => {
  expect(getApples()).toBeDefined();
});
```

toBeUndefined

- **类型:** `() => Awaitable<void>`

与 `toBeDefined` 相反，`toBeUndefined` 断言检查值是否等于 `undefined`。在检查函数是否没有返回任何内容时非常有用。

```
import { expect, test } from "vitest";

function getApplesFromStock(stock) {
  if (stock === "Bill") return 13;
}

test("mary doesn't have a stock", () => {
  expect(getApplesFromStock("Mary")).toBeUndefined();
});
```

toBeTruthy

- **类型:** `() => Awaitable<void>`

`toBeTruthy` 会将检查值转换为布尔值，断言该值是否为 `true`。该方法在当你不关心检查值的内容，而只想知道它是否可以转换为 `true` 时很有用。

例如下面这段代码，我们就不需要关心 `stocks.getInfo` 的返回值，可能是复杂的对象、字符串或者是其他内容，代码仍然可以运行。

```
import { Stocks } from "./stocks";
const stocks = new Stocks();
stocks.sync("Bill");
if (stocks.getInfo("Bill")) stocks.sell("apples", "Bill");
```

所以如果我们想测试 `stocks.getInfo` 是否为 `true`, 我们可以这样写 :

```
import { expect, test } from "vitest";
import { Stocks } from "./stocks";
const stocks = new Stocks();

test("if we know Bill stock, sell apples to him", () => {
  stocks.sync("Bill");
  expect(stocks.getInfo("Bill")).toBeTruthy();
});
```

JavaScript 中除了 `false`, `0`, `''`, `null`, `undefined` 和 `Nan`, 其他一切都是为真。

toBeFalsy

- **类型:** `() => Awaitable<void>`

`toBeFalsy` 会将检测值转换为布尔值, 断言该值是否为 `false`。该方法在当你不关心该检查值的内容, 但只想知道它是否可以转换为 `false` 时很有用。

例如下面这段代码, 我们就不需要关心 `stocks.stockFailed` 的返回值, 可能是复杂的对象、字符串或者是其他内容, 代码仍然可以运行。

```
import { Stocks } from "./stocks";
const stocks = new Stocks();
stocks.sync("Bill");
if (!stocks.stockFailed("Bill")) stocks.sell("apples", "Bill");
```

所以如果我们想测试 `stocks.stockFailed` 是否为 `false`, 我们可以这样写 :

```
import { expect, test } from "vitest";
import { Stocks } from "./stocks";
const stocks = new Stocks();

test("if Bill stock hasn't failed, sell apples to him", () => {
  stocks.syncStocks("Bill");
  expect(stocks.stockFailed("Bill")).toBeFalsy();
});
```

JavaScript 中除了 `false`, `0`, `''`, `null`, `undefined` 和 `Nan`, 其他一切都是为真。

toBeNull

- **类型:** `() => Awaitable<void>`

`toBeNull` 将简单地断言检查值是否为 `null`。是 `.toBe(null)` 的别名。

```
import { expect, test } from "vitest";

function apples() {
  return null;
```

```

    }

    test("we don't have apples", () => {
      expect(apples()).toBeNull();
    });
}

```

toBeNaN

- **类型:** `(() => Awaitable<void>)`

`toBeNaN` 将简单地断言是否为 `NaN`, 是 `.toBe(NaN)` 的别名。

```

import { expect, test } from "vitest";

let i = 0;

function getApplesCount() {
  i++;
  return i > 1 ? NaN : i;
}

test("getApplesCount has some unusual side effects...", () => {
  expect(getApplesCount()).not.toBeNaN();
  expect(getApplesCount()).toBeNaN();
});

```

toBeTypeOf

- **类型:** `(c: 'bigint' | 'boolean' | 'function' | 'number' | 'object' | 'string' | 'symbol' | 'undefined') => Awaitable<void>`

`toBeTypeOf` 用于断言检查值是否属于接收的类型。

```

import { expect, test } from "vitest";
const actual = "stock";

test("stock is type of string", () => {
  expect(actual).toBeTypeOf("string");
});

```

toBeInstanceOf

- **类型:** `(c: any) => Awaitable<void>`

`toBeInstanceOf` 用于断言检查值是否为接收的类的实例。

```

import { expect, test } from "vitest";
import { Stocks } from "./stocks";
const stocks = new Stocks();

test("stocks are instance of Stocks", () => {
  expect(stocks).toBeInstanceOf(Stocks);
});

```

toBeGreaterThan

- **类型:** `(n: number | bigint) => Awaitable<void>`

`toBeGreaterThan` 用于断言检查值是否大于接收值，如果相等将无法通过测试。

```
import { expect, test } from "vitest";
import { getApples } from "./stock";

test("have more than 10 apples", () => {
  expect(getApples()).toBeGreaterThan(10);
});
```

toBeGreaterThanOrEqual

- **类型:** `(n: number | bigint) => Awaitable<void>`

`toBeGreaterThanOrEqual` 用于断言检查值是否大于等于接收值。

```
import { expect, test } from "vitest";
import { getApples } from "./stock";

test("have 11 apples or more", () => {
  expect(getApples()).toBeGreaterThanOrEqual(11);
});
```

toBeLessThan

- **类型:** `(n: number | bigint) => Awaitable<void>`

`toBeLessThan` 用于断言检查值是否小于接收值，如果相等将无法通过测试。

```
import { expect, test } from "vitest";
import { getApples } from "./stock";

test("have less than 20 apples", () => {
  expect(getApples()).toBeLessThan(20);
});
```

toBeLessThanOrEqual

- **类型:** `(n: number | bigint) => Awaitable<void>`

`toBeLessThanOrEqual` 用于断言检查值是否小于等于接收值。

```
import { expect, test } from "vitest";
import { getApples } from "./stock";

test("have 11 apples or less", () => {
  expect(getApples()).toBeLessThanOrEqual(11);
});
```

toEqual

- **类型:** `(received: any) => Awaitable<void>`

`toEqual` 断言检查值是否等于接收值，或者是同样的结构，如果是对象类型（将会使用递归的方法进行比较）。在本例中，你可以看到 `toEqual` 和 `toBe` 之间的区别：

```
import { expect, test } from "vitest";

const stockBill = {
  type: "apples",
  count: 13,
};

const stockMary = {
  type: "apples",
  count: 13,
};

test("stocks have the same properties", () => {
  expect(stockBill).toEqual(stockMary);
});

test("stocks are not the same", () => {
  expect(stockBill).not.toBe(stockMary);
});
```

警告 该方法不会对`Error`对象执行深度相同比较。如果要测试是否抛出了某个错误，建议使用`[`toThrowError`]`(#tothrowerror)断言。

toStrictEqual

- **类型:** `(received: any) => Awaitable<void>`

`toStrictEqual` 断言检查值是否等于接收值或者同样的结构，如果是对象类型（将会使用递归的方法进行比较），并且会比较它们是否是相同的类型。

与 `.toEqual` 之间的区别：

- 检查属性值为 `undefined` 的键。例如使用 `.toStrictEqual` 时，`{a: undefined, b: 2}` 与 `{b: 2}` 不会匹配。
- 检查数组的稀疏性。例如使用 `.toStrictEqual` 时，`[, 1]` 与 `[undefined, 1]` 不会匹配。
- 检查对象类型是否相等。例如具有字段 `a` 和 `b` 的实例对象不等于具有字段 `a` 和 `b` 的字面量对象。

```
import { expect, test } from "vitest";

class Stock {
  constructor(type) {
    this.type = type;
  }
}

test("structurally the same, but semantically different", () => {
  expect(new Stock("apples")).toEqual({ type: "apples" });
  expect(new Stock("apples")).not.toStrictEqual({ type: "apples" });
});
```

toContain

- **类型:** `(received: string) => Awaitable<void>`

`toContain` 用于断言检查值是否在数组中。还可以检查一个字符串是否为另一个字符串的子串。

```
import { expect, test } from "vitest";
import { getAllFruits } from "./stock";

test("the fruit list contains orange", () => {
  expect(getAllFruits()).toContain("orange");
});
```

toContainEqual

- **类型:** `(received: any) => Awaitable<void>`

`toContainEqual` 用于断言在数组中是否包含具有特定结构和值的元素。它就像对每个元素进行 `toEqual` 操作。

```
import { expect, test } from "vitest";
import { getFruitStock } from "./stock";

test("apple available", () => {
  expect(getFruitStock()).toContainEqual({ fruit: "apple", count: 5 });
});
```

toHaveLength

- **类型:** `(received: number) => Awaitable<void>`

`toHaveLength` 用于断言一个对象是否具有 `.length` 属性，并且它被设置为某个数值。

```
import { expect, test } from "vitest";

test("toHaveLength", () => {
  expect("abc").toHaveLength(3);
  expect([1, 2, 3]).toHaveLength(3);

  expect("").not.toHaveLength(3); // .length 的值并不是3
  expect({ length: 3 }).toHaveLength(3);
});
```

toHaveProperty

- **类型:** `(key: any, received?: any) => Awaitable<void>`

`toHaveProperty` 用于断言对象上是否存在指定 `key` 的属性。

同时该方法还提供了一个可选参数，用于进行深度对比，就像使用 `toEqual` 匹配器来比较接收到的属性值。

```
import { expect, test } from "vitest";

const invoice = {
  isActive: true,
  "P.O": "12345",
  customer: {
```

```

        first_name: "John",
        last_name: "Doe",
        location: "China",
    },
    total_amount: 5000,
    items: [
        {
            type: "apples",
            quantity: 10,
        },
        {
            type: "oranges",
            quantity: 5,
        },
    ],
};

test("John Doe Invoice", () => {
    expect(invoice).toHaveProperty("isActive"); // 断言 key 存在
    expect(invoice).toHaveProperty("total_amount", 5000); // 断言 key 存在且值相等

    expect(invoice).not.toHaveProperty("account"); // 断言 key 不存在

    // 使用 dot 进行深度引用
    expect(invoice).toHaveProperty("customer.first_name");
    expect(invoice).toHaveProperty("customer.last_name", "Doe");
    expect(invoice).not.toHaveProperty("customer.location", "India");

    // 使用包含 key 的数组进行深度引用
    expect(invoice).toHaveProperty("items[0].type", "apples");
    expect(invoice).toHaveProperty("items.0.type", "apples"); // 使用 dot 也可以工作

    // 在数组中包装你的 key 来避免它作为深度引用
    expect(invoice).toHaveProperty(["P.0"], "12345");
});

```

toMatch

- **类型:** `(received: string | regexp) => Awaitable<void>`

`toMatch` 用于断言字符串是否匹配指定的正则表达式或字符串。

```

import { expect, test } from "vitest";

test("top fruits", () => {
    expect("top fruits include apple, orange and grape").toMatch(/apple/);
    expect("applefruits").toMatch("fruit"); // toMatch 也可以是一个字符串
});

```

toMatchObject

- **类型:** `(received: object | array) => Awaitable<void>`

`toMatchObject` 用于断言对象是否匹配指定的对象属性的子集。

我们还可以传递对象数组。如果我们只想检查两个数组的元素数量是否匹配，该方法就会很有用，它不同于 `arrayContaining`，它允许接收数组中的额外元素。

```

import { expect, test } from "vitest";

const johnInvoice = {
  isActive: true,
  customer: {
    first_name: "John",
    last_name: "Doe",
    location: "China",
  },
  total_amount: 5000,
  items: [
    {
      type: "apples",
      quantity: 10,
    },
    {
      type: "oranges",
      quantity: 5,
    },
  ],
};

const johnDetails = {
  customer: {
    first_name: "John",
    last_name: "Doe",
    location: "China",
  },
};

test("invoice has john personal details", () => {
  expect(johnInvoice).toMatchObject(johnDetails);
});

test("the number of elements must match exactly", () => {
  // 断言对象数组是否匹配
  expect([{} foo: "bar", {} baz: 1 ]).toMatchObject([
    { foo: "bar" },
    { baz: 1 },
  ]);
});

```

toThrowError

- **类型:** `(received: any) => Awaitable<void>`

`toThrowError` 用于断言函数在调用时是否抛出错误。

例如，如果我们想测试 `getFruitStock('pineapples')` 是否会抛出异常，我们可以这样写：

你可以提供一个可选参数来测试是否引发了指定的错误：

- 正则表达式：错误信息通过正则表达式匹配
- 字符串：错误消息包含指定子串

提示 你必须将代码包装在一个函数中，否则将无法捕获错误并且断言将会失败。

```

import { expect, test } from "vitest";

function getFruitStock(type) {
  if (type === "pineapples") {
    throw new DiabetesError(
      "Pineapples is not good for people with diabetes"
    );
  }

  // 可以做一些其他的事情
}

test("throws on pineapples", () => {
  // 测试错误消息是否在某处显示 "diabetes" : 这些是等效的
  expect(() => getFruitStock("pineapples")).toThrowError(/diabetes/);
  expect(() => getFruitStock("pineapples")).toThrowError("diabetes");

  // 测试确切的错误信息
  expect(() => getFruitStock("pineapples")).toThrowError(
    /^Pineapples is not good for people with diabetes$/
  );
});

```

toMatchSnapshot

- **类型:** <T>(shape?: Partial<T> | string, message?: string) => void

这可确保某个值匹配最近的快照。

你可以提供附加到测试名称的可选 `hint` 字符串参数。尽管 Vitest 总是在快照名称的末尾附加一个数字，但在区分单个 it 或测试块中的多个快照时，简短的描述性提示可能比数字更有用。Vitest 在相应的 `.snap` 文件中按名称对快照进行排序。

提示 当快照不匹配导致测试失败时，如果不匹配是预期的，你可以按 `u` 键更新一次快照。或者可以通过 `--update` CLI 选项使 Vitest 始终更新测试。

```

import { expect, test } from "vitest";

test("matches snapshot", () => {
  const data = { foo: new Set(["bar", "snapshot"]) };
  expect(data).toMatchSnapshot();
});

```

如果你只是测试一个对象的结构，并且不需要它是 100% 兼容的，还可以提供一个对象的结构：

```

import { expect, test } from "vitest";

test("matches snapshot", () => {
  const data = { foo: new Set(["bar", "snapshot"]) };
  expect(data).toMatchSnapshot({ foo: expect.any(Set) });
});

```

toMatchInlineSnapshot

- **类型:** <T>(shape?: Partial<T> | string, snapshot?: string, message?: string) => void

这可确保某个值匹配最近的快照。

Vitest 将 inlineSnapshot 字符串参数添加并更新到测试文件中的匹配器（而不是外部 `.snap` 文件）。

```
import { expect, test } from "vitest";

test("matches inline snapshot", () => {
  const data = { foo: new Set(["bar", "snapshot"]) };
  // 当更新快照时, Vitest 将更新以下内容
  expect(data).toMatchInlineSnapshot(`{
    "foo": Set {
      "bar",
      "snapshot",
    },
  `);
});
```

如果你只是测试一个对象的结构，并且不需要它是 100% 兼容的，还可以提供一个对象的结构：

```
import { expect, test } from "vitest";

test("matches snapshot", () => {
  const data = { foo: new Set(["bar", "snapshot"]) };
  expect(data).toMatchInlineSnapshot(
    { foo: expect.any(Set) },
    {
      "foo": Any<Set>,
    }
  );
});
```

toThrowErrorMatchingSnapshot

- **类型:** `(message?: string) => void`

与 `toMatchSnapshot` 相同，但需要与 `toThrowError` 相同的值。

如果函数抛出 `Error`，则快照将是错误消息。否则，快照将是函数抛出的值。

toThrowErrorMatchingInlineSnapshot

- **类型:** `(snapshot?: string, message?: string) => void`

与 `toMatchInlineSnapshot` 相同，但需要与 `toThrowError` 相同的值。

如果函数抛出 `Error`，则快照将是错误消息。否则，快照将是函数抛出的值。

toHaveBeenCalled

- **类型:** `() => Awaitable<void>`

此断言可以测试一个函数是否被调用过。需要给 `expect` 传递一个监听函数。

```

import { expect, test, vi } from "vitest";

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
};

test("spy function", () => {
  const buySpy = vi.spyOn(market, "buy");

  expect(buySpy).not.toHaveBeenCalled();

  market.buy("apples", 10);

  expect(buySpy).toHaveBeenCalled();
});

```

toHaveBeenCalledTimes

- **类型:** `(amount: number) => Awaitable<void>`

此断言将会检查一个函数是否被调用了一定的次数。需要给 `expect` 传递一个监听函数。

```

import { expect, test, vi } from "vitest";

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
};

test("spy function called two times", () => {
  const buySpy = vi.spyOn(market, "buy");

  market.buy("apples", 10);
  market.buy("apples", 20);

  expect(buySpy).toHaveBeenCalledTimes(2);
});

```

toHaveBeenCalledWith

- **类型:** `(...args: any[]) => Awaitable<void>`

此断言将会检查一个函数是否被调用过，并且传入了指定的参数。需要给 `expect` 传递一个监听函数。

```

import { expect, test, vi } from "vitest";

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
};

test("spy function", () => {
  const buySpy = vi.spyOn(market, "buy");

  market.buy("apples", 10);
}

```

```
market.buy("apples", 20);

expect(buySpy).toHaveBeenCalledWith("apples", 10);
expect(buySpy).toHaveBeenCalledWith("apples", 20);
});
```

toHaveBeenCalledWith

- **类型:** (...args: any[]) => Awaitable<void>

此断言将会检查一个函数在最后一次被调用时，是否使用了某些参数。需要给 `expect` 传递一个监听函数。

```
import { expect, test, vi } from "vitest";

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
};

test("spy function", () => {
  const buySpy = vi.spyOn(market, "buy");

  market.buy("apples", 10);
  market.buy("apples", 20);

  expect(buySpy).not.toHaveBeenCalledWith("apples", 10);
  expect(buySpy).toHaveBeenCalledWith("apples", 20);
});
```

toHaveBeenCalledNthWith

- **类型:** (time: number, ...args: any[]) => Awaitable<void>

此断言将会检查一个函数在第某次调用时是否使用了某些参数，从第 1 次开始。所以如果要检查第 2 次调用，你可以这样写 `.toHaveBeenCalledNthWith(2, ...)`。

需要给 `expect` 传递一个监听函数。

```
import { expect, test, vi } from "vitest";

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
};

test("first call of spy function called with right params", () => {
  const buySpy = vi.spyOn(market, "buy");

  market.buy("apples", 10);
  market.buy("apples", 20);

  expect(buySpy).toHaveBeenCalledWith(1, "apples", 10);
});
```

toHaveReturned

- **类型:** `(() => Awaitable<void>)`

此断言检查一个函数是否至少成功返回了一次值（即没有抛出错误）。需要给 `expect` 传递一个监听函数。

```
import { expect, test, vi } from "vitest";

const getApplesPrice = (amount: number) => {
  const PRICE = 10;
  return amount * PRICE;
};

test("spy function returned a value", () => {
  const getPriceSpy = vi.fn(getApplesPrice);

  const price = getPriceSpy(10);

  expect(price).toBe(100);
  expect(getPriceSpy).toHaveReturned();
});
```

toHaveReturnedTimes

- **类型:** `(amount: number) => Awaitable<void>`

此断言将会检查一个函数是否成功返回了确切的次数（即没有抛出错误）。需要给 `expect` 传递一个监听函数。

```
import { expect, test, vi } from "vitest";

test("spy function returns a value two times", () => {
  const sell = vi.fn((product: string) => ({ product }));

  sell("apples");
  sell("bananas");

  expect(sell).toHaveReturnedTimes(2);
});
```

toHaveReturnedWith

- **类型:** `(returnValue: any) => Awaitable<void>`

此断言将会检查一个函数是否至少一次成功返回了指定的值（即没有抛出错误）。需要给 `expect` 传递一个监听函数。

```
import { expect, test, vi } from "vitest";

test("spy function returns a product", () => {
  const sell = vi.fn((product: string) => ({ product }));

  sell("apples");

  expect(sell).toHaveReturnedWith({ product: "apples" });
});
```

toHaveLastReturnedWith

- **类型:** `(returnValue: any) => Awaitable<void>`

此断言将会检查一个函数是否在最后一次被调用时返回了指定的值。需要给 `expect` 传递一个监听函数。

