

32 | 单元测试：如何打造Vue.js和Node.js全栈项目的单元测试？

2023-02-24 杨文坚 来自北京

《Vue 3 企业级项目实战课》

[课程介绍 >](#)



讲述：杨文坚

时长 14:06 大小 12.88M



你好，我是杨文坚。

从这一课开始，我们进入平台优化、扩展设计开发阶段的增强篇，主要会分为五个部分：单元测试、页面功能、服务端功能、多进程部署、日志收集与问题排错。

今天我们学习全栈项目如何进行单元测试。

前端领域的单元测试，之前（[@14 讲](#)）我们已经学习了，使用 **Vue.js** 官方维护的测试工具 **Vitest**，结合项目本身的 **Vite** 配置，自动编译和测试 **Vue.js** 代码。在前端单元测试过程中，我们主要用 **Node.js** 运行环境，模拟浏览器的 **API**，直接对开发中的前端代码进行单元测试，保障前端代码的质量。

现在回到课程的全栈项目，我们还需要对 **Node.js** 服务端代码进行单元测试。

因为运营搭建平