```
import { expect, test, vi } from "vitest";

test("spy function returns bananas on a last call", () => {
  const sell = vi.fn((product: string) => ({ product }));
  sell("apples");
  sell("bananas");
  expect(sell).toHaveLastReturnedWith({ product: "bananas" });
});
```

toHaveNthReturnedWith

- **类型:** `(time: number, returnValue: any) => Awaitable<void>`

此断言将会检查一个函数是否第某次被调用时返回了指定的值。需要给 `expect` 传递一个监听函数。

```
import { expect, test, vi } from "vitest";

test("spy function returns bananas on second call", () => {
  const sell = vi.fn((product: string) => ({ product }));
  sell("apples");
  sell("bananas");
  expect(sell).toHaveNthReturnedWith(2, { product: "bananas" });
});
```

toSatisfy

- **类型:** `(predicate: (value: any) => boolean) => Awaitable<void>`

此断言检查一个值是否满足某个谓词。

```
describe("toSatisfy()", () => {
  const isOdd = (value: number) => value % 2 !== 0;

  it("pass with 0", () => {
    expect(1).toSatisfy(isOdd);
  });

  it("pass with negotiation", () => {
    expect(2).not.toSatisfy(isOdd);
  });
});
```

resolves

- **类型:** `Promisify<Assertions>`

`resolves` 可以在断言异步代码时有意地删除样板语法。使用它可以从待处理的 `Promise` 中去展开它的值，并使用通常的断言语句来断言它的值。如果 `Promise` 被拒绝，则断言将会失败。

它返回相同的 `Assertions` 对象，但所有匹配器现在都是返回 `Promise`，因此你需要使用 `await` 去阻塞它。同样也适用于 `chai` 断言。

例如，如果我们有一个函数，它调用 API 并返回一些数据，你可以使用下列代码来断言它的返回值：

```
import { expect, test } from "vitest";

async function buyApples() {
    return fetch("/buy/apples").then((r) => r.json());
}

test("buyApples returns new stock id", async () => {
    // toEqual 现在返回一个 Promise，所以我们必须等待它
    await expect(buyApples()).resolves.toEqual({ id: 1 }); // jest API
    await expect(buyApples()).resolves.to.equal({ id: 1 }); // chai API
});
```

警告 如果没有等待断言，那么我们将有一个每次都会通过的误报测试。为了确保断言确实发生，我们可以使用 `[`expect.assertions(number)`](#expect-assertions)`。

rejects

- **类型:** `Promisify<Assertions>`

`rejects` 可以在断言异步代码时有意地删除样板语法。使用它来展开 `Promise` 被拒绝的原因，并使用通常的断言语句来断言它的值。如果 `Promise` 成功解决，则断言将失败。

它返回相同的 `Assertions` 对象，但所有匹配器现在都返回 `Promise`，因此你需要使用 `await` 去阻塞它。同样也适用于 `chai` 断言。

例如，如果我们有一个调用失败的函数，我们可以使用此代码来断言原因：

```
import { expect, test } from "vitest";

async function buyApples(id) {
    if (!id) throw new Error("no id");
}

test("buyApples throws an error when no id provided", async () => {
    // toThrow 现在返回一个 Promise，所以你必须等待它
    await expect(buyApples()).rejects.toThrow("no id");
});
```

警告 如果不等待断言，那么我们将有一个每次都会通过的误报测试。为确保断言确实发生，我们可以使用 `[`expect.assertions(number)`](#expect-assertions)`。

expect.assertions

- **类型:** `(count: number) => void`

在测试通过或失败后，它将会验证在测试期间调用了多少次断言。它常用于检查异步代码是否被调用了。

例如，如果我们有一个异步调用两个匹配器的函数，我们可以断言它们实际上是被调用的。

```
import { expect, test } from "vitest";

async function doAsync(...cbs) {
    await Promise.all(cbs.map((cb, index) => cb({ index })));
}

test("all assertions are called", async () => {
    expect.assertions(2);
    function callback1(data) {
        expect(data).toBeTruthy();
    }
    function callback2(data) {
        expect(data).toBeTruthy();
    }

    await doAsync(callback1, callback2);
});
```

expect.hasAssertions

- **类型:** `(() => void)`

在测试通过或失败后，它将会验证在测试期间是否至少调用了一个断言。它常用于检查是否调用了异步代码。

例如，如果我们有一个调用回调的代码，我们可以在回调中进行断言，但如果我不检查是否调用了断言，测试将始终通过。

```
import { expect, test } from "vitest";
import { db } from "./db";

const cbs = [];

function onSelect(cb) {
    cbs.push(cb);
}

// 从 db 中选择后，我们调用所有的回调
function select(id) {
    return db.select({ id }).then((data) => {
        return Promise.all(cbs.map((cb) => cb(data)));
    });
}

test("callback was called", async () => {
    expect.hasAssertions();
    onSelect((data) => {
        // 在选择时调用
        expect(data).toBeTruthy();
    });
    // 如果不等待，测试将失败
    // 如果你没有 expect.hasAssertions()，测试将通过
    await select(3);
});
```

expect.anything

- **类型:** `(() => any)`

这种非对称匹配器在与相等检查一起使用时，将始终返回 `true`。如果你只是想确保该属性存在时很有用。

```
import { expect, test } from "vitest";

test('object has "apples" key', () => {
  expect({ apples: 22 }).toEqual({ apples: expect.anything() });
});
```

expect.any

- **类型:** `(constructor: unknown) => any`

此非对称匹配器与相等检查一起使用时，仅当 `value` 是指定构造函数的实例时才会返回 `true`。如果你有一个每次都生成的值，并且只想知道它以正确的类型存在是很有用。

```
import { expect, test } from "vitest";
import { generateId } from "./generators";

test('"id" is a number', () => {
  expect({ id: generateId() }).toEqual({ id: expect.any(Number) });
});
```

expect.arrayContaining

- **类型:** `<T>(expected: T[]) => any`

当与相等检查一起使用时，如果 `value` 是一个数组并包含指定的选项，则此非对称匹配器将返回 `true`。

```
import { expect, test } from "vitest";

test("basket includes fuji", () => {
  const basket = {
    varieties: ["Empire", "Fuji", "Gala"],
    count: 3,
  };
  expect(basket).toEqual({
    count: 3,
    varieties: expect.arrayContaining(["Fuji"]),
  });
});
```

提示 你可以将 `expect.not` 与此匹配器一起使用来否定预期值。

expect.objectContaining

- **类型:** `(expected: any) => any`

当与相等检查一起使用时，如果 `value` 具有相似的结构，则此非对称匹配器将返回 `true`。

```
import { expect, test } from "vitest";

test("basket has empire apples", () => {
  const basket = {
    varieties: [
      {
        name: "Empire",
        count: 1,
      },
    ],
  };
  expect(basket).toEqual({
    varieties: [expect.objectContaining({ name: "Empire" })],
  });
});
```

提示 你可以将 `expect.not` 与此匹配器一起使用来否定预期值。

expect.stringContaining

- **类型:** `(expected: any) => any`

当与相等检查一起使用时，如果 value 是字符串并且包含指定的子字符串，则此非对称匹配器将返回 `true`。

```
import { expect, test } from "vitest";

test('variety has "Emp" in its name', () => {
  const variety = {
    name: "Empire",
    count: 1,
  };
  expect(basket).toEqual({
    name: expect.stringContaining("Emp"),
    count: 1,
  });
});
```

提示 你可以将 `expect.not` 与此匹配器一起使用来否定预期值。

expect.stringMatching

- **类型:** `(expected: any) => any`

当与相等检查一起使用时，如果 value 是字符串并且包含指定的子字符串或字符串匹配正则表达式，则此非对称匹配器将返回 `true`。

```
import { expect, test } from "vitest";

test('variety ends with "re"', () => {
  const variety = {
    name: "Empire",
    count: 1,
  };
  expect(basket).toEqual({
    name: expect.stringMatching(/re$/),
  });
});
```

```
    count: 1,
  });
});
```

提示 你可以将 `expect.not` 与此匹配器一起使用来否定预期值。

expect.addSnapshotSerializer

- **类型:** `(plugin: PrettyFormatPlugin) => void`

此方法在创建快照时添加调用的自定义序列化程序。这是高级功能 - 如果你想了解更多信息, 请阅读 [自定义序列化程序指南](#)。

如果你要添加自定义序列化程序, 则应在 `setupFiles` 中调用此方法。它将影响每个快照。

提示 如果你以前使用 Vue CLI 和 Jest, 可能需要安装 `[jest-serializer-vue]` (<https://www.npmjs.com/package/jest-serializer-vue>)。否则, 你的快照将被包裹在一个字符串中, 这会导致 `"` 被转义。

expect.extend

- **类型:** `(matchers: MatchersObject) => void`

你可以使用自己的扩展默认匹配器。此函数用于使用自定义匹配器扩展匹配器对象。

当你以这种方式定义匹配器时, 还创建了可以像 `expect.stringContaining` 一样使用的不对称匹配器。

```
import { expect, test } from "vitest";

test("custom matchers", () => {
  expect.extend({
    toBeFoo: (received, expected) => {
      if (received !== "foo") {
        return {
          message: () => `expected ${received} to be foo`,
          pass: false,
        };
      }
    },
  });
  expect("foo").toBeFoo();
  expect({ foo: "foo" }).toEqual({ foo: expect.toBeFoo() });
});
```

如果你希望匹配器出现在每个测试中, 应该在 `setupFiles` 中调用此方法。

这个函数与 Jest 的 `expect.extend` 兼容, 因此任何使用它来创建自定义匹配器的库都可以与 Vitest 一起使用。

如果你使用的是 TypeScript, 可以使用以下代码扩展默认的 Matchers 接口 :

```

interface CustomMatchers<R = unknown> {
  toBeFoo(): R;
}

declare global {
  namespace Vi {
    interface Assertion extends CustomMatchers {}
    interface AsymmetricMatchersContaining extends CustomMatchers {}
  }
}

```

注意：增加 `jest.Matchers` 接口也可以工作。

提示 如果你想了解更多信息，请查看 [关于扩展匹配器的指南](/guide/extending-matchers)。

expectTypeOf

- **Type:** `<T>(a: unknown) => ExpectTypeof`

not

- **Type:** `ExpectTypeof`

You can negate all assertions, using `.not` property.

toEqualTypeOf

- **Type:** `<T>(expected: T) => void`

This matcher will check, if types are fully equal to each other. This matcher will not fail, if two objects have different values, but the same type, but will fail, if object is missing a property.

```

import { expectTypeOf } from "vitest";

expectTypeOf({ a: 1 }).toEqualTypeOf<{ a: number }>();
expectTypeOf({ a: 1 }).toEqualTypeOf({ a: 1 });
expectTypeOf({ a: 1 }).toEqualTypeOf({ a: 2 });
expectTypeOf({ a: 1, b: 1 }).not.toEqualTypeOf<{ a: number }>();

```

toMatchTypeOf

- **Type:** `<T>(expected: T) => void`

This matcher checks if expect type extends provided type. It is different from `toEqual` and is more similar to expect's `toMatch`. With this matcher you can check, if an object "matches" a type.

```

import { expectTypeOf } from "vitest";

expectTypeOf({ a: 1, b: 1 }).toMatchTypeOf({ a: 1 });
expectTypeOf<number>().toMatchTypeOf<string | number>();
expectTypeOf<string | number>().not.toMatchTypeOf<number>();

```

extract

- **Type:** `ExpectTypeOf<ExtractedUnion>`

You can use `.extract` to narrow down types for further testing.

```
import { expectTypeOf } from "vitest";

type ResponsiveProp<T> = T | T[] | { xs?: T; sm?: T; md?: T };
const getResponsiveProp = <T>(_props: T): ResponsiveProp<T> => ({});
interface CSSProperties {
  margin?: string;
  padding?: string;
}

const cssProperties: CSSProperties = { margin: "1px", padding: "2px" };

expectTypeOf(getResponsiveProp(cssProperties))
  .extract<{ xs?: any }>() // extracts the last type from a union
  .toEqualTypeOf<{
    xs?: CSSProperties;
    sm?: CSSProperties;
    md?: CSSProperties;
  }>();

expectTypeOf(getResponsiveProp(cssProperties))
  .extract<unknown[]>() // extracts an array from a union
  .toEqualTypeOf<CSSProperties[]>();
```

If no type is found in the union, `.extract` will return `'never'`.

exclude

- **Type:** `ExpectTypeOf<NonExcludedUnion>`

You can use `.exclude` to remove types from a union for further testing.

```
import { expectTypeOf } from "vitest";

type ResponsiveProp<T> = T | T[] | { xs?: T; sm?: T; md?: T };
const getResponsiveProp = <T>(_props: T): ResponsiveProp<T> => ({});
interface CSSProperties {
  margin?: string;
  padding?: string;
}

const cssProperties: CSSProperties = { margin: "1px", padding: "2px" };

expectTypeOf(getResponsiveProp(cssProperties))
  .exclude<unknown[]>()
  .exclude<{ xs?: unknown }>() // or just .exclude<unknown[] | { xs?: unknown }>()
  .toEqualTypeOf<CSSProperties>();
```

If no type is found in the union, `.exclude` will return `'never'`.

returns

- **Type:** `ExpectTypeOf<ReturnValue>`

You can use `.returns` to extract return value of a function type.

```
import { expectTypeOf } from "vitest";

expectTypeOf(() => {}).returns.toBeVoid();
expectTypeOf((a: number) => [a, a]).returns.toEqualTypeOf([1, 2]);
```

If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

parameters

- **Type:** `ExpectTypeOf<Parameters>`

You can extract function arguments with `.parameters` to perform assertions on its value. Parameters are returned as an array.

```
import { expectTypeOf } from "vitest";

type NoParam = () => void;
type HasParam = (s: string) => void;

expectTypeOf<NoParam>().parameters.toEqualTypeOf<[]>();
expectTypeOf<HasParam>().parameters.toEqualTypeOf<[string]>();
```

If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

You can also use `[`.toBeCallableWith`](#tobecallablewith)` matcher as a more expressive assertion.

parameter

- **Type:** `(nth: number) => ExpectTypeOf`

You can extract a certain function argument with `.parameter(number)` call to perform other assertions on it.

```
import { expectTypeOf } from "vitest";

const foo = (a: number, b: string) => [a, b];

expectTypeOf(foo).parameter(0).toBeNumber();
expectTypeOf(foo).parameter(1).toBeString();
```

If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

constructorParameters

- **Type:** `ExpectTypeOf<ConstructorParameters>`

You can extract constructor parameters as an array of values and perform assertions on them with this method.

```
import { expectTypeOf } from "vitest";

expectTypeOf(Date).constructorParameters.toEqualTypeOf<
  [ ] | [string | number | Date]
>();
```

If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

You can also use `[`.toBeConstructibleWith`](#tobeconstructiblewith)` matcher as a more expressive assertion.

instance

- **Type:** `ExpectTypeOf<ConstructableInstance>`

This property gives access to matchers that can be performed on an instance of the provided class.

```
import { expectTypeOf } from "vitest";

expectTypeOf(Date).instance.toHaveProperty("toISOString");
```

If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

items

- **Type:** `ExpectTypeOf<T>`

You can get array item type with `.items` to perform further assertions.

```
import { expectTypeOf } from "vitest";

expectTypeOf([1, 2, 3]).items.toEqualTypeOf<number>();
expectTypeOf([1, 2, 3]).items.not.toEqualTypeOf<string>();
```

resolves

- **Type:** `ExpectTypeOf<ResolvedPromise>`

This matcher extracts resolved value of a `Promise`, so you can perform other assertions on it.

```
import { expectTypeOf } from "vitest";

const asyncFunc = async () => 123;

expectTypeOf(asyncFunc).returns.resolves.toBeNumber();
expectTypeOf(Promise.resolve("string")).resolves.toBeString();
```

If used on a non-promise type, it will return `never`, so you won't be able to chain it with other matchers.

guards

- **Type:** `ExpectTypeOf<Guard>`

This matcher extracts guard value (e.g., `v is number`), so you can perform assertions on it.

```
import { expectTypeOf } from "vitest";

const isString = (v: any): v is string => typeof v === "string";
expectTypeOf(isString).guards.toBeString();
```

Returns `never`, if the value is not a guard function, so you won't be able to chain it with other matchers.

asserts

- **Type:** `ExpectTypeOf<Assert>`

This matcher extracts assert value (e.g., `assert v is number`), so you can perform assertions on it.

```
import { expectTypeOf } from "vitest";

const assertNumber = (v: any): asserts v is number => {
  if (typeof v !== "number") throw new TypeError("Nope !");
};

expectTypeOf(assertNumber).asserts.toBeNumber();
```

Returns `never`, if the value is not an assert function, so you won't be able to chain it with other matchers.

toBeAny

- **Type:** `() => void`

With this matcher you can check, if provided type is `any` type. If the type is too specific, the test will fail.

```
import { expectTypeOf } from "vitest";

expectTypeOf<any>().toBeAny();
expectTypeOf({} as any).toBeAny();
expectTypeOf("string").not.toBeAny();
```

toBeUnknown

- **Type:** `() => void`

This matcher checks, if provided type is `unknown` type.

```
import { expectTypeOf } from "vitest";

expectTypeOf().toBeUnknown();
expectTypeOf({} as unknown).toBeUnknown();
expectTypeOf("string").not.toBeUnknown();
```

toBeNever

- **Type:** `() => void`

This matcher checks, if provided type is a `never` type.

```
import { expectTypeOf } from "vitest";

expectTypeOf<never>().toBeNever();
expectTypeOf((): never => {}).returns.toBeNever();
```

toBeFunction

- **Type:** `() => void`

This matcher checks, if provided type is a `function`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(42).not.toBeFunction();
expectTypeOf((): never => {}).toBeFunction();
```

toBeObject

- **Type:** `() => void`

This matcher checks, if provided type is an `object`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(42).not.toBeObject();
expectTypeOf({}).toBeObject();
```

toBeArray

- **Type:** `() => void`

This matcher checks, if provided type is `Array<T>`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(42).not.toBeArray();
```

```
expectTypeOf([]).toBeArray();
expectTypeOf([1, 2]).toBeArray();
expectTypeOf([{}, 42]).toBeArray();
```

toEqual

- **Type:** `() => void`

This matcher checks, if provided type is a `string`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(42).not.toBeString()
expectTypeOf('').toBeString()
expectTypeOf('a').toBeString()
```

toEqual

- **Type:** `() => void`

This matcher checks, if provided type is `boolean`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(42).not.toBeBoolean();
expectTypeOf(true).toBeBoolean();
expectTypeOf<boolean>().toBeBoolean();
```

toEqual

- **Type:** `() => void`

This matcher checks, if provided type is `void`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(() => {}).returns.toBeVoid();
expectTypeOf<void>().toBeVoid();
```

toEqual

- **Type:** `() => void`

This matcher checks, if provided type is a `symbol`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(Symbol(1)).toBeSymbol();
expectTypeOf<symbol>().toBeSymbol();
```

toEqual

- **Type:** `() => void`

This matcher checks, if provided type is `null`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(null).toBeNull();
expectTypeOf<null>().toBeNull();
expectTypeOf(undefined).not.toBeNull();
```

toBeUndefined

- **Type:** `() => void`

This matcher checks, if provided type is `undefined`.

```
import { expectTypeOf } from "vitest";

expectTypeOf(undefined).toBeUndefined();
expectTypeOf<undefined>().toBeUndefined();
expectTypeOf(null).not.toBeUndefined();
```

toBeNullable

- **Type:** `() => void`

This matcher checks, if you can use `null` or `undefined` with provided type.

```
import { expectTypeOf } from "vitest";

expectTypeOf<1 | undefined>().toBeNullable();
expectTypeOf<1 | null>().toBeNullable();
expectTypeOf<1 | undefined | null>().toBeNullable();
```

toBeCallableWith

- **Type:** `() => void`

This matcher ensures you can call provided function with a set of parameters.

```
import { expectTypeOf } from "vitest";

type NoParam = () => void;
type HasParam = (s: string) => void;

expectTypeOf<NoParam>().toBeCallableWith();
expectTypeOf<HasParam>().toBeCallableWith("some string");
```

If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

toBeConstructibleWith

- **Type:** `() => void`

This matcher ensures you can create a new instance with a set of constructor parameters.

```
import { expectTypeOf } from "vitest";

expectTypeOf(Date).toBeConstructibleWith(new Date());
expectTypeOf(Date).toBeConstructibleWith("01-01-2000");
```

If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

toHaveProperty

- Type:** <K extends keyof T>(property: K) => ExpectTypeOf<T[K]>

This matcher checks if a property exists on provided object. If it exists, it also returns the same set of matchers for the type of this property, so you can chain assertions one after another.

```
import { expectTypeOf } from "vitest";

const obj = { a: 1, b: "" };

expectTypeOf(obj).toHaveProperty("a");
expectTypeOf(obj).not.toHaveProperty("c");

expectTypeOf(obj).toHaveProperty("a").toBeNumber();
expectTypeOf(obj).toHaveProperty("b").toBeString();
expectTypeOf(obj).toHaveProperty("a").not.toBeString();
```

assertType

- Type:** <T>(value: T): void

You can use this function as an alternative for `expectTypeOf` to easily assert that argument type is equal to provided generic.

```
import { assertType } from "vitest";

function concat(a: string, b: string): string;
function concat(a: number, b: number): number;
function concat(a: string | number, b: string | number): string | number;

assertType<string>(concat("a", "b"));
assertType<number>(concat(1, 2));
// @ts-expect-error wrong types
assertType(concat("a", 2));
```

Setup and Teardown

这些功能允许我们连接到测试的生命周期，以避免重复设置和拆卸代码。它们适用于当前上下文：如果它们在顶层使用，则适用于文件；如果它们在 `describe` 块内，则适用于当前测试套件。

beforeEach

- 类型:** beforeEach(fn: () => Awaitable<void>, timeout?: number)

注册一个回调，在当前上下文中的每个测试运行之前被调用。如果函数返回一个 `Promise`，`Vitest` 会等到 `Promise` 解决后再运行测试。

或者，我们可以传递一个超时（以毫秒为单位），定义在终止之前等待多长时间。默认值为 5 秒。

```
import { beforeEach } from "vitest";

beforeEach(async () => {
  // 在每次测试运行之前清除模拟并添加一些测试数据
  await stopMocking();
  await addUser({ name: "John" });
});
```

这里的 `beforeEach` 确保为每个测试都添加用户。

从 Vitest v0.10.0 开始，`beforeEach` 还接受一个可选的清理功能（相当于 `afterEach`）。

```
import { beforeEach } from "vitest";

beforeEach(async () => {
  // 在所有测试运行之前调用一次
  await prepareSomething();

  // 清理方法，在所有测试运行后调用一次
  return async () => {
    await resetSomething();
  };
});
```

forEach

- **类型:** `forEach(fn: () => Awaitable<void>, timeout?: number)`

注册一个回调，在当前上下文中的每个测试运行之后被调用。如果函数返回一个 `Promise`，`Vitest` 会等到 `Promise` 解决后再继续。

或者，你可以设置超时（以毫秒为单位）以指定在终止前等待多长时间。默认值为 5 秒。

```
import { afterEach } from "vitest";

afterEach(async () => {
  await clearTestingData(); // 每次测试运行后清除测试数据
});
```

这里的 `afterEach` 确保在每次测试运行后清除测试数据。

beforeAll

- **类型:** `beforeAll(fn: () => Awaitable<void>, timeout?: number)`

注册一个回调，在开始运行当前上下文中的所有测试之前被调用一次。如果函数返回一个 `Promise`，`Vitest` 会等到 `Promise` 解决后再运行测试。

或者，你可以提供超时（以毫秒为单位）以指定在终止之前等待多长时间。默认值为 5 秒。

```
import { beforeAll } from "vitest";

beforeAll(async () => {
  await startMocking(); // 在所有测试运行之前调用一次
});
```

这里的 `beforeAll` 确保在测试运行之前设置模拟数据。

从 Vitest v0.10.0 开始, `beforeAll` 还可以接受一个可选的清理功能 (相当于 `afterAll`)。

```
import { beforeAll } from "vitest";

beforeAll(async () => {
  // 在所有测试运行之前调用一次
  await startMocking();

  // 清理函数, 在所有测试运行后调用一次
  return async () => {
    await stopMocking();
  };
});
```

afterAll

- **类型:** `afterAll(fn: () => Awaitable<void>, timeout?: number)`

注册一个回调, 在当前上下文中运行所有测试后被调用一次。如果函数返回一个 Promise, Vitest 会等到 Promise 解决后再继续。

或者, 你可以提供超时 (以毫秒为单位) 以指定在终止之前等待多长时间。默认值为 5 秒。

```
import { afterAll } from "vitest";

afterAll(async () => {
  await stopMocking(); // 在所有测试运行后调用此方法
});
```

这里的 `afterAll` 确保在所有测试运行后调用 `stopMocking` 方法。

Vi

Vitest 通过 `vi` 提供工具函数来帮助你。你可以 `import { vi } from 'vitest'` 或 **全局地** 访问它 (当 `globals configuration` 启用时)。

vi.advanceTimersByTime

- **类型:** `(ms: number) => Vitest`

就像 `runAllTimers` 一样工作, 但会在经过几毫秒后结束。例如, 这将输出 `1, 2, 3` 并且不会抛出 :

```
let i = 0;
setInterval(() => console.log(++i), 50);

vi.advanceTimersByTime(150);
```

vi.advanceTimersToNextTimer

- **类型:** `(() => Vitest)`

调用下一个可调用的计时器。这在每个计时器调用间隔内进行断言很有用。你可以链式调用它来自管理计时器。

```
let i = 0;
setInterval(() => console.log(++i), 50);

vi.advanceTimersToNextTimer() // log 1
    .advanceTimersToNextTimer() // log 2
    .advanceTimersToNextTimer(); // log 3
```

vi.clearAllMocks

将在所有对象监听(spies)上调用 `.mockClear()`。这将清除模拟对象(mock)历史，但不会将其实现重置为默认实现。

vi.clearAllTimers

删除所有计划运行的计时器。这些计时器后续将不会运行。

vi.dynamicImportSettled

等待加载所有导入。如果你有一个开始导入模块的同步调用时很有用，否则你不能等待。

vi.fn

- **类型:** `(fn?: Function) => CallableMockInstance`

为函数创建一个监听，但也可以在没有监听的情况下启动。每次调用函数时，存储其调用参数、返回值和实例。此外，你可以使用 `methods` 操纵它的行为。如果没有给出函数，`mock` 将在调用时返回 `undefined`。

```
const getApples = vi.fn(() => 0);

getApples();

expect(getApples).toHaveBeenCalled();
expect(getApples).toHaveReturnedWith(0);

getApples.mockReturnValueOnce(5);

const res = getApples();
expect(res).toBe(5);
expect(getApples).toHaveNthReturnedWith(2, 5);
```

vi.getMockedSystemTime

- **类型:** `() => Date | null`

返回使用 `setSystemTime` 设置的模拟的当前日期。如果日期没有被模拟，将返回 `null`。

vi.getRealSystemTime

- **类型:** `() => number`

使用 `vi.useFakeTimers` 时，会模拟 `Date.now` 调用。如果需要获取毫秒级的实时时间，你可以调用这个函数。

vi.mock

- **类型:** `(path: string, factory?: () => unknown) => void`

会使传递的模块的所有 `imports` 都被模拟。在 `path` 中，你可以使用配置好的 Vite 别名。对于 `vi.mock` 的调用会被提升，不管你在那调用它都会被提升到 `import` 语句之前。

- 如果定义了 `factory`，将返回其结果。工厂函数可以是异步的。你可以在内部调用 `vi.importActual` 来获取原始模块。对 `vi.mock` 的调用将被提升到文件的顶部，因此你无法访问在全局文件范围内声明的变量！
- 如果使用默认导出模拟模块，你需要在返回的工厂函数对象中提供一个 `default` key。这是 ES 模块特定的警告，因此 `jest` 文档可能会有所不同，因为 `jest` 使用 commonJS 模块。示例：

```
vi.mock("path", () => {
  return {
    default: { myDefaultKey: vi.fn() },
    namedExport: vi.fn(),
    // etc...
  };
});
```

- 如果 `__mocks__` 文件夹下存在同名文件，则所有导入都将返回其导出。例如，带有 `<root>/__mocks__/axios.ts` 文件夹的 `vi.mock('axios')` 将返回从 `axios.ts` 中导出的所有内容。
- 如果里面没有 `__mocks__` 文件夹或同名文件，将调用原始模块并对其进行模拟。(有关应用的规则，请参阅 [自动模拟算法](#)。)

vi.mocked

- **类型:** `<T>(obj: T, deep?: boolean) => MaybeMockedDeep<T>`
- **类 型 :** `<T>(obj: T, options?: { partial?: boolean; deep?: boolean }) => MaybePartiallyMockedDeep<T>`

TypeScript 的类型助手。实际上只是返回传递的对象。

当 `partial` 为 `true` 时，它会期望 `Partial<T>` 作为返回值。

```
import example from "./example";
vi.mock("./example");

test("1+1 equals 2", async () => {
  vi.mocked(example.calc).mockRestore();
  const res = example.calc(1, "+", 1);
  expect(res).toBe(2);
});
```

vi.importActual

- **类型:** `<T>(path: string) => Promise<T>`

导入模块，如果它应该被模拟，则绕过所有检查。如果你想部分模拟模块，这可能会很有用。

```
vi.mock("./example", async () => {
  const axios = await vi.importActual("./example");

  return { ...axios, get: vi.fn() };
});
```

vi.importMock

- **类型:** `<T>(path: string) => Promise<MaybeMockedDeep<T>>`

导入一个被模拟的包含其所有属性（包括嵌套属性）的模块。遵循与 `vi.mock` 相同的规则。有关应用的规则，请参阅[自动模拟算法](#)。

vi.resetAllMocks

将在所有对象监听(spies)上调用 `.mockReset()`。这将清除模拟对象历史并将其实现重置为空函数（将返回 `undefined`）。

vi.resetConfig

- **类型:** `RuntimeConfig`

如果之前调用了 `vi.setConfig`，这会将配置重置为原始状态。

vi.resetModules

- **类型:** `() => Vitest`

通过清除所有模块的缓存来重置模块的注册表。在我们对隔离测试本地状态冲突的模块时很有用。

```
import { vi } from "vitest";

beforeAll(() => {
  vi.resetModules();
});

test("change state", async () => {
  const mod = await import("./some/path");
  mod.changeLocalState("new value");
  expect(mod.getLocalState()).toBe("new value");
});

test("module has old state", async () => {
  const mod = await import("./some/path");
  expect(mod.getLocalState()).toBe("old value");
});
```

vi.restoreAllMocks

将在所有对象监听(spies)上调用 `.mockRestore()`。这将清除模拟对象历史并将其实现重置为原始实现。

vi.restoreCurrentDate

- **类型:** `() => void`

将 `Date` 恢复为系统时间。

vi.runAllTicks

- **类型:** `() => Vitest`

调用每个微任务。它们通常排列在 `process.nextTick` 中。它也将运行它们自己安排的所有微任务。

vi.runAllTimers

- **类型:** `() => Vitest`

此方法将调用每个被创建的计时器，直到计时器队列为空。这意味着在 `runAllTimers` 期间调用的每个计时器都将被触发。如果你有一个无限的区间，它会在 10000 次尝试后抛出。例如，这将输出 `1, 2, 3`：

```
let i = 0;
setTimeout(() => console.log(++i));
const interval = setInterval(() => {
  console.log(++i);
  if (i === 3) clearInterval(interval);
}, 50);

vi.runAllTimers();
```

vi.runOnlyPendingTimers

- **类型:** `() => Vitest`

此方法将调用在 `vi.useFakeTimers()` 调用之后创建的每个计时器。它不会触发在其调用期间创建的任何计时器。例如，这只会输出 `1`：

```
let i = 0;
setInterval(() => console.log(++i), 50);

vi.runOnlyPendingTimers();
```

vi.setSystemTime

- **类型:** `(date: string | number | Date) => void`

将当前日期设置为一个过去的日期。所有 `Date` 调用都将返回此日期。

有助于你测试依赖当前日期的任何内容 —— 例如，你代码中的 `luxon` 调用。

```
const date = new Date(1998, 11, 19);

vi.useFakeTimers();
vi.setSystemTime(date);
```

```
expect(Date.now()).toBe(date.valueOf());
vi.useRealTimers();
```

vi.setConfig

- **类型:** RuntimeConfig

更新当前测试文件的配置。在执行测试时，你只影响使用的值。

vi.spyOn

- **类型:** <T, K extends keyof T>(object: T, method: K, accessType?: 'get' | 'set') => MockInstance

在对象的方法或 getter/setter 上创建一个监听。

```
let apples = 0;
const obj = {
  getApples: () => 13,
};

const spy = vi.spyOn(obj, "getApples").mockImplementation(() => apples);
apples = 1;

expect(obj.getApples()).toBe(1);

expect(spy).toHaveBeenCalled();
expect(spy).toHaveReturnedWith(1);
```

vi.stubGlobal

- **类型:** (key: keyof globalThis & Window, value: any) => Vitest

给全局变量赋值。如果你使用 `jsdom` 或 `happy-dom`，也可以将值放在 `window` 对象上。

请参考["全局 Mock"](#)部分查看更多。

vi.unmock

- **类型:** (path: string) => void

从模拟注册表中删除模块。所有后续的导入调用都将返回原始模块，即使它被模拟了。

vi.useFakeTimers

- **类型:** () => Vitest

要启用模拟计时器，你需要调用此方法。它将包装对计时器的所有进一步调用（例如 `setTimeout`、`setInterval`、`clearTimeout`、`clearInterval`、`nextTick`、`setImmediate`、`clearImmediate` 和 `Date`），直到 `vi.useRealTimers()` 被调用。

它的内部实现基于 `@sinonjs/fake-timers`。

vi.useRealTimers

- **类型:** `(() => Vitest)`

当计时器结束时，你可以调用此方法，将模拟计时器恢复其原始实现。之前运行的所有计时器将不会恢复。

MockInstance Methods

getMockName

- **类型:** `(() => string)`

它返回使用 `.mockName(name)` 方法设置给模拟对象的名称。

mockClear

- **类型:** `(() => MockInstance)`

清除每一个对象模拟调用的所有信息。调用后，`spy.mock.calls`、`spy.mock.results` 将返回空数组。如果你需要清理不同断言之间的对象监听，这会很有用。

如果你希望在每次测试之前自动调用此方法，你可以在配置中启用 `clearMocks` 设置。

mockName

- **类型:** `(name: string) => MockInstance`

设置内部模拟对象名称。有助于查看哪些模拟对象导致断言失败。

mockImplementation

- **类型:** `(fn: Function) => MockInstance`

接收一个用于模拟对象实现的函数。

例如：

```
const mockFn = vi.fn().mockImplementation((apples) => apples + 1);
// or: vi.fn(apples => apples + 1);

const NelliesBucket = mockFn(0);
const BobsBucket = mockFn(1);

NelliesBucket === 1; // true
BobsBucket === 2; // true

mockFn.mock.calls[0][0] === 0; // true
mockFn.mock.calls[1][0] === 1; // true
```

mockImplementationOnce

- **类型:** `(fn: Function) => MockInstance`

接收一个只会被对象模拟函数调用一次，用于模拟对象实现的函数。可以链式调用，以便多个函数调用产生不同的结果。

```
const myMockFn = vi
  .fn()
  .mockImplementationOnce(() => true)
  .mockImplementationOnce(() => false);

myMockFn(); // true
myMockFn(); // false
```

当对象模拟函数执行完毕，它将调用 `vi.fn(() => defaultValue)` 或 `.mockImplementation(() => defaultValue)` 设置的默认实现。如果它们被调用：

```
const myMockFn = vi
  .fn(() => "default")
  .mockImplementationOnce(() => "first call")
  .mockImplementationOnce(() => "second call");

// 'first call', 'second call', 'default', 'default'
console.log(myMockFn(), myMockFn(), myMockFn(), myMockFn());
```

mockRejectedValue

- **类型:** `(value: any) => MockInstance`

当异步函数被调用时，接收一个将被拒绝（`reject`）的错误。

```
test("async test", async () => {
  const asyncMock = vi.fn().mockRejectedValue(new Error("Async error"));

  await asyncMock(); // throws "Async error"
});
```

mockRejectedValueOnce

- **类型:** `(value: any) => MockInstance`

接收一个只会被对象模拟函数拒绝一次的值。如果链式调用，每个连续调用都将拒绝传入的值。

```
test("async test", async () => {
  const asyncMock = vi
    .fn()
    .mockResolvedValueOnce("first call")
    .mockRejectedValueOnce(new Error("Async error"));

  await asyncMock(); // first call
  await asyncMock(); // throws "Async error"
});
```

mockReset

- **类型:** `() => MockInstance`

执行 `mockClear` 同样的操作，并将内部实现设置为空函数（调用时返回 `undefined`）。当你想要完全重置一个模拟对象为其初始状态时，这会很有用。

如果你希望在每次测试之前自动调用此方法，你可以在配置中启用 `mockReset` 设置。

mockRestore

- **类型:** `() => MockInstance`

执行 `mockReset` 同样的操作，并将内部实现恢复为初始的函数。

请注意，从 `vi.fn()` 恢复模拟对象会将实现设置为返回 `undefined` 的空函数。恢复 `vi.fn(impl)` 会将实现恢复为 `impl`。

如果你希望在每次测试之前自动调用此方法，我们可以在配置中启用 `restoreMocks` 设置。

mockResolvedValue

- **类型:** `(value: any) => MockInstance`

当异步函数被调用时，接收一个将被决议（`resolve`）的值。

```
test("async test", async () => {
  const asyncMock = vi.fn().mockResolvedValue(43);

  await asyncMock(); // 43
});
```

mockResolvedValueOnce

- **类型:** `(value: any) => MockInstance`

接收一个只会被对象模拟函数决议一次的值。如果链式调用，每个连续调用都将决议传入的值。

```
test("async test", async () => {
  const asyncMock = vi
    .fn()
    .mockResolvedValue("default")
    .mockResolvedValueOnce("first call")
    .mockResolvedValueOnce("second call");

  await asyncMock(); // first call
  await asyncMock(); // second call
  await asyncMock(); // default
  await asyncMock(); // default
});
```

mockReturnThis

- **类型:** `() => MockInstance`

设置内部实现返回 `this` 上下文。

mockReturnValue

- **类型:** `(value: any) => MockInstance`

接收一个调用对象模拟函数时将返回的值。

```
const mock = vi.fn();
mock.mockReturnValue(42);
mock(); // 42
mock.mockReturnValue(43);
mock(); // 43
```

mockReturnValueOnce

- **类型:** `(value: any) => MockInstance`

接收一个只会被对象模拟函数返回一次的值。如果链式调用，每个连续调用都会返回传入的值。当没有更多的 `mockReturnValueOnce` 值要使用时，调用由 `mockImplementation` 或其他 `mockReturn*` 方法指定的函数。

```
const myMockFn = vi
  .fn()
  .mockReturnValue("default")
  .mockReturnValueOnce("first call")
  .mockReturnValueOnce("second call");

// 'first call', 'second call', 'default', 'default'
console.log(myMockFn(), myMockFn(), myMockFn(), myMockFn());
```

MockInstance Properties

mock.calls

这是一个包含每个调用的所有参数的数组。数组的每一项都是该调用的参数。

如果一个函数被调用两次，并依次使用以下参数 `fn(arg1, arg2)`、`fn(arg3, arg4)`，那么 `mock.calls` 将是：

```
[["arg1", "arg2"],
 ["arg3", "arg4"],];
```

mock.lastCall

这包含最后一次调用的参数。如果未调用 `spy`，将返回 `undefined`。

mock.results

这是一个包含所有函数 `return` 的值的数组。该数组的一项是具有 `type` 和 `value` 属性的对象。可用类型有：

- `'return'` - function returned without throwing.
- `'throw'` - function threw a value.

`value` 属性包含返回值或抛出的错误。

如果函数返回 `result`，然后抛出错误，那么 `mock.results` 将是：

```
[{
  type: "return",
  value: "result",
```

```

},
{
  type: "throw",
  value: Error,
},
];

```

mock.instances

这是一个数组，包含在使用 `new` 关键字调用 `mock` 时实例化的所有实例。请注意，这是函数的实际上下文（`this`），而不是返回值。

例如，如果 `mock` 是用 `new MyClass()` 实例化的，那么 `mock.instances` 将是一个包含一个值的数组：

```

import { expect, vi } from "vitest";
const MyClass = vi.fn();
const a = new MyClass();
expect(MyClass.mock.instances[0]).toBe(a);

```

如果从构造函数返回一个值，它将不在 `instances` 数组中，而是在 `results` 中：

```

import { expect, vi } from "vitest";
const Spy = vi.fn(() => ({ method: vi.fn() }));
const a = new Spy();
expect(Spy.mock.instances[0]).not.toBe(a);
expect(Spy.mock.results[0]).toBe(a);

```

配置索引

配置

`vitest` 将读取你的项目根目录的 `vite.config.ts` 文件以匹配插件并设置为你的 Vite 应用程序。如果你想使用不同的配置进行测试，你可以：

- 创建 `vitest.config.ts`，优先级更高。
- 将 `--config` 选项传递给 CLI，例如 `vitest --config ./path/to/vitest.config.ts`。
- 在 `defineConfig` 中使用 `process.env.VITEST` 或 `mode` 属性（默认值是 `test`）在 `vite.config.ts` 中有条件的应用不同的配置。

要配置 `vitest` 本身，请在你的 Vite 配置中添加 `test` 属性。如果你使用 `vite` 的 `defineConfig` 你还需要将 [三斜线指令](#) 写在配置文件的顶部。

使用 `vite` 的 `defineConfig` 可以参考下面的格式：

```
/// <reference types="vitest" />
import { defineConfig } from "vite";

export default defineConfig({
  test: {
    // ...
  },
});
```

使用 `vitest` 的 `defineConfig` 可以参考下面的格式：

```
import { defineConfig } from "vitest/config";

export default defineConfig({
  test: {
    // ...
  },
});
```

如果有需要，你可以获取到 Vitest 的默认选项以扩展它们：

```
import { configDefaults, defineConfig } from "vitest/config";

export default defineConfig({
  test: {
    exclude: [...configDefaults.exclude, "packages/template/*"],
  },
});
```

选项

当使用单独的 `vitest.config.js` 时，如果需要，你还可以从另一个配置文件扩展 Vite 的选项：

```

import { mergeConfig } from "vite";
import { defineConfig } from "vitest/config";
import viteConfig from "./vite.config";

export default mergeConfig(
  viteConfig,
  defineConfig({
    test: {
      exclude: ["packages/template/*"],
    },
  })
);

```

提醒 除了以下选项，你还可以使用 [Vite](https://vitejs.dev/config/) 中的任何配置选项。例如，`define` 定义全局变量，或`resolve.alias` 定义别名。

include

- 类型:** `string[]`
- 默认值:** `['**/*.test, **.{js,mjs,cjs,ts,mts,cts,jsx,tsx}']`

匹配包含测试文件的 glob 规则。

exclude

- Type:** `string[]`
- Default:** `['**/node_modules/**', '**/dist/**', '**/cypress/**', '**/.idea,git,cache,output,temp/**', '**/{karma,rollup,webpack,vite,vitest,jest,ava,babel,nyc,cypress}.config.*']`

匹配排除测试文件的 glob 规则。

deps

- 类型:** `{ external?, inline? }`

对依赖关系进行内联或外联的处理

deps.external

- 类型:** `(string | RegExp)[]`
- 默认值:** `['**/node_modules/**', '**/dist/**']`

Externalize 意味着 Vite 会绕过包到原生 Node.js 中。Vite 的转换器和解析器不会应用外部依赖项，因此不会支持重新加载时的热更新。通常，`node_modules` 下的包是外部依赖。

deps.inline

- 类型:** `(string | RegExp)[] | true`
- 默认值:** `[]`

Vite 将会处理的内联模块。这有助于处理以 ESM 格式（Node 无法处理）发布 `.js` 的包。

如果为 `true`，则每个依赖项都将被内联。在 `ssr.noExternal` 中指定的所有依赖项将默认内联。

deps.fallbackCJS

- **类型:** `boolean`
- **默认值:** `false`

当一个依赖项是有效的 ESM 包时，将会尝试根据路径猜测 cjs 版本。

如果包在 ESM 和 CJS 模式下具有不同的逻辑，可能会导致一些错误的产生。

deps.registerNodeLoader

- **类型:** `boolean`
- **默认值:** `false`

使用 [实验性 Node 加载器](#) 解析 `node_modules` 中的导入，使用 Vite 解析算法。

如果禁用，你的 `alias` 和 `<plugin>.resolveId` 不会影响 `node_modules` 或 `deps.external` 中的导入。

deps.interopDefault

- **类型:** `boolean`
- **默认值:** `true`

将 CJS 模块的默认值视为命名导出。

benchmark

- **类型:** `{ include?, exclude?, ... }`

运行 `vitest bench` 时使用的选项。

benchmark.include

- **类型:** `string[]`
- **默认值:** `['**/*.{bench,benchmark}.{js,mjs,cjs,ts,mts,cts,jsx,tsx}']`

匹配包含基准测试文件的 glob 规则。

benchmark.exclude

- **类型:** `string[]`
- **默认值:** `['node_modules', 'dist', '.idea', '.git', '.cache']`

匹配排除基准测试文件的 glob 规则。

benchmark.includeSource

- **类型:** `string[]`
- **默认值:** `[]`

匹配包含内联基准测试文件的 glob 规则。此选项类似于 `includeSource`。

定义后，Vitest 将运行所有匹配的文件，其中包含 `import.meta.vitest`。

benchmark.reporters

- **类型:** `Arrayable<BenchmarkBuiltinReporters | Reporter>`
- **默认值:** `'default'`

用于定义输出的自定义报告器。它可以包含一个或多个内置报告名称、报告实例和(或)自定义报告的路径。

benchmark.outputFile

- **类型:** `string | Record<string, string>`

当指定了 `--reporter=json` 选项时，可以将基准测试结果写入文件。通过提供对象而不是字符串，您可以在使用多个报告器时定义单独的输出。

通过 CLI 命令提供对象，请使用以下语法：`--outputFile.json=./path --outputFile.junit=./other-path`。

alias

- **类型:** `Record<string, string> | Array<{ find: string | RegExp, replacement: string, customResolver?: ResolverFunction | ResolverObject }>`

在测试内部运行时定义自定义别名。它们将与来自 `resolve.alias` 的别名合并。

globals

- **类型:** `boolean`
- **默认值:** `false`

默认情况下，`vitest` 不显式提供全局 API。如果你更倾向于使用类似 jest 中的全局 API，可以将 `--globals` 选项传递给 CLI 或在配置中添加 `globals: true`。

```
// vite.config.ts
import { defineConfig } from "vitest/config";

export default defineConfig({
  test: {
    globals: true,
  },
});
```

为了可以让全局 API 支持 Typescript，请将 `vitest/globals` 添加到 `tsconfig.json` 中的 `types` 选项中

```
// tsconfig.json
{
  "compilerOptions": {
    "types": ["vitest/globals"]
  }
}
```

如果你已经在项目中使用 `unplugin-auto-import`，你也可以直接用它来自动导入这些 API。

```
// vite.config.ts
import { defineConfig } from "vitest/config";
import AutoImport from "unplugin-auto-import/vite";

export default defineConfig({
  plugins: [
    AutoImport({
      imports: ["vitest"],
      dts: true, // generate TypeScript declaration
    }),
  ],
});
```

environment

- 类型:** 'node' | 'jsdom' | 'happy-dom' | 'edge-runtime' | string
- 默认值:** 'node'

Vitest 中的默认测试环境是一个 Node.js 环境。如果你正在构建 Web 端应用程序，你可以使用 `jsdom` 或 `happy-dom` 这种类似浏览器(browser-like)的环境来替代 Node.js。如果你正在构建边缘计算函数，你可以使用 `edge-runtime` 环境

你可以通过在文件顶部添加包含 `@vitest-environment` 的文档块或注释，为某个测试文件中的所有测试指定环境：

文档块格式：

```
/**
 * @vitest-environment jsdom
 */

test("use jsdom in this test file", () => {
  const element = document.createElement("div");
  expect(element).not.toBeNull();
});
```

注释格式：

```
// @vitest-environment happy-dom

test("use happy-dom in this test file", () => {
  const element = document.createElement("div");
  expect(element).not.toBeNull();
});
```

为了与 Jest 兼容，还存在一个配置 `@jest-environment`：

```
/**
 * @jest-environment jsdom
 */

test("use jsdom in this test file", () => {
  const element = document.createElement("div");
  expect(element).not.toBeNull();
});
```

如果你使用 `--no-threads` 标志运行 Vitest，你的测试将按以下顺序运行：`node`, `jsdom`, `happy-dom`, `edge-runtime`, `custom environments`。这意味着，具有相同环境的每个测试都组合在一起，但仍按顺序运行。

从 0.23.0 开始，你还可以定义自定义环境。当使用非内置环境时，Vitest 将尝试加载包 `vitest-environment-${name}`。该包应导出一个具有 `Environment` 属性的对象：

```
import type { Environment } from "vitest";

export default <Environment>{
  name: "custom",
  setup() {
    // custom setup
    return {
      teardown() {
        // called after all tests with this env have been run
      },
    },
  },
};
```

Vitest 还通过 `vitest/environments` 入口导出 `builtinEnvironments`，以防你只想扩展它。你可以在 [测试环境指南](#) 中阅读有关扩展测试环境的更多信息。

environmentOptions

- **类型:** `Record<'jsdom' | string, unknown>`
- **默认值:** `{}`

这些选项被传递给当前 `environment` 的 `setup` 方法。默认情况下，如果你将其用作测试环境，则只能配置 JSDOM 选项。

update

- **类型:** `boolean`
- **默认值:** `false`

更新快照文件。这将更新所有更改的快照并删除过时的快照。

watch

- **Type:** `boolean`
- **Default:** `true`

启动监听模式

root

- **类型:** `string`

项目的根目录

reporters

- **类型:** `Reporter | Reporter[]`
- **默认值:** `'default'`

用于输出的自定义 reporters。Reporters 可以是 [一个 Reporter 实例](#) 或选择内置的 reporters 字符串：

- `'default'` - 当他们经过测试套件
- `'verbose'` - 保持完整的任务树可见
- `'dot'` - 将每个任务显示为一个点
- `'junit'` - JUnit XML 报告器
- `'json'` - 给出一个简单的 JSON 总结
- 自定义报告的路径 (例如 `'./path/to/reporter.ts'`, `'@scope/reporter'`)

outputTruncateLength

- **类型:** `number`
- **默认值:** `80`

指定截断输出差异的字符行数，最多 80 个字符。你可能希望对此进行调整，取决于你的终端窗口宽度。

outputDiffLines

- **类型:** `number`
- **默认值:** `15`

指定输出差线的数量，最多 `15` 个。

outputFile

- **类型:** `string | Record<string, string>`

当指定 `--reporter=json` 或 `--reporter=junit` 时，将测试结果写入一个文件。通过提供对象而不是字符串，你可以在使用多个报告器时定义单独的输出。

要通过 CLI 命令提供对象，请使用以下语法：`--outputFile.json=./path --outputFile.junit=./other-path`。

threads

- **类型:** `boolean`
- **默认值:** `true`

通过使用 `tinypool` ([Piscina](#) 的轻量级分支) 可以启用多线程。

警告 此选项与 Jest 的 `--runInBand` 不同。 Vitest 使用工作线程不仅可以并行运行测试，还可以提供隔离。通过禁用此选项，你的测试将按顺序运行，但在相同的全局上下文中，因此你必须自己提供隔离。

如果你依赖全局状态（前端框架通常这样做）或者你的代码依赖于为每个测试单独定义的环境，这可能会导致各种问题。但是可以提高你的测试速度（最多快 3 倍），这不一定依赖于全局状态或可以轻松绕过它。

maxThreads

- **类型:** `number`
- **默认值:** 可用的 CPU 数量

允许的最大线程数。你也可以使用 `VITEST_MAX_THREADS` 环境变量。

minThreads

- **类型:** `number`
- **默认值:** 可用的 CPU 数量

允许的最小线程数。你也可以使用 `VITEST_MIN_THREADS` 环境变量。

testTimeout

- **类型:** `number`
- **默认值:** `5000`

测试的默认超时时间（以毫秒为单位）。

hookTimeout

- **类型:** `number`
- **默认值:** `10000`

钩子(hook)的默认超时时间（以毫秒为单位）。

teardownTimeout

- **Type:** `number`
- **Default:** `1000`

Vitest 关闭时等待关闭的默认超时时间，以毫秒为单位

silent

- **类型:** `boolean`
- **默认值:** `false`

静默模式下启动测试。

setupFiles

- **类型:** `string | string[]`

`setup` 文件的路径。它们将运行在每个测试文件之前。

你可以在内部使用 `process.env.VITEST_WORKER_ID` (类似整数的字符串) 来区分线程 (如果 `threads: false`, 那么这个值将永远是 `1`)。

提醒 请注意, 如果你正在运行 `[`--no-threads`](#threads)`, 则此设置文件将在同一全局范围内多次运行。这意味着, 你在每次测试之前都在访问同一个全局对象, 因此请确保你做的事情没有超出你的需要。

比如, 你可能依赖于一个全局变量 :

```
import { config } from "@some-testing-lib";

if (!globalThis.defined) {
  config.plugins = [myCoolPlugin];
  computeHeavyThing();
  globalThis.defined = true;
}

// hooks are reset before each suite
afterEach(() => {
  cleanup();
});

globalThis.resetBeforeEachTest = true;
```

globalSetup

- **类型:** `string | string[]`

全局的 `setup` 文件的路径, 相对于项目的根目录。

全局的 `setup` 文件可以导出命名函数 `setup` 和 `teardown` 或返回拆卸函数的 `default` 函数 ([示例](#))。

`info` 提示 可以存在多个 `globalSetup`。`setup` 和 `teardown` 依次执行, 而 `teardown` 则以相反的顺序执行。

警告 请注意, 全局设置在不同的全局范围内运行, 因此你的测试无权访问此处定义的变量。

watchExclude

- **类型:** `string[]`
- **默认值:** `['**/node_modules/**', '**/dist/**']`

触发监视重新运行时要忽略的文件路径的全局 glob 模式。

forceRerunTriggers

- **类型:** `string[]`
- **默认值:** `['**/package.json/**', '**/vitest.config.*/**', '**/vite.config.*/**']`

将触发整个套件重新运行的文件路径的全局 glob 模式。如果在 git diff 中找到触发器，则与 --changed 参数配对时，将运行整个测试套件。

如果你正在测试调用 CLI 命令时很有用，因为 Vite 无法构建模块依赖图：

```
test("execute a script", async () => {
  // Vitest cannot rerun this test, if content of `dist/index.js` changes
  await execa("node", ["dist/index.js"]);
});
```

提醒 确保你的的文件未被 `watchExclude` 排除。

isolate

- 类型:** `boolean`
- 默认值:** `true`

是否为每个测试文件构建隔离环境。如果你禁用 `--threads`，它将不会工作。

coverage

- 类型:** `CoverageC8Options | CoverageIstanbulOptions`
- 默认值:** `undefined`

你可以使用 `c8` 或 `istanbul` 收集测试覆盖率。

provider

- 类型:** `'c8' | 'istanbul'`
- 默认值:** `'c8'`

使用 `provider` 选择收集测试覆盖率的工具。

CoverageC8Options

设置了 `provider: 'c8'` 时启用。测试覆盖率选项被传递给 `c8`。

CoverageIstanbulOptions

设置了 `provider: 'istanbul'` 时启用。

include

- Type:** `string[]`
- Default:** `['**']`

List of files included in coverage as glob patterns

exclude

- 类型:** `string[]`

- **默认值:** `['coverage/**', 'dist/**', 'packages/*/test{,s}/**', '**/*.d.ts', 'cypress/**', 'test{,s}/**', 'test{,-*}.{js,cjs,mjs,ts,tsx,jsx}', '**/*{.,-}test.{js,cjs,mjs,ts,tsx,jsx}', '**/*{.,-}spec.{js,cjs,mjs,ts,tsx,jsx}', '**/_tests_/**', '**/{karma,rollup,webpack,vite,vitest,jest,ava,babel,nyc,cypress}.config.{js,cjs,ts}', '**/.{eslint,mocha,prettier}rc.{js,cjs,yml}']`

使用全局模式排除在覆盖范围之外的文件列表。

skipFull

- **类型:** `boolean`
- **默认值:** `false`

是否显示具有 100% 语句、分支和函数的测试覆盖率的文件。

perFile

- **类型:** `boolean`
- **默认值:** `false`

检查每个文件的阈值。

lines

- **类型:** `number`

行的阈值。

functions

- **类型:** `number`

函数的阈值。

branches

- **类型:** `number`

分支的阈值。

statements

- **类型:** `number`

语句的阈值。

ignoreClassMethods

- **类型:** `string[]`
- **默认值:** `[]`

设置忽略测试覆盖率的类方法名称的数组。

watermarks

- **类型:**

```
{
  statements?: [number, number],
  functions?: [number, number],
  branches?: [number, number],
  lines?: [number, number]
}
```

- 默认值:

```
{
  statements: [50, 80],
  functions: [50, 80],
  branches: [50, 80],
  lines: [50, 80]
}
```

指定语句、行、分支和函数的水印位置。

all

- 类型: `boolean`
- 默认值: `false`

是否包含所有文件，包括未测试的文件并包含在测试报告中。

testNamePattern

- 类型 `string | RegExp`

使用与模式匹配的全名运行测试。如果你将 `OnlyRunThis` 添加到此属性，将跳过测试名称中不包含单词 `OnlyRunThis` 的测试。

```
import { expect, test } from "vitest";

// run
test("OnlyRunThis", () => {
  expect(true).toBe(true);
});

// skipped
test("doNotRun", () => {
  expect(true).toBe(true);
});
```

open

- 类型: `boolean`
- 默认值: `false`

打开 Vitest UI (WIP: 赞助者计划可用)

api

- 类型: `boolean | number`
- 默认值: `false`

提供 API 服务的端口。当设置为 `true` 时， 默认端口为 51204

clearMocks

- **类型:** `boolean`
- **默认值:** `false`

是否在每次测试前对所有监听(Spy)调用 `.mockClear()`。这将清除模拟历史记录，但不会将其实现重置为默认值。

mockReset

- **类型:** `boolean`
- **默认值:** `false`

是否在每次测试之前对所有监听(Spy)调用 `.mockReset()`。 这将清除模拟历史并将其实现重置为空函数（将返回 `undefined`）。

restoreMocks

- **类型:** `boolean`
- **默认值:** `false`

是否在每次测试之前对所有监听(Spy)调用 `.mockRestore()`。 这将清除模拟历史并将其实现重置为原始历史。

transformMode

- **类型:** `{ web?, ssr? }`

决定模块的转换方式。

transformMode.ssr

- **类型:** `RegExp[]`
- **默认值:** `[/\.(cm|jt)sx?|json)$/]`

对指定的文件使用 SSR 转换管道。

Vite 插件在处理这些文件时会收到 `ssr: true` 标志。

transformMode.web

- **类型:** `RegExp[]`
- **默认值:** 除了 `transformMode.ssr` 以外的所有文件

首先会进行正常的转换管道（针对浏览器），然后进行 SSR 重写以在 Node 中运行代码。

Vite 插件在处理这些文件时会收到 `ssr: false` 标志。

当你使用 JSX 作为 React 以外的组件模型（例如 Vue JSX 或 SolidJS）时，你可能需要进行如下配置以使 `.tsx` / `.jsx` 转换为客户端组件：

```
import { defineConfig } from "vitest/config";
export default defineConfig({
```

```
test: {
  transformMode: {
    web: [/^.[jt]sx$/],
  },
},
});
```

snapshotFormat

- **类型:** `PrettyFormatOptions`

测试快照的格式选项。这些选项被传递给 `pretty-format`。

resolveSnapshotPath

- **类型:** `(testPath: string, snapExtension: string) => string`
- **默认值:** 存储快照文件在 `__snapshots__` 目录

覆盖快照的默认路径。例如，要在测试文件旁边存储一下快照：

```
import { defineConfig } from "vitest/config";

export default defineConfig({
  test: {
    resolveSnapshotPath: (testPath, snapExtension) => testPath + snapExtension,
  },
});
```

allowOnly

- **类型:** `boolean`
- **默认值:** `false`

允许标记为 `only` 的测试和套件。

dangerouslyIgnoreUnhandledErrors

- **类型:** `boolean`
- **默认值:** `false`

忽略发生的任何未处理的错误。

passWithNoTests

- **类型:** `boolean`
- **默认值:** `false`

如果没有找到测试，Vitest 不会失败。

logHeapUsage

- **Type:** `boolean`
- **Default:** `false`

每次测试后显示堆的使用情况。用于调试内存是否泄漏。

CSS

- **类型:** `boolean | { include?, exclude?, modules? }`

配置是否应处理 CSS。排除后，CSS 文件将被替换为空字符串以绕过后续处理。CSS 模块将返回一个代理以不影响运行时。

css.include

- **类型:** `RegExp | RegExp[]`
- **默认值:** `[]`

将返回匹配正则表达式并将由 Vite 管道处理的实际 CSS 文件。

如果需要处理所有 CSS 文件，请使用 `/.+/?`。

css.exclude

- **类型:** `RegExp | RegExp[]`
- **默认值:** `[]`

将返回匹配正则表达式的空 CSS 文件。

css.modules

- **类型:** `{ classNameStrategy? }`
- **默认值:** `{}`

css.modules.classNameStrategy

- **类型:** `'stable' | 'scoped' | 'non-scoped'`
- **默认值:** `'stable'`

如果你决定处理 CSS 文件，你可以配置 CSS 模块中的类名是否在限定范围内。默认情况下，Vitest 会导出一个代理，绕过 CSS 模块处理。你可以选择以下选项之一：

- `stable`：类名将生成为 `_${name}_${hashedFilename}`，这意味着如果 CSS 内容发生变化，生成的类将保持不变，但如果文件名被修改，或者文件名将发生变化 被移动到另一个文件夹。如果你使用快照功能，此设置很有用。
- `scoped`：类名将照常生成，遵照 `css.modules.generateScopeName` 方法，如果你有的话。默认情况下，文件名将生成为 `_${name}_${hash}`，其中 `hash` 包括文件名和文件内容。
- `non-scoped`：类名将保留 CSS 中定义的名称。

在默认的情况下，Vitest 导出代理会绕过 CSS 模块处理。如果你依赖类的 CSS 属性，就必须使用 `include` 选项启用 CSS 处理。

maxConcurrency

- **类型:** `number`
- **默认值:** `5`

使用 `test.concurrent` 标记允许同时运行的最大测试数量。

当出现可用插槽时，超过此限制的测试将排队运行。

cache

- **类型:** `false | { dir? }`

配置 Vitest 缓存策略的选项。目前 Vitest 为测试结果存储缓存，以便先运行更长且失败的测试。

cache.dir

- **类型:** `string`
- **默认值:** `node_modules/.vitest`

缓存目录的路径。

sequence

- **类型:** `{ sequencer?, shuffle?, seed?, hooks? }`

配置测试运行顺序的选项。

sequence.sequencer

- **类型:** `TestSequencerConstructor`
- **默认值:** `BaseSequencer`

定义分片和排序的自定义类。你可以从 `vitest/node` 扩展 `BaseSequencer`，如果你只需要重新定义 `sort` 和 `shard` 方法之一，但两者都应该存在。

分片是在排序之前进行的，并且只有提供了 `--shard` 选项的情况下才会生效。

sequence.shuffle

- **类型:** `boolean`
- **默认值:** `false`

如果你希望测试随机运行，可以使用此选项或 CLI 参数 `--sequence.shuffle` 启用它。

Vitest 通常使用缓存对测试进行排序，因此长时间运行的测试会更早开始 - 这会使测试运行得更快。如果你的测试将以随机顺序运行，你将失去这种性能改进，但跟踪意外依赖于先前运行的测试可能很有用。

sequence.seed

- **类型:** `number`
- **默认值:** `Date.now()`

如果测试以随机顺序运行，则设置随机化种子。

sequence.hooks

- **类型:** `'stack' | 'list' | 'parallel'`
- **默认值:** `'parallel'`

更改钩子的执行顺序。

- `stack` 将以相反的顺序排列 "after" 钩子，"before" 钩子将按照它们定义的顺序运行
- `list` 将按照定义的顺序对所有钩子进行排序
- `parallel` 将并行运行单个组中的钩子（父套件中的钩子仍将在当前套件的钩子之前运行）

typecheck

用于配置 [typechecking](#) 测试环境的选项。

typecheck.checker

- **类型:** `'tsc' | 'vue-tsc' | string`
- **默认值:** `tsc`

设置类型检查的检测器。Vitest 将根据类型生成具有某些参数的进程，以便于解析。Checker 应该实现与 `tsc` 相同的输出格式。

你需要安装一个包才能使用 typecheker:

- `tsc` requires `typescript` package
- `vue-tsc` requires `vue-tsc` package

你还可以将路径传递到自定义二进制文件或命令名称，该路径会产生与 `tsc --noEmit --pretty false` 相同的输出。

typecheck.include

- **类型:** `string[]`
- **默认值:** `['**/*.{test,spec}-d.{ts,js}']`

匹配包含测试文件的 glob 规则。

typecheck.exclude

- **类型:** `string[]`
- **默认值:** `['**/node_modules/**', '**/dist/**', '**/cypress/**', '**/.{idea,git,cache,output,temp}/**']`

匹配排除测试文件的 glob 规则。

typecheck.allowJs

- **类型:** `boolean`
- **默认值:** `false`

检查有 `@ts-check` 注释的 JS 文件。 如果你在 `tsconfig` 中启用它，则不会覆盖它。

typecheck.ignoreSourceErrors

- **类型:** `boolean`
- **默认值:** `false`

如果 Vitest 在测试文件之外发现错误，不要失败。这根本不会向你显示非测试错误。

默认情况下，如果 Vitest 发现源错误，它将测试套件中抛出失败。

typecheck.tsconfig

- **类型:** `string`
- **默认值:** *tries to find closest tsconfig.json*

自定义 `tsconfig` 的路径，相对于项目根目录。

命令行界面

命令

`vitest`

在当前目录中启动 Vitest 将进入浏览模式，它运行在开发环境并且在 CI 中自动运行。

你可以通过添加参数作为过滤器来运行测试文件，比如：

```
vitest foobar
```

只运行包含 `foobar` 路径下的测试文件。

`vitest run`

在没有浏览模式的情况下执行单次运行。

`vitest watch`

运行所有测试套件，但要注意更改并在更改时重新运行测试。类似于在没有命令的情况下调用 `vitest`。在 CI 环境中，此命令将回退到 `vitest run`。

`vitest dev`

`vitest watch` 的别名。

`vitest related`

仅运行涵盖源文件列表的测试。适用于静态惰性导入(例如, `import('./index.ts')` 或者 `import index from './index.ts'`)，但不适用于动态导入(例如, `import(filepath)`)。所有文件都应该相对于根文件夹。

与 `lint-staged` 或你的 CI 设置一起运行很有用。

```
vitest related /src/index.ts /src/hello-world.js
```

`vitest clean cache`

清空缓存目录。

选项

选项	描述
<code>-v, --version</code>	显示版本号
<code>-r, --root <path></code>	定义项目根目录
<code>-c, --config <path></code>	定义配置文件路径

选项	描述
<code>-u, --update</code>	更新快照
<code>-w, --watch</code>	智能即时浏览模式
<code>-t, -- testNamePattern <pattern></code>	使用与模式匹配的全名运行测试
<code>--dir <path></code>	用于扫描测试文件的基本目录
<code>--ui</code>	启用 UI
<code>--open</code>	如果启用, 则自动打开 UI (default: <code>true</code>)
<code>--api [api]</code>	服务 API, 可用选项: <code>--api.port <port></code> 、 <code>--api.host [host]</code> 和 <code>--api.strictPort</code>
<code>--threads</code>	启用线程 (default: <code>true</code>)
<code>--silent</code>	测试的控制台输出
<code>--isolate</code>	为每个测试文件隔离环境 (default: <code>true</code>)
<code>--reporter <name></code>	选择报告器: <code>default</code> 、 <code>verbose</code> 、 <code>dot</code> 、 <code>junit</code> 、 <code>json</code> 或自定义报告器的路径
<code>--outputTruncateLength <length></code>	使用 <code><length></code> 指定截断输出差异的字符行数
<code>--outputDiffLines <lines></code>	使用 <code><lines></code> 指定输出差线的数量
<code>--outputFile <filename/-s></code>	当还指定了 <code>--reporter=json</code> 或 <code>--reporter=junit</code> 选项时, 将测试结果写入文件 通过 [cac's dot notation] 可以为多个报告器指定单独的输出
<code>--coverage</code>	使用 c8 进行覆盖率输出
<code>--run</code>	不使用浏览模式
<code>--mode</code>	覆盖 Vite 模式 (default: <code>test</code>)
<code>--mode <name></code>	覆盖 Vite 模式 (default: <code>test</code>)
<code>--globals</code>	注入全局 API
<code>--dom</code>	使用 happy-dom 模拟浏览器 API
<code>--browser</code>	在浏览器中运行测试
<code>--environment <env></code>	设置运行的环境 (default: <code>node</code>)
<code>--passWithNoTests</code>	未找到测试时通过
<code>--allowOnly</code>	允许标记为 <code>only</code> 的测试和套件 (default: <code>false</code> in CI, <code>true</code> otherwise)
<code>--dangerouslyIgnoreUnhandledErrors</code>	忽略发生的任何未处理的错误
<code>--changed [since]</code>	运行受更改文件影响的测试 (default: <code>false</code>), 更多内容请查看 文档
<code>--shard <shard></code>	配置测试分片

选项	描述
--sequence	定义运行测试的顺序，使用 [cac's dot notation] 来指定选项（例如，使用 <code>--sequence.shuffle</code> 以随机顺序运行测试）
--no-color	从控制台输出中禁用颜色
-h, --help	显示可用的 CLI 选项

变更

- **类型:** boolean | string
- **默认值:** false

再次运行有变更的测试文件。如果未发生变更，它将再次运行未提交的变更（包括分段和非分段）。

在上次提交的变更模式下，再次运行测试，你可以使用 `--changed HEAD~1`。也可以使用提交哈希或者分支名称。

分片

假如找到匹配项，如果配置了 `forceRerunTriggers` 选项，那么它将运行整个测试套件。

- **类型:** string
- **默认值:** disabled

要以 `<index> / <count>` 格式来执行的测试套件分片，其中

- `count` 是一个正整数，它表示拆分测试分片的数量
- `index` 是一个正整数，它表示拆分测试分片的索引

该命令将所有测试通过 `count` 来拆分，并仅运行恰好位于 `index` 部分的测试。比如，将你的测试套件拆分为三份，可以这样使用：

```
vitest run --shard=1/3
vitest run --shard=2/3
vitest run --shard=3/3
```

警告 你不能使用 `--watch` 选项（在开发环境下默认启用）。

跟其他的测试框架进行对比

Jest

[Jest](#) 通过为大多数的 JavaScript 项目提供了开箱即用的测试支持，填补了测试框架的空白，有着舒适的 API（例如 `it` 和 `expect`），以及大多数所需要的全套测试功能（例如快照，对象模拟，代码测试覆盖率）。我们十分感谢 Jest 团队和社区创建了完美的 API，并推动了很多测试模式的发展，这些模式现在已经成为 Web 生态系统的标准。现在也可以在 Vite 中使用 Jest。[@sodatea](#) 正在编写 [vite-jest](#)，准备提供一套完美跟 Vite 集成的 Jest。最后一个 [Jest 中的阻碍](#) 也已经解决，所以 Jest 也是 Vite 生态下单元测试框架的可选项。然而，在我们将 Vite 作为常见的 Web 工具（TypeScript，JSX，流行的 UI 框架）所支持的工具里面，Jest 有着重复的复杂性。如果你的项目由 Vite 驱动，那么配置和维护两个不同的容器是一件极其不合理的操作。使用 Vitest，你就可以将开发，构建和测试环境的配置定义为单个容器，共享相同的插件和 `vite.config.js`。即使项目并不是由 Vite 提供支持（例如，使用了 esbuild 或者 rollup 进行构建），Vitest 也将会是一个有趣的选择，因为它为你的单元测试提供更快的运行速度，并且由于使用 Vite 即时热模块重载（HMR）的默认监听模式，你的开发体验将会有飞跃的提升。Vitest 与大多数 Jest API 和生态系统库都有较好的兼容性，因此在大多数项目中，我们应该可以无缝的将 Jest 替换成 Vitest。

Cypress

[Cypress](#) 是基于浏览器的测试工具，是 Vitest 的补充工具之一。如果你想使用 Cypress，建议将 Vitest 用于测试项目中非浏览器逻辑，将 Cypress 用于测试依赖浏览器的逻辑。

Cypress 是著名的端到端测试工具，他们[最新的组件测试运行器](#) 对测试 Vite 组件有很好的支持，是测试任何在浏览器中渲染的东西的理想选择。

基于浏览器运行测试的框架，例如 Cypress 和 Web Test，会捕获到 Vitest 无法捕获的问题，因为他们都是使用真实的浏览器和 APIs。

Cypress 的测试更加专注于确定元素是否可见，是否可以访问和交互。Cypress 专门为 UI 开发和测试而构建，它的开发体验趋向于测试你的视觉组件，你会看到程序的组件和测试报告一起出现。测试完成后，组件将保持交互状态，您可以使用浏览器开发工具调试发生的任何故障。

相比之下，Vitest 专注于为非浏览器逻辑提供最佳的开发体验。像 Vitest 这样的基于 Node.js 的测试框架支持各种实现部分浏览器环境的第三方包，例如 `jsdom`，他们实现的足够多，就可以让我们快速的对于任何引用浏览器 APIs 的代码进行单元测试。其代价是，这些浏览器环境在实现上有局限性。例如，[jsdom 缺少相当数量的特性](#)，诸如 `window.navigation` 或者布局引擎（`offsetTop` 等）。

最后，与 Web Test Runner 相比，Cypress 更像是一个 IDE 而不是测试框架，因为您还可以在浏览器中看到真实呈现的组件，以及它的测试结果和日志。

Cypress 还一直在[尝试将 Vite 集成进他们自己的产品中](#)：使用 [Vitesse](#) 重新构建他们的应用程序的 UI，并使用 Vite 来测试驱动他们项目的开发。

我们认为 Cypress 不是对业务代码进行单元测试好选择，但使用 Cypress（用于端对端和组件测试）配合 Vitest（用于非浏览器逻辑的单元测试）将满足你应用程序的测试需求。

Web Test Runner

[@web/test-runner](#) 在无头浏览器中进行测试，提供与你的 Web 应用程序相同的运行环境，而不需要模拟浏览器的 API 和 DOM。虽然没有像 Cypress 那样显示用于单步执行测试的 UI，但也使得我们可以使用 devtools 在浏览器中进行调试。[@web/test-runner](#) 有一个监听模式，但是不如 Vitest 智能，而且有时候不一定运行你想要的那个测试。要在 Vite 项目中使用 [@web/test-runner](#)，有一个 [plugin](#)，尽管某些功能还 [尚不可用](#)（例如在测试中更改视口大小）。同时 [@web/test-runner](#) 不包含断言或对象模拟库，因此要靠你自己来添加它们。

uvu

[uvu](#) 是用于 Node.js 和浏览器的测试框架。它在单线程中进行测试，所以测试没有被隔离，可以跨文件测试，而 Vitest 使用 Worker 线程来隔离测试并且并发运行它们。为了转换我们的代码，[uvu](#) 依赖于 `require` 和 `loader` 方法。但是 Vitest 使用 [Vite](#)，因此文件可以使用 Vite 插件系统的全部功能进行代码的转换。在我们将 Vite 作为常见的 Web 工具（TypeScript，JSX，常见的 UI 框架）所支持的工具里面，[uvu](#) 有着重复的复杂性。但是如果项目由 Vite 提供支持，那么配置和维护两个不同的容器是一件极其不合理的操作。使用 Vitest，就可以将开发、构建和测试环境的配置定义为单个容器，共享插件和 `vite.config.js`。[uvu](#) 不提供智能监听模式来重新运行已更改的测试，但 Vitest 为你提供了惊人开发体验，这都归功于默认的监听模式中使用了 Vite 的即时热重载模块（HMR）。[uvu](#) 是运行简单测试的快速选项，但对于更复杂的测试和项目，Vitest 可以更快、更可靠。

测试覆盖率

Vitest 通过 `c8` 支持本机代码覆盖率。同时也支持 `istanbul`。

覆盖率提供者

提醒 从 Vitest v0.22.0 开始支持

`c8` 和 `istanbul` 的支持都是可选的。默认情况下，启用 `c8`。

你可以通过将 `test.coverage.provider` 设置为 `c8` 或 `istanbul` 来选择覆盖工具：

```
// vite.config.ts
import { defineConfig } from 'vitest/config'

export default defineConfig({
  test: {
    coverage: {
      provider: 'istanbul' // or 'c8'
    },
  },
})
```

当你启动 Vitest 进程时，它会提示你自动安装相应的支持包。

或者，如果你更喜欢手动安装它们：

```
# For c8
npm i -D @vitest/coverage-c8

# For istanbul
npm i -D @vitest/coverage-istanbul
```

覆盖率配置

要在启用的情况下进行测试，你可以在 CLI 中传递 `--coverage` 标志。

```
{
  "scripts": {
    "test": "vitest",
    "coverage": "vitest run --coverage"
  }
}
```

要对其进行配置，需要在配置文件中设置 `test.coverage` 选项：

```
// vite.config.ts
import { defineConfig } from 'vitest/config'

export default defineConfig({
  test: {
```

```
coverage: {
    reporter: ['text', 'json', 'html'],
},
},
})
```

自定义覆盖率提供者

也可以通过将对象传递给 `test.coverage.provider` 来配置你的自定义覆盖率提供者：

```
// vite.config.ts
import { defineConfig } from 'vitest/config'
import CustomCoverageProvider from 'my-custom-coverage-provider'

export default defineConfig({
  test: {
    coverage: {
      provider: CustomCoverageProvider()
    },
  },
})
```

请参阅类型定义查看有关详细信息。

调试

Terminal

在非集成开发环境下，为了调试一个测试文件，你可以使用 `ndb`。仅仅在你的代码的任何位置添加一个 `debugger` 语句，然后运行 `ndb`：

```
# 全局安装 ndb
npm install -g ndb

# 或者使用 yarn
yarn global add ndb

# 在启用 debugger 的情况下运行测试
ndb npm run test
```

VSCode

要在 VSCode 中调试测试文件，请创建以下启动配置。

```
{
  // 想了解更多的信息，请访问：https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Debug Current Test File",
      "autoAttachChildProcesses": true,
      "skipFiles": ["<node_internals>/**", "**/node_modules/**"],
      "program": "${workspaceRoot}/node_modules/vitest/vitest.mjs",
      "args": ["run", "${relativeFile}"],
      "smartStep": true,
      "console": "integratedTerminal"
    }
  ]
}
```

然后在调试选项卡中确保选择 'Debug Current Test File'，然后你可以打开要调试的测试文件并按 F5 开始调试。

IntelliJ IDEA

创建一个 'Node.js' 运行配置。使用以下配置在调试模式下运行所有测试：

Setting	Value
Working directory	/path/to/your-project-root
JavaScript file	./node_modules/vitest/vitest.mjs
Application parameters	run --threads false

然后在调试模式下运行此配置。IDE 将在编辑器中设置的 JS/TS 断点处停止。

[Go to TOC](#)

测试环境

Vitest 提供 `environment` 选项以在特定环境中运行代码。你可以使用 `environmentOptions` 选项修改环境的行为方式。

默认情况下，你可以使用这些环境：

- `node` 为默认环境
- `jsdom` 通过提供 Browser API 模拟浏览器环境，使用 `jsdom` 包
- `happy-dom` 通过提供 Browser API 模拟浏览器环境，被认为比 `jsdom` 更快，但缺少一些 API，使用 `happy-dom` 包
- `edge-runtime` 模拟 Vercel 的 `edge-runtime`，使用 `@edge-runtime/vm` 包

从 0.23.0 开始，你可以创建自己的包来扩展 Vitest 环境。为此，请创建名为 `vitest-environment-${name}` 的包。该包应导出一个具有 `Environment` 属性的对象：

```
import type { Environment } from 'vitest'

export default <Environment>{
  name: 'custom',
  setup() {
    // custom setup
    return {
      teardown() {
        // called after all tests with this env have been run
      }
    }
  }
}
```

你还可以通过 `vitest/environments` 访问默认的 Vitest 环境：

```
import { builtinEnvironments, populateGlobal } from 'vitest/environments'
console.log(builtinEnvironments) // { jsdom, happy-dom, node, edge-runtime }
```

Vitest 还提供了 `populateGlobal` 实用函数，可用于将属性从对象移动到全局命名空间：

```
interface PopulateOptions {
  // should non-class functions be bind to the global namespace
  bindFunctions?: boolean
}

interface PopulateResult {
  // a list of all keys that were copied, even if value doesn't exist on original
  // object
  keys: Set<string>
  // a map of original object that might have been overriden with keys
  // you can return these values inside `teardown` function
  originals: Map<string | symbol, any>
}
```

```
export function populateGlobal(global: any, original: any, options:  
  PopulateOptions): PopulateResult
```

[Go to TOC](#)

扩展断言(Matchers)

由于 Vitest 兼容 Chai 和 Jest，所以可以根据个人喜好使用 `chai.use` API 或者 `expect.extend`。

本文将以 `expect.extend` 为例探讨扩展断言。如果你对 Chai 的 API 更感兴趣，可以查看[它的指南](#)。

为了扩展默认的断言，可以使用对象包裹断言的形式调用 `expect.extend` 方法。

```
expect.extend({
  toBeFoo(received, expected) {
    const { isNot } = this
    return {
      // 请勿根据 isNot 参数更改你的 "pass" 值，Vitest 为你做了这件事情
      pass: received === 'foo',
      message: () => `${received} ${isNot ? 'not' : ''} foo`,
    },
  },
})
```

断言的返回值应该兼容如下接口：

```
interface MatcherResult {
  pass: boolean
  message: () => string
  // 如果你传了这些参数，它们将自动出现在 diff 信息中,
  // 所以即便断言不通过，你也不必自己输出 diff
  actual?: unknown
  expected?: unknown
}
```

如果你创建了一个异步断言，记得在测试代码的结果前使用 `await` 关键字(` await expect('foo').toBeFoo() `)

断言的第一个参数是接收值(即 `expect(received)` 中的 `received`)，其余参数将直接传给断言。

断言方法可以访问上下文 `this` 对象中的这些属性：

- `isNot`

如果断言是在 `not` 方法上调用的(`expect(received).not.toBeFoo()`)，则返回 `true`。

- `promise`

如果断言是在 `resolved/rejected` 中调用的，它的值将包含此断言的名称。否则，它将是一个空字符串。

- `equals`

这是一个工具函数，他可以帮助你比较两个值。如果是相同的则返回 `true`，反之返回 `false`。这个方法几乎在每个断言内部都有使用。默认情况下，它支持非对称的断言。

- `utils`

它包含了一系列工具函数，你可以使用它们来显示信息。

`this` 上下文也包含了当前测试的信息，你可以通过调用 `expect.getState()` 来获取它，其中最有用的属性是：

- `currentTestName`

当前测试的全称(包括 `describe` 块)。

- `testPath`

当前测试的路径。

主要功能

一套配置可以运用在多种环境

与 Vite 的配置、转换器、解析器和插件通用，将会使用应用程序中的相同配置来运行测试。

了解更多信息 [配置 Vitest](#)

监听模式(watch mode)

```
$ vitest
```

当你修改源代码或测试文件时，Vitest 智能搜索模块依赖树并只重新运行相关测试，[just like how HMR works in Vite!](#)

`vitest` 在开发环境下默认 启动时使用 监听模式，在 CI 环境（当 `process.env.CI` 出现时）中以 运行模式(run mode) 启动。你可以使用 `vitest watch` 或 `vitest run` 明确指定所需的模式。

开箱即用的常见 Web 支持

开箱即用的 ES Module / TypeScript / JSX support / PostCSS

多线程

通过 `tinypool` 使用 Worker 线程尽可能多地并发运行（`Piscina` 的轻量级分支），允许多个测试同时运行。Vitest 默认启动多线程，可以通过 CLI 中的 `--no-threads` 禁用。

Vitest 还隔离了每个测试文件的运行环境，因此一个文件中的运行环境改变不会影响其他文件。可以通过将 `--no-isolate` 传递给 CLI 来禁用隔离（以正确性换取运行性能）。

测试可过滤

Vitest 提供了许多缩小测试范围的方法，以便在开发过程中加快速度并集中精力。

了解更多信息 [测试筛选](#)

同时运行多个测试

在连续测试中使用 `.concurrent` 将会并发运行它们。

```
import { describe, it } from "vitest";
// The two tests marked with concurrent will be run in parallel
describe("suite", () => {
  it("serial test", async () => {
    /* ... */
  });
  it.concurrent("concurrent test 1", async () => {
    /* ... */
  });
});
```

```
it.concurrent("concurrent test 2", () => {
  /* ... */
});
```

如果在测试套件中使用 `.concurrent`，则其中的每个测试用例都将并发运行。

```
import { describe, it } from "vitest";
// All tests within this suite will be run in parallel
describe.concurrent("suite", () => {
  it("concurrent test 1", () => {
    /* ... */
  });
  it("concurrent test 2", () => {
    /* ... */
  });
  it.concurrent("concurrent test 3", () => {
    /* ... */
  });
});
```

你还可以将 `.skip`、`.only` 和 `.todo` 用于并发测试套件和测试用例。

了解更多信息 [API 索引](#)

快照

兼容 Jest 快照测试 功能。

```
import { expect, it } from "vitest";
it("renders correctly", () => {
  const result = render();
  expect(result).toMatchSnapshot();
});
```

了解更多信息 [快照](#)

Chai 和 Jest 的 `expect` 语法兼容

内置 Chai 进行断言和与 Jest `expect` 兼容的 APIs

注意，如果你正在使用添加匹配器的第三方库，将 `test.globals` 设置为 `true` 将提供更好的兼容性。

对象模拟(Mocking)

内置 `Tinyspy` 用于在 `vi` 对象上使用 `jest` 兼容的 API 进行对象模拟。

```
import { expect, vi } from "vitest";
const fn = vi.fn();
fn("hello", 1);
expect(vi.isMockFunction(fn)).toBe(true);
expect(fn.mock.calls[0]).toEqual(["hello", 1]);
fn.mockImplementation((arg) => arg);
fn("world", 2);
expect(fn.mock.results[1].value).toBe("world");
```

Vitest 支持 [happy-dom](#) 或 [jsdom](#) 来模拟 DOM 和浏览器 API。Vitest 并不内置它们，所以你可能需要安装：

```
$ npm i -D happy-dom
# or
$ npm i -D jsdom
```

然后，更改 `environment` 配置文件中的选项：

```
// vite.config.ts
import { defineConfig } from "vitest/config";
export default defineConfig({
  test: {
    environment: "happy-dom", // or 'jsdom', 'node'
  },
});
```

了解更多信息 [模拟对象](#)

测试覆盖率

Vitest 通过 `c8` 支持本机代码覆盖率。同时也支持 `istanbul`。

```
{
  "scripts": {
    "test": "vitest",
    "coverage": "vitest run --coverage"
  }
}
```

可以在配置文件中设置 `test.coverage` 选项来配置它：

```
// vite.config.ts
import { defineConfig } from "vitest/config";
export default defineConfig({
  test: {
    coverage: {
      reporter: ["text", "json", "html"],
    },
  },
});
```

了解更多信息 [测试覆盖率](#)

源码内联测试

Vitest 还提供了一种方式，可以运行与你的代码实现放在一起的测试，类似 [Rust's module tests](#)。

这使得测试与实现共享相同的闭包，并且能够在不导出的情况下针对私有状态进行测试。同时，它也使开发更加接近反馈循环。

```
// src/index.ts
// the implementation
export function add(...args: number[]) {
  return args.reduce((a, b) => a + b, 0);
```

```

}
// in-source test suites
if (import.meta.vitest) {
  const { it, expect } = import.meta.vitest;
  it("add", () => {
    expect(add()).toBe(0);
    expect(add(1)).toBe(1);
    expect(add(1, 2, 3)).toBe(6);
  });
}

```

了解更多信息 [源码内联测试](#)

基准测试 experimental

从 Vitest 0.23.0 开始，你可以使用 `bench` 运行基准测试通过 `Tinybench` 函数来比较基准测试结果。

```

import { bench, describe } from "vitest";

describe("sort", () => {
  bench("normal", () => {
    const x = [1, 5, 4, 2, 3];
    x.sort((a, b) => {
      return a - b;
    });
  });

  bench("reverse", () => {
    const x = [1, 5, 4, 2, 3];
    x.reverse().sort((a, b) => {
      return a - b;
    });
  });
});

```

类型测试 experimental

从 Vitest 0.25.0 开始，你可以 [编写测试](#) 来捕获类型回归。Vitest 附带 `expect-type` 包，为你提供类似且易于理解的 API。

```

import { assertType, expectTypeOf } from "vitest";
import { mount } from "./mount.js";

test("my types work properly", () => {
  expectTypeOf(mount).toBeFunction();
  expectTypeOf(mount).parameter(0).toMatchTypeOf<{ name: string }>();

  // @ts-expect-error name is a string
  assertType(mount({ name: 42 }));
});

```

[Go to TOC](#)

测试筛选

用于测试套件(suite)和测试的筛选(filtering)、超时(timeouts)、并发(concurrent)。

CLI

你可以使用 CLI 按名称筛选测试文件：

```
$ vitest basic
```

将只执行包含 `basic` 的测试文件，例如：

```
basic.test.ts
basic-foo.test.ts
```

指定超时阈值

你可以选择将超时阈值（以毫秒为单位）作为第三个参数传递给测试。默认值为 5 秒。

```
import { test } from 'vitest'
test('name', async () => { /* ... */ }, 1000)
```

Hooks 也可以接收超时阈值，默认值为 5 秒。

```
import { beforeAll } from 'vitest'
beforeAll(async () => { /* ... */ }, 1000)
```

跳过测试套件和测试

使用 `.skip` 以避免运行某些测试套件或测试

```
import { assert, describe, it } from 'vitest'

describe.skip('skipped suite', () => {
  it('test', () => {
    // 已跳过此测试套件，无错误
    assert.equal(Math.sqrt(4), 3)
  })
}

describe('suite', () => {
  it.skip('skipped test', () => {
    // 已跳过此测试，无错误
    assert.equal(Math.sqrt(4), 3)
  })
})
```

选择要运行的测试套件和测试

使用 `.only` 仅运行某些测试套件或测试

```
import { assert, describe, it } from 'vitest'

// 仅运行此测试套件（以及标记为 Only 的其他测试套件）
describe.only('suite', () => {
  it('test', () => {
    assert.equal(Math.sqrt(4), 3)
  })
})

describe('another suite', () => {
  it('skipped test', () => {
    // 已跳过测试，因为测试在 Only 模式下运行
    assert.equal(Math.sqrt(4), 3)
  })
}

it.only('test', () => {
  // 仅运行此测试（以及标记为 Only 的其他测试）
  assert.equal(Math.sqrt(4), 2)
})
})
```

未实现的测试套件和测试

使用 `.todo` 留存将要实施的测试套件和测试的待办事项

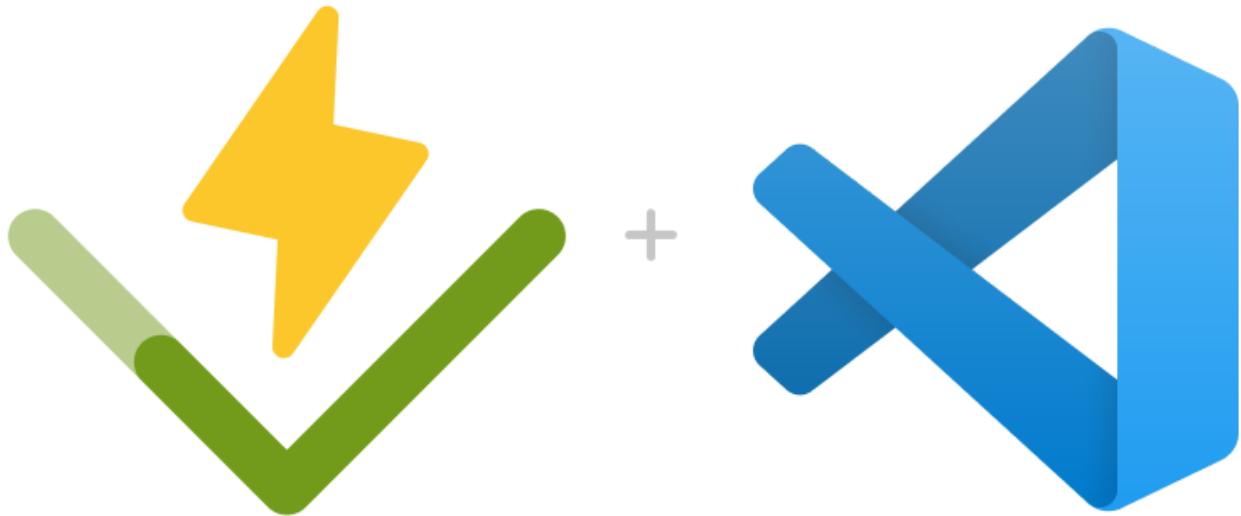
```
import { describe, it } from 'vitest'

// 此测试套件的报告中将显示一个条目
describe.todo('unimplemented suite')

// 此测试的报告中将显示一个条目
describe('suite', () => {
  it.todo('unimplemented test')
})
```

IDE 集成

VS Code 官方



[GitHub | VS Code 插件](#)

The screenshot shows the Vitest extension integrated into an IDE. The left sidebar is the 'TEST EXPLORER' showing a tree of test files and their execution status: run, failed, skipped, todo, or same name. The right panel is the code editor with a file named 'add.test.ts' open. The code contains Jest-style test cases, including describe blocks for 'addition' and 'testing', and various test cases like 'run', 'should failed', 'skipped', 'todo', and 'same name'. The status bar at the bottom includes toolbars for main+, Auto Attach, Go Live, Spell, GraphQL, Prettier, and other development settings.

IntelliJ/WebStorm 社区

由 [@linux-china](#) 来维护。



[GitHub](#) | [JetBrains 插件](#)

```

suite.test.ts
1 import {assert, describe, expect, it} from 'vitest'
2
3 describe('suite name', factory() => {
4   it('name: "foo", fn: () => {
5     assert.equal(actual: 2*2, expected: 4)
6   })
7
8   it('name: "bar2", fn: () => {
9     expect(val: 1 + 1).eq(value: 2)
10  })
11})
12

index.ts
3 return a + b;
4
5 // in-source test suites
6 if (import.meta.vitest) {
7   if (name: 'add', fn: () => {
8     it('name: "add", fn: () => {
9       expect(add(a: 1, b: 2)).toBe(3)
10    })
11  })
12}

basic.test.ts
1 import {assert, expect, test} from 'vitest'
2
3 test('name: "demo", fn: () => {
4   console.log("hello world.")
5 })
6

```

Run: vitest run -t 'demo' tests/basic.test.ts

```

RUN v8.12.2 /Users/linux_china/WebstormProjects/vitest-demo
stdout | tests/basic.test.ts > demo
hello world.
| v tests/basic.test.ts (1 test) 3ms
Test Files 1 passed (1)
Tests 1 passed (1)
Time 1.04s (in thread 3ms, 34705.74%)

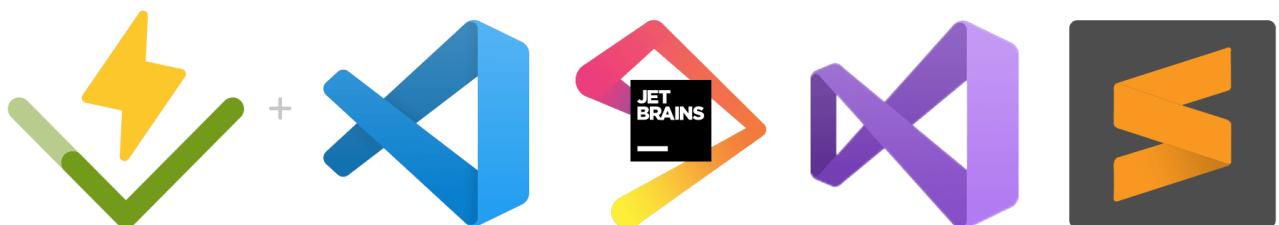
```

Vitest WebStorm Screenshot

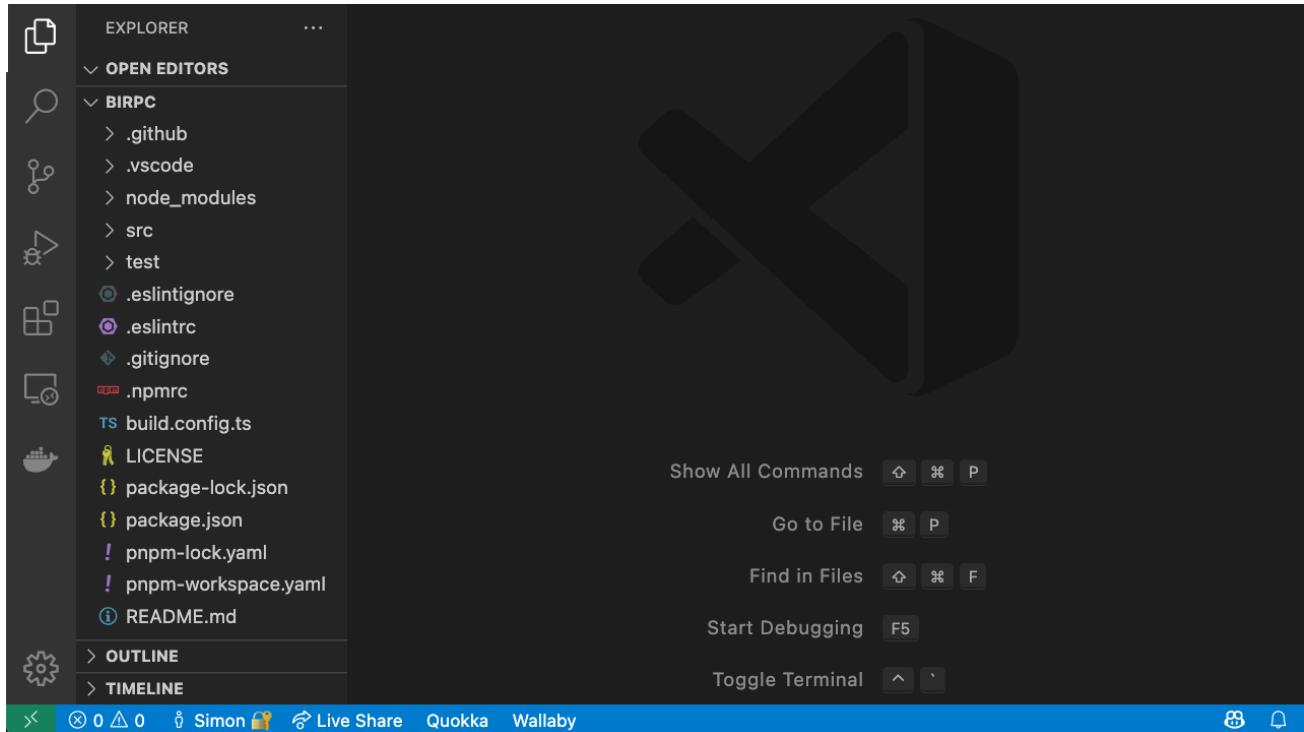
Wallaby.js Paid (free for OSS)

由 [The Wallaby Team](#) 团队创建。

[Wallaby.js](#) 在你键入时立即运行 Vitest 测试，在代码旁边高亮显示 IDE 中的结果。



[VS Code](#) | [JetBrains](#) | [Visual Studio](#) | [Sublime Text](#)



[Go to TOC](#)

源码内联测试

Vitest 还提供了一种方式，可以运行与你的代码实现放在一起的测试，就像是 Rust 语言的模块测试一样。

这允许测试与实现共享相同的闭包，并且能够在不导出的情况下针对私有状态进行测试。同时，它也使开发更加接近反馈循环。

指引

首先，在 `if (import.meta.vitest)` 代码块内写一些测试代码并放在文件的末尾，例如：

```
// src/index.ts

// 函数实现
export function add(...args: number[]) {
  return args.reduce((a, b) => a + b, 0)
}

// 源码内的测试套件
if (import.meta.vitest) {
  const { it, expect } = import.meta.vitest
  it('add', () => {
    expect(add()).toBe(0)
    expect(add(1)).toBe(1)
    expect(add(1, 2, 3)).toBe(6)
  })
}
```

更新 Vitest 配置文件内的 `includeSource` 以获取到 `src/` 下的文件：

```
// vite.config.ts
import { defineConfig } from 'vitest/config'

export default defineConfig({
  test: {
    includeSource: ['src/**/*.{js,ts}'],
  },
})
```

然后你就可以开始执行测试了！

```
$ npx vitest
```

生产环境构建

对于生产环境的构建，你需要设置配置文件内的 `define` 选项，让打包器清除无用的代码。例如，在 Vite 中

```
// vite.config.ts
import { defineConfig } from 'vitest/config'

export default defineConfig({
  + define: {
    + 'import.meta.vitest': 'undefined',
  }
})
```

```
+ },
  test: {
    includeSource: ['src/**/*.{js,ts}']
  },
})
```

其它的打包器

unbuild rollup

TypeScript

要获得对 `import.meta.vitest` 的 TypeScript 支持，添加 `vitest/importMeta` 到 `tsconfig.json`：

```
// tsconfig.json
{
  "compilerOptions": {
    "types": [
      + "vitest/importMeta"
    ]
  }
}
```

完整的示例请参考 `test/import-meta`。

说明

此功能可用于：

- 小范围的功能或utils工具的单元测试
- 原型设计
- 内联断言

对于更复杂的测试，比如组件测试或 E2E 测试，建议**使用单独的测试文件取而代之**。

[Go to TOC](#)

快速起步

总览

Vitest 是一个由 Vite 提供支持的极速单元测试框架。

你可以在 [为什么是 Vitest](#) 中了解有关该项目背后的基本原理的更多信息。

在线试用 Vitest

你可以在 [StackBlitz](#) 上在线尝试 Vitest。它直接在浏览器中运行 Vitest，它几乎与本地设置相同，但不需要在你的计算机上安装任何东西。

将 Vitest 安装到项目

使用 npm

```
npm install -D vitest
```

使用 yarn

```
yarn add -D vitest
```

使用 pnpm

```
pnpm add -D vitest
```

提示 Vitest 需要 Vite >=v3.0.0 和 Node >=v14

配置 Vitest

Vitest 的主要优势之一是它与 Vite 的统一配置。如果存在，`vitest` 将读取你的根目录 `vite.config.ts` 以匹配插件并设置为你的 Vite 应用程序。例如，你的 Vite 有 `resolve.alias` 和 `plugins` 的配置将会在 Vitest 中开箱即用。如果你想在测试期间想要不同的配置，你可以：

- 创建 `vitest.config.ts`，优先级将会最高。
- 将 `--config` 选项传递给 CLI，例如 `vitest --config ./path/to/vitest.config.ts`。
- 在 `defineConfig` 上使用 `process.env.VITEST` 或 `mode` 属性（如果没有被覆盖，将设置为 `test`）有条件地在 `vite.config.ts` 中应用不同的配置。

如果要配置 `vitest` 本身，请在你的 Vite 配置中添加 `test` 属性。你还需要使用 [三斜线命令](#)，同时如果是从 `vite` 本身导入 `defineConfig`，请在配置文件的顶部加上三斜线命令。

```
import { defineConfig } from "vitest/config";
export default defineConfig({
  test: {
```

```
// ...
};

});
```

可以参阅 [配置索引](#) 中的配置选项列表

命令行

在安装了 Vitest 的项目中，你可以在 npm 脚本中使用 `vitest` 脚本，或者直接使用 `npx vitest` 运行它。以下是脚手架 Vitest 项目中的默认 npm 脚本：

```
{
  "scripts": {
    "test": "vitest",
    "coverage": "vitest run --coverage"
  }
}
```

要在不监视文件更改的情况下运行一次测试，请使用 `vitest run`。你还可以指定其他 CLI 选项，例如 `--port` 或 `--https`。有关 CLI 选项的完整列表，可以在你的项目中运行 `npx vitest --help`。

了解更多有关 [命令行界面](#) 的更多信息

IDE 集成

我们还提供了 Visual Studio Code 的官方扩展，以增强你使用 Vitest 的测试体验。

[从 VS Code 插件市场进行安装](#)

了解更多有关 [IDE 插件](#) 的更多信息

示例

示例	源代码	演练场
<code>basic</code>	GitHub	Play Online
<code>graphql</code>	GitHub	Play Online
<code>lit</code>	GitHub	Play Online
<code>mocks</code>	GitHub	Play Online
<code>nextjs</code>	GitHub	Play Online
<code>puppeteer</code>	GitHub	
<code>react-enzyme</code>	GitHub	Play Online
<code>react-mui</code>	GitHub	Play Online
<code>react-storybook</code>	GitHub	Play Online
<code>react-testing-lib-msw</code>	GitHub	Play Online
<code>react-testing-lib</code>	GitHub	Play Online

示例	源代码	演练场
react	GitHub	Play Online
ruby	GitHub	Play Online
solid	GitHub	Play Online
svelte	GitHub	Play Online
vitesse	GitHub	Play Online
vue-jsx	GitHub	Play Online
vue	GitHub	Play Online
vue2	GitHub	Play Online

使用 Vitest 的项目

- uno.css
- unplugin-auto-import
- unplugin-vue-components
- vitesse
- vitesse-lite
- fluent-vue
- vueuse
- milkdown
- gridjs-svelte
- spring-easing
- bytemd
- faker
- million
- Vitamin
- neodrag
- svelte-multiselect
- iconify
- tdesign-vue-next
- cz-git

使用未发布的功能

如果你迫不及待想要体验最新的功能，可以自行克隆 [vitest 仓库](#) 到本地机器上然后自行将其链接（将需要 pnpm）：

```
git clone https://github.com/vitest-dev/vitest.git
cd vitest
pnpm install
cd packages/vitest
pnpm run build
pnpm link --global # 你可以使用你喜爱的任何包管理工具来设置这个步骤
```

然后，回到你的 Vitest 项目并运行 `pnpm link --global vitest`（或者使用你的其他包管理工具来全局链接 Vitest）。

社区

如果你有疑问或者需要帮助，可以到 [Discord](#) 和 [GitHub Discussions](#) 社区来寻求帮助。

[Go to TOC](#)

迁移指南

从 Jest 迁移

Vitest 设计了与 Jest 兼容的 API，方便你从 Jest 的迁移尽可能简单。尽管做出了这些努力，你仍然可能会遇到以下差异：

全局变量作为默认值

Jest 默认启用全局 API。然而 Vitest 没有。你既可以通过 `globals` 配置选项启用全局 API，也可以通过更新你的代码以便使用来自 `vitest` 模块的导入。

如果你决定禁用全局 API，请注意像 `testing-library` 这样的通用库不会自动运行 DOM cleanup。

模拟模块

在 Jest 中模拟一个模块时，工厂参数的返回值是默认导出。在 Vitest 中，工厂参数必须返回一个明确定义了每个导出的对象。例如，下面的 `jest.mock` 必须更新如下：

```
- jest.mock('./some-path', () => 'hello')
+ vi.mock('./some-path', () => ({
+   default: 'hello',
+ }))


```

有关更深入的详细描述，请参阅 [vi.mock api](#)。

自动模拟行为

区别于 Jest，在 `<root>/__mocks__` 中的模拟模块只有在 `vi.mock()` 被调用时才会加载。如果你需要它们像在 Jest 中一样，在每个测试中都被模拟，你可以在 `setupFiles` 中模拟它们。

Jasmine API

Jest 导出各种 `jasmine` 全局 API（例如 `jasmine.any()`）。任何此类实例都需要迁移成 Vitest 的对应 API。

测试环境

如果之前没有设置，Vitest 会像 Jest 一样，把 `NODE_ENV` 设置为 `test`。Vitest 也有一个 `JEST_WORKER_ID` 的对应项，是 `VITEST_WORKER_ID`，所以如果你依赖它，不要忘记重命名它。

回调

从 Vitest v0.10.0 开始，声明测试的回调样式被弃用。你可以重写它们以使用 `async / await` 函数，或者使用 `Promise` 来模仿回调样式。

```
- it('should work', (done) => {
+ it('should work', () => new Promise(done => {
  // ...
  done()
- })
+ }))
```

钩子

`beforeAll` / `beforeEach` 钩子可能在 Vitest 的 `teardown` 函数中返回。因此，如果它们返回的不是 `undefined` 或 `null`，你可能需要重写你的钩子声明：

```
- beforeEach(() => setActivePinia(createTestingPinia()))
+ beforeEach(() => { setActivePinia(createTestingPinia()) })
```

类型

Vitest 没有在 `vi` 命名空间上导出很多类型，它的存在主要是为了与匹配器兼容，因此你可能需要直接从 `vitest` 导入类型，而不是依赖 `vi` 命名空间：

```
- let fn: jest.Mock<string, [string]>
+ import type { Mock } from 'vitest'
+ let fn: Mock<[string], string>
```

此外，Vitest 将 `Args` 类型作为第一个参数，而不是 `Returns`，正如你在 diff 中看到的那样。

定时器

Vitest 不支持 `jest` 的传统计时器。

it.each

Vitest 特意不支持 `it.each` 的模板字面量。你需要将其重写为参数数组或对象数组：

之前：

```
it.each`  
  a | b | expected  
  ${1} | ${3} | ${4}  
  ${2} | ${2} | ${4}  
`('adds $a to $b', ({ a, b, expected }) => {  
  expect(add(a, b)).toEqual(expected)  
})
```

之后：

```
it.each([
  [1, 3, 4],
  [2, 2, 4],
])('adds %d to %d', (a, b, expected) => {
  expect(add(a, b)).toEqual(expected)
})
```

Vue 快照

如果你以前在 `vue-cli preset` 中使用 `Jest`，那么这不是一个 `Jest` 独有的新特性。你可能需要安装 `jest-serializer-vue` 包，然后在 `setupFiles` 中配置：

```
import vueSnapshotSerializer from 'jest-serializer-vue'  
  
// 添加快照系列化  
expect.addSnapshotSerializer(vueSnapshotSerializer)
```

否则你的快照将出现大量的 " 字符。

模拟对象

在编写测试时，你可能会因为时间问题，需要创建内部或外部服务的“假”版本，这通常被称为 **对象模拟** 操作。Vitest 通过 **vi** 提供了一些实用的函数用于解决这个问题。你可以使用 `import { vi } from 'vitest'` 或者 **全局配置** 进行访问它（当 [启用 全局配置](#) 时）。

不要忘记在每次测试运行前后清除或恢复模拟对象，以撤消运行测试时模拟对象状态的更改！有关更多信息，请参阅 `[`mockReset`](..api.html#mockreset)` 文档。

如果你想从头开始，请查看 [API 部分](#) 的 `vi` 部分，或者继续跟着文档深入了解一下这个对象模拟的世界。

日期

有些时候，你可能需要控制日期来确保测试时的一致性。Vitest 使用了 `@sinonjs/fake-timers` 库来操作计时器以及系统日期。可以在 [此处](#) 找到有关特定 API 的更多详细信息。

示例

```
import { afterEach, beforeEach, describe, expect, it, vi } from 'vitest'

const businessHours = [9, 17]

const purchase = () => {
  const currentHour = new Date().getHours()
  const [open, close] = businessHours

  if (currentHour > open && currentHour < close)
    return { message: 'Success' }

  return { message: 'Error' }
}

describe('purchasing flow', () => {
  beforeEach(() => {
    // 告诉 vitest 我们使用模拟时间
    vi.useFakeTimers()
  })

  afterEach(() => {
    // 每次测试运行后恢复日期
    vi.useRealTimers()
  })

  it('allows purchases within business hours', () => {
    // 在工作时间内设置时间
    const date = new Date(2000, 1, 1, 13)
    vi.setSystemTime(date)

    // 访问 Date.now() 将生成上面设置的日期
    expect(purchase()).toEqual({ message: 'Success' })
  })

  it('disallows purchases outside of business hours', () => {
```

```
// 在工作时间之外设置时间
const date = new Date(2000, 1, 1, 19)
vi.setSystemTime(date)

// 访问 Date.now() 将生成上面设置的日期
expect(purchase()).toEqual({ message: 'Error' })
})  
})
```

函数

函数的模拟可以分为两个不同的类别：对象监听(*spying*) & 对象模拟。

有时你可能只需要验证是否调用了特定函数（以及可能传递了哪些参数）。在这种情况下，我们就需要使用一个对象监听，可以直接使用 `vi.spyOn()` ([在此处阅读更多信息](#))。

然而，对象监听只能帮助你 **监听** 函数，他们无法改变这些函数的实现。如果我们需要创建一个函数的假（或模拟）版本，可以使用它 `vi.fn()` ([在此处阅读更多信息](#))。

我们使用 `Tinyspy` 作为模拟函数的基础，同时也有一套自己的封装来使其与 `Jest` 兼容。`vi.fn()` 和 `vi.spyOn()` 共享相同的方法，但是只有 `vi.fn()` 的返回结果是可调用的。

示例

```
import { afterEach, describe, expect, it, vi } from 'vitest'

const getLatest = (index = messages.items.length - 1) => messages.items[index]

const messages = {
  items: [
    { message: 'Simple test message', from: 'Testman' },
    // ...
  ],
  getLatest, // 也可以是一个 `getter` 或 `setter` 如果支持`
}

describe('reading messages', () => {
  afterEach(() => {
    vi.restoreAllMocks()
  })

  it('should get the latest message with a spy', () => {
    const spy = vi.spyOn(messages, 'getLatest')
    expect(spy.getMockName()).toEqual('getLatest')

    expect(messages.getLatest()).toEqual(
      messages.items[messages.items.length - 1],
    )

    expect(spy).toHaveBeenCalledTimes(1)

    spy.mockImplementationOnce(() => 'access-restricted')
    expect(messages.getLatest()).toEqual('access-restricted')

    expect(spy).toHaveBeenCalledTimes(2)
  })

  it('should get with a mock', () => {
```

```

const mock = vi.fn().mockImplementation(getLatest)

expect(mock()).toEqual(messages.items[messages.items.length - 1])
expect(mock).toHaveBeenCalledTimes(1)

mock.mockImplementationOnce(() => 'access-restricted')
expect(mock()).toEqual('access-restricted')

expect(mock).toHaveBeenCalledTimes(2)

expect(mock()).toEqual(messages.items[messages.items.length - 1])
expect(mock).toHaveBeenCalledTimes(3)
})
}

```

了解更多

- [Jest's Mock Functions](#)

全局(Globals)

你可以通过使用 `vi.stubGlobal` 来模拟 `jsdom` 或 `node` 中不存在的全局变量。它将把全局变量的值放入 `globalThis` 对象。

```

import { vi } from 'vitest'

const IntersectionObserverMock = vi.fn(() => ({
  disconnect: vi.fn(),
  observe: vi.fn(),
  takeRecords: vi.fn(),
  unobserve: vi.fn(),
})) 

vi.stubGlobal('IntersectionObserver', IntersectionObserverMock)

// 现在你可以通过 `IntersectionObserver` 或 `window.IntersectionObserver` 访问

```

模块

模拟模块监听在其他代码中调用的第三方库，允许你测试参数、输出甚至重新声明其实现。

参见 `vi.mock()` API 部分 以获得更深入详细 API 描述。

自动模拟算法(Automocking algorithm)

如果你的代码导入了模拟模块，并且没有任何与此模块相关联的 `__mocks__` 文件或 `factory`，Vitest 将通过调用模块并模拟每个导出来的模拟模块本身。

以下原则适用

- 所有的数组将被清空
- 所有的基础类型和集合将保持不变
- 所有的对象都将被深度克隆
- 类的所有实例及其原型都将被深度克隆

示例

```

import { afterEach, beforeEach, describe, expect, it, vi } from 'vitest'
import { Client } from 'pg'
import { failure, success } from './handlers'

// handlers
export function success(data) {}
export function failure(data) {}

// get todos
export const getTodos = async (event, context) => {
  const client = new Client({
    // ...clientOptions
  })

  await client.connect()

  try {
    const result = await client.query('SELECT * FROM todos;')

    client.end()

    return success({
      message: `${result.rowCount} item(s) returned`,
      data: result.rows,
      status: true,
    })
  } catch (e) {
    console.error(e.stack)

    client.end()

    return failure({ message: e, status: false })
  }
}

vi.mock('pg', () => {
  const Client = vi.fn()
  Client.prototype.connect = vi.fn()
  Client.prototype.query = vi.fn()
  Client.prototype.end = vi.fn()

  return { Client }
})

vi.mock('./handlers', () => {
  return {
    success: vi.fn(),
    failure: vi.fn(),
  }
})

describe('get a list of todo items', () => {
  let client

  beforeEach(() => {
    client = new Client()
  })

  afterEach(() => {
    client.end()
  })
})

```

```

    vi.clearAllMocks()
})

it('should return items successfully', async () => {
  client.query.mockResolvedValueOnce({ rows: [], rowCount: 0 })

  await getTodos()

  expect(client.connect).toBeCalledTimes(1)
  expect(client.query).toBeCalledWith('SELECT * FROM todos;')
  expect(client.end).toBeCalledTimes(1)

  expect(success).toBeCalledWith({
    message: '0 item(s) returned',
    data: [],
    status: true,
  })
})

it('should throw an error', async () => {
  const mError = new Error('Unable to retrieve rows')
  client.query.mockRejectedValueOnce(mError)

  await getTodos()

  expect(client.connect).toBeCalledTimes(1)
  expect(client.query).toBeCalledWith('SELECT * FROM todos;')
  expect(client.end).toBeCalledTimes(1)
  expect(failure).toBeCalledWith({ message: mError, status: false })
})
})
}

```

请求

因为 Vitest 运行在 Node 环境中，所以模拟网络请求是一件非常棘手的事情；由于没有办法使用 Web API，因此我们需要一些可以为我们模拟网络行为的包。推荐使用 [Mock Service Worker](#) 来进行这个操作。它可以模拟 REST 和 GraphQL 网络请求，并且与框架无关。

Mock Service Worker (MSW) 通过拦截测试发出的请求进行工作，允许你在不更改任何应用程序代码的情况下使用它。在浏览器中，它使用 [Service Worker API](#)。在 Node 中，对于 Vitest，它使用 [node-request-interceptor](#)。了解有关 MSW 的更多信息，可以去阅读他们的 [introduction](#)。

配置

如下，你可以在你的 [配置文件](#) 中使用。

```

import { afterAll, afterEach, beforeEach } from 'vitest'
import { setupServer } from 'msw/node'
import { graphql, rest } from 'msw'

const posts = [
  {
    userId: 1,
    id: 1,
    title: 'first post title',
    body: 'first post body',
  },
  ...
]

```

```

]
export const restHandlers = [
  rest.get('https://rest-endpoint.example/path/to/posts', (req, res, ctx) => {
    return res(ctx.status(200), ctx.json(posts))
  }),
]

const graphqlHandlers = [
  graphql.query('https://graphql-endpoint.example/api/v1/posts', (req, res, ctx) => {
    return res(ctx.data(posts))
  }),
]

const server = setupServer(...restHandlers, ...graphqlHandlers)

// 在所有测试之前启动服务器
beforeAll(() => server.listen({ onUnhandledRequest: 'error' }))

// 所有测试后关闭服务器
afterAll(() => server.close())

// 每次测试后重置处理程序 `对测试隔离很重要`
afterEach(() => server.resetHandlers())

```

使用 `onUnhandleRequest: 'error'` 配置服务器可以确保每当有没有相应请求处理程序的请求时都会引发错误。

示例

我们有一个使用 MSW 的完整工作示例：[React Testing with MSW](#)。

了解更多

MSW 能做的还有很多。你可以访问 cookie 和查询参数、定义模拟错误响应等等！要查看您可以使用 MSW 做什么，请阅读 [their documentation](#).

计时器

每当我们的测试代码涉及到 `超时` 或者间隔时，并不是让我们的测试程序进行等待或者超时。我们也可以通过模拟对 `setTimeout` 和 `setInterval` 的调用来使用“假”计时器来加速测试。

有关更深入的详细 API 描述，参阅 [vi.usefaketimers api 部分](#)。

示例

```

import { afterEach, beforeEach, describe, expect, it, vi } from 'vitest'

const executeAfterTwoHours = (func) => {
  setTimeout(func, 1000 * 60 * 60 * 2) // 2 hours
}

const executeEveryMinute = (func) => {
  setInterval(func, 1000 * 60) // 1 minute

```

```

}

const mock = vi.fn(() => console.log('executed'))

describe('delayed execution', () => {
  beforeEach(() => {
    vi.useFakeTimers()
  })
  afterEach(() => {
    vi.restoreAllMocks()
  })
  it('should execute the function', () => {
    executeAfterTwoHours(mock)
    vi.runAllTimers()
    expect(mock).toHaveBeenCalledTimes(1)
  })
  it('should not execute the function', () => {
    executeAfterTwoHours(mock)
    // advancing by 2ms won't trigger the func
    vi.advanceTimersByTime(2)
    expect(mock).not.toHaveBeenCalled()
  })
  it('should execute every minute', () => {
    executeEveryMinute(mock)
    vi.advanceTimersToNextTimer()
    expect(mock).toHaveBeenCalledTimes(1)
    vi.advanceTimersToNextTimer()
    expect(mock).toHaveBeenCalledTimes(2)
  })
})
})

```

备忘单

info 下列示例中的 `vi` 是直接从 `vitest` 导入的。如果在你的 [config](/config/) 中将 `globals` 设置为 `true`，则可以全局使用它。

我想...

- 监听一个 `method`

```

const instance = new SomeClass()
vi.spyOn(instance, 'method')

```

- 监听模块导出 function

```

// some-path.ts
export const getter = 'variable'

```

```

// some-path.test.ts
import * as exports from 'some-path'
vi.spyOn(exports, 'getter', 'get').mockReturnValue('mocked')

```

- 监听模块导出 setter/getter

```

import * as exports from 'some-path'
vi.spyOn(exports, 'getter', 'get')
vi.spyOn(exports, 'setter', 'set')

```

- 模拟模块导出 function

`vi.mock` 的示例：

```
// some-path.ts
export function method() {}
```

```
import { method } from 'some-path'
vi.mock('some-path', () => ({
  method: vi.fn()
}))
```

`vi.spyOn` 的示例：

```
import * as exports from 'some-path'
vi.spyOn(exports, 'method').mockImplementation(() => {})
```

- 模拟模块导出 class implementation

`vi.mock` and `prototype` 的示例：

```
// some-path.ts
export class SomeClass {}
```

```
import { SomeClass } from 'some-path'
vi.mock('some-path', () => {
  const SomeClass = vi.fn()
  SomeClass.prototype.someMethod = vi.fn()
  return { SomeClass }
})
// SomeClass.mock.instances will have SomeClass
```

`vi.mock` and `return value` 的示例：

```
import { SomeClass } from 'some-path'
vi.mock('some-path', () => {
  const SomeClass = vi.fn(() => ({
    someMethod: vi.fn()
  }))
  return { SomeClass }
})
// SomeClass.mock.returns will have returned object
```

`vi.spyOn` 的示例：

```
import * as exports from 'some-path'
vi.spyOn(exports, 'SomeClass').mockImplementation(() => {
  // whatever suites you from first two examples
})
```

- 监听一个函数是否返回了一个对象

使用 `cache` 的示例：

```
// some-path.ts
export function useObject() {
  return { method: () => true }
}
```

```
// useObject.js
import { useObject } from 'some-path'
const obj = useObject()
obj.method()
```

```
// useObject.test.js
import { useObject } from 'some-path'
vi.mock('some-path', () => {
  let _cache
  const useObject = () => {
    if (!_cache) {
      _cache = {
        method: vi.fn(),
      }
    }
    // now everytime useObject() is called it will
    // return the same object reference
    return _cache
  }
  return { useObject }
})

const obj = useObject()
// obj.method was called inside some-path
expect(obj.method).toHaveBeenCalled()
```

- 模拟部分 module

```
import { mocked, original } from 'some-path'
vi.mock('some-path', async () => {
  const mod = await vi.importActual<typeof import('some-path')>('some-path')
  return {
    ...mod,
    mocked: vi.fn()
  }
})
original() // has original behaviour
mocked() // is a spy function
```

- 模拟当前日期

```
const mockDate = new Date(2022, 0, 1)
vi.setSystemTime(mockDate)
const now = new Date()
expect(now.valueOf()).toBe(mockDate.valueOf())
```

- 模拟全局变量

```
vi.stubGlobal('__VERSION__', '1.0.0')
expect(__VERSION__).toBe('1.0.0')
```

测试快照

当你希望确保函数的输出不会意外更改时，快照测试是一个非常有用的工具。

使用快照时，Vitest 将获取给定值的快照，将其比较时将参考存储在测试旁边的快照文件。如果两个快照不匹配，则测试将失败：要么更改是意外的，要么参考快照需要更新到测试结果的新版本。

使用快照

要将一个值快照，你可以使用 `expect()` 的 `toMatchSnapshot()` API:

```
import { expect, it } from 'vitest'

it('toUpperCase', () => {
  const result = toUpperCase('foobar')
  expect(result).toMatchSnapshot()
})
```

此测试在第一次运行时，Vitest 会创建一个快照文件，如下所示：

```
// Vitest Snapshot v1
exports['toUpperCase 1'] = '"FOOBAR"
```

快照文件应该与代码更改一起提交，并作为代码审查过程的一部分进行审查。在随后的测试运行中，Vitest 会将执行的输出与之前的快照进行比较。如果他们匹配，测试就会通过。如果它们不匹配，要么测试运行时在你的代码中发现了应该修复的错误，要么实现已经更改，需要更新快照。

内联快照

如同前文，你可以使用 `toMatchInlineSnapshot()` 将内联快照存储在测试文件中。

```
import { expect, it } from 'vitest'

it('toUpperCase', () => {
  const result = toUpperCase('foobar')
  expect(result).toMatchSnapshot()
})
```

Vitest 将修改测试文件目录，以将快照更新为字符串，而不是创建快照文件：

```
import { expect, it } from 'vitest'

it('toUpperCase', () => {
  const result = toUpperCase('foobar')
  expect(result).toMatchSnapshot('"FOOBAR"')
})
```

这允许你直接查看期望输出，而无需跨不同的文件跳转。

更新快照

当接收到的值与快照不匹配时，测试将失败，并显示它们之间的差异。当需要更改快照时，你可能希望从当前状态更新快照。

在监听(watch)模式下，你可以在终端中键入 `u` 键直接更新失败的快照。

或者，你可以在 CLI 中使用 `--update` 或 `-u` 标记使 Vitest 进入快照更新模式。

```
vitest -u
```

图像快照

快照图像也可以使用 `jest-image-snapshot`。

```
npm i -D jest-image-snapshot
```

```
test('image snapshot', () => {
  expect(readFileSync('./test/stubs/input-image.png'))
    .toMatchImageSnapshot()
})
```

你可以在 `examples/image-snapshot` 中学习更多案例。

自定义序列化程序

你可以添加自己的逻辑来修改快照的序列化方式。像 Jest 一样，Vitest 为内置的 JavaScript 类型、HTML 元素、`ImmutableJS` 和 React 元素提供了默认的序列化程序。

序列化模块示例：

```
expect.addSnapshotSerializer({
  serialize(val, config, indentation, depth, refs, printer) {
    // `printer` is a function that serializes a value using existing plugins.
    return `Pretty foo: ${printer(val.foo)}`
  },
  test(val) {
    return val && Object.prototype.hasOwnProperty.call(val, 'foo')
  },
})
```

如下所示的测试添加后：

```
test('foo snapshot test', () => {
  const bar = {
    foo: {
      x: 1,
      y: 2,
    },
  }
  expect(bar).toMatchSnapshot()
})
```

你将获得以下快照：

```
Pretty foo: Object {
  "x": 1,
  "y": 2,
}
```

我们使用的是 Jest 的 `pretty-format` 来序列化快照。你可以在这里阅读更多相关内容：[pretty-format](#).

与 Jest 的区别

Vitest 提供了与 Jest 几乎兼容的快照功能，除少数例外：

1. 快照文件中的注释标头不同

```
- // Jest Snapshot v1
+ // Vitest Snapshot v1
```

这实际上不会影响功能，但在从 Jest 迁移时可能会影响提交差异。

2. `printBasicPrototype` 默认为 `false`

Jest 和 Vitest 的快照都是由 `pretty-format` 支持的。在 Vitest 中，我们将 `printBasicPrototype` 的默认值设置为 `false` 以提供更清晰的快照输出，Jest 实际上将其默认设置为 `true`。

```
import { expect, test } from 'vitest'

test('snapshot', () => {
  const bar = [
    {
      foo: 'bar',
    },
  ]

  // in Jest
  expect(bar).toMatchSnapshot(`

    Array [
      Object {
        "foo": "bar",
      },
    ]
  `)

  // in Vitest
  expect(bar).toMatchSnapshot(`

    [
      {
        "foo": "bar",
      },
    ]
  `)
})
```

我们相信这种预设有更好的可读性和开发体验。如果你仍然喜欢 Jest 的行为，可以通过以下方式更改配置：

```
// vitest.config.js
export default defineConfig({
  test: {
    snapshotFormat: {
      printBasicPrototype: true
    }
  }
})
```

测试上下文

受 [Playwright Fixtures](#) 的启发, Vitest 的测试上下文允许你定义可在测试中使用的工具(utils)、状态(states)和固定装置/fixtures)。

用法

第一个参数或每个测试回调是一个测试上下文。

```
import { it } from 'vitest'

it('should work', (ctx) => {
  // prints name of the test
  console.log(ctx.meta.name)
})
```

内置测试上下文

`context.meta`

包含关于测试的元数据的只读对象。

`context.expect`

绑定到当前测试的 `expect` API。

扩展测试上下文

每个测试的上下文都不同。 你可以在 `beforeEach` 和 `afterEach` hooks 中访问和扩展它们。

```
import { beforeEach, it } from 'vitest'

beforeEach(async (context) => {
  // extend context
  context.foo = 'bar'
})

it('should work', ({ foo }) => {
  console.log(foo) // 'bar'
})
```

TypeScript

你可以通过添加聚合(aggregate)类型 `TestContext`, 为你的自定义上下文属性提供类型支持。

```
declare module 'vitest' {
  export interface TestContext {
    foo?: string
  }
}
```

如果你只想为特定的 `beforeEach`、`afterEach`、`it` 或 `test` hooks 提供属性类型，则可以将类型作为泛型 (generic) 传递。

```
interface LocalTestContext {
  foo: string
}

beforeEach<LocalTestContext>(async (context) => {
  // typeof context is 'TestContext & LocalTestContext'
  context.foo = 'bar'
})

it<LocalTestContext>('should work', ({ foo }) => {
  // typeof foo is 'string'
  console.log(foo) // 'bar'
})`<span style='float: footnote;'><a href='../index.html#toc'>Go to TOC</a></span>
```

Testing Types

Vitest allows you to write tests for your types, using `expectTypeOf` or `assertType` syntaxes. By default all tests inside `*.test-d.ts` files are considered type tests, but you can change it with `typecheck.include` config option.

Under the hood Vitest calls `tsc` or `vue-tsc`, depending on your config, and parses results. Vitest will also print out type errors in your source code, if it finds any. You can disable it with `typecheck.ignoreSourceErrors` config option.

Keep in mind that Vitest doesn't run or compile these files, they are only statically analyzed by the compiler, and because of that you cannot use any dynamic statements. Meaning, you cannot use dynamic test names, and `test.each`, `test.runIf`, `test.skipIf`, `test.each`, `test.concurrent` APIs. But you can use other APIs, like `test`, `describe`, `.only`, `.skip` and `.todo`.

Using CLI flags, like `--allowOnly` and `-t` are also supported for type checking.

```
import { assertType, expectTypeOf } from 'vitest'
import { mount } from './mount.js'

test('my types work properly', () => {
  expectTypeOf(mount).toBeFunction()
  expectTypeOf(mount).parameter(0).toMatchTypeOf<{ name: string }>()

  // @ts-expect-error name is a string
  assertType(mount({ name: 42 }))
})
```

Any type error triggered inside a test file will be treated as a test error, so you can use any type trick you want to test types of your project.

You can see a list of possible matchers in [API section](#).

Reading Errors

If you are using `expectTypeOf` API, you might notice hard to read errors or unexpected:

```
expectTypeOf(1).toEqualTypeOf<string>()
//                                     ^^^^^^^^^^
// index-c3943160.d.ts(90, 20): Arguments for the rest parameter 'MISMATCH' were
not provided.
```

This is due to how `expect-type` handles type errors.

Unfortunately, TypeScript doesn't provide type metadata without patching, so we cannot provide useful error messages at this point, but there are [works in TypeScript project](#) to fix this. If you want better messages, please, ask TypeScript team to have a look at mentioned PR.

If you find it hard working with `expectTypeof` API and figuring out errors, you can always use more simple `assertType` API:

```
const answer = 42

assertType<number>(answer)
// @ts-expect-error answer is not a string
assertType<string>(answer)
```

When using `@ts-expect-error` syntax, you might want to make sure that you didn't make a typo.

You can do that by including your type files in [`test.include`](../config.html#include) config option, so Vitest will also actually *run* these tests and fail with `ReferenceError`.

This will pass, because it expects an error, but the word "answer" has a typo, so it's a false positive error:

```
// @ts-expect-error answer is not a string
assertType<string>(answr) //
```

Run typechecking

Add this command to your `scripts` section in `package.json`:

```
{
  "scripts": {
    "typecheck": "vitest typecheck"
  }
}
```

Now you can run typecheck:

```
# npm
npm run typecheck

# yarn
yarn typecheck

# pnpm
pnpm run typecheck
```

Vitest uses `tsc --noEmit` or `vue-tsc --noEmit`, depending on your configuration, so you can remove these scripts from your pipeline.

[Go to TOC](#)

Vitest UI

Vitest 由 Vite 提供能力，在运行测试时有一个开发服务器。这允许 Vitest 提供一个漂亮的 UI 界面来查看并与测试交互。Vitest 的 UI 界面是可选的，你可以通过以下安装：

```
npm i -D @vitest/ui
```

接下来，你可以通过传入 `--ui` 参数来启动测试的 UI 界面：

```
vitest --ui
```

最后，你可以访问 Vitest UI 界面，通过 `http://localhost:51204/_vitest_/_`

The screenshot shows the Vitest interface with two panes. The left pane displays a tree view of test files and their execution status. The right pane shows the source code of the selected file, `basic.test.ts`. The code includes assertions for `Math.sqrt` and `JSON.stringify`.

```

1 import { assert, expect, it, suite, test } from 'vitest'
2 import { two } from '../src/submodule'
3 import { timeout } from '../src/timeout'
4
5 test('Math.sqrt()', async () => {
6   assert.equal(Math.sqrt(5), two)
7   ^ AssertionError: expected 2.23606797749979 to equal 2
8   assert.equal(Math.sqrt(2), Math.SQRT2)
9   expect(Math.sqrt(144)).toStrictEqual(12)
10 // await new Promise(resolve => setTimeout(resolve, 3000))
11
12 test('JSON', () => {
13   const input = {
14     foo: 'hello',
15     bar: 'world',
16   }
17
18   const output = JSON.stringify(input)
19
20   expect(input).toEqual({
21     foo: 'hello',
22     bar: 'world',
23   })
24   expect(output).toEqual('{"foo":"hello","bar":"world"}')
25   assert.deepEqual(JSON.parse(output), input, 'matches original')
26 })
27
28 test('mode and NODE_ENV is test by default', () => {
29   expect(process.env.NODE_ENV).toBe('test')
30   expect(import.meta.env.MODE).toBe('test')
31 })

```

This screenshot is identical to the one above, showing the same test results and code editor for `basic.test.ts`. The assertion for `Math.sqrt` is highlighted in red.

```

1 import { assert, expect, it, suite, test } from 'vitest'
2 import { two } from '../src/submodule'
3 import { timeout } from '../src/timeout'
4
5 test('Math.sqrt()', async () => {
6   assert.equal(Math.sqrt(5), two)
7   ^ AssertionError: expected 2.23606797749979 to equal 2
8   assert.equal(Math.sqrt(2), Math.SQRT2)
9   expect(Math.sqrt(144)).toStrictEqual(12)
10 // await new Promise(resolve => setTimeout(resolve, 3000))
11
12 test('JSON', () => {
13   const input = {
14     foo: 'hello',
15     bar: 'world',
16   }
17
18   const output = JSON.stringify(input)
19
20   expect(input).toEqual({
21     foo: 'hello',
22     bar: 'world',
23   })
24   expect(output).toEqual('{"foo":"hello","bar":"world"}')
25   assert.deepEqual(JSON.parse(output), input, 'matches original')
26 })
27
28 test('mode and NODE_ENV is test by default', () => {
29   expect(process.env.NODE_ENV).toBe('test')
30   expect(import.meta.env.MODE).toBe('test')
31 })

```

[Go to TOC](#)

使用插件

Vitest 可以使用插件进行扩展...

为什么是 Vitest

提示 该文档假设你是熟悉 Vite 的。开始阅读之前建议先浏览 [为什么选 Vite] (<https://cn.vitejs.dev/guide/why.html>) 和 [下一代前端工具 ViteJS] (<https://www.bilibili.com/video/BV1kh411Q7WN>)，在视频中 [尤雨溪] (<https://github.com/yyx990803>) 做了一个示范来解释 Vite 的主要概念。

Vite 原生测试运行器的必要性

Vite 天然支持常见的 Web 模式，同时支持 glob 导入和 SSR 等功能，而且它拥有许多插件和集成框架，从而慢慢形成一个活跃的生态社区。它的开发和构建模式是其成功的关键。对于文档构建框架，Vite 提供了一些基于 SSG 的替代方案。但是 Vite 的单元测试形式还不是十分清晰，而对于目前一些现有方案，比如 Jest 而言，它们会在不同的上下文环境中被创建。并且 Jest 和 Vite 之间有很多重复的部分，让用户不得不创建两个不同的配置文件。

如果使用 Vite 开发服务器在测试期间来转换你的文件，此时我们可以只创建一个简单运行程序，它无需处理复杂的源代码转换，而只专注于在测试期间能够提供最佳的开发体验。并且该测试运行程序可以使用你的 App 的配置文件（通过 `vite.config.js`），在开发、构建和测试期间共享一个通用的转换容器。同时可以通过相同的插件 API 进行扩展，让你和你的工具与 Vite 形成完美的集成。一个从一开始就考虑到使用 Vite 构建的工具，利用了它对开发体验的提升，比如它的即时热模块重载 (HMR)。这就是 Vitest，一个由 Vite 提供支持的极速单元测试框架。

由于 Jest 的大规模使用，Vitest 提供了与之兼容的API，允许大家在大多数项目中将其作为备选使用。同时还包括了单元测试时最常见的功能（模拟，快照以及覆盖率）。Vitest 非常注重性能，尽可能多地使用 Worker 线程进行并发运行。并且在一些端口的测试运行速度提高了一个数量级别。监听模式默认启用，与 Vite 推动开发者优先体验的理念保持一致。即使在开发体验上进行了改进，Vitest 通过仔细挑选其依赖项（或直接内联所需的部分）来保持轻量级。

Vitest 旨在将自己定位为 Vite 项目的首选测试框架，即使对于不使用 Vite 的项目也是一个可靠的替代方案。

继续阅读 [快速起步](#)

Vitest 与 X 有何不同？

你可以查看 [比较](#) 部分，了解有关 Vitest 与其他类似工具有何不同的更多详细信息。

[Go to TOC](#)

Colophon

This book is created by using the following sources:

- Vitest - 简体中文
- GitHub source: vitest-dev/docs-cn
- Created: 2022-11-27
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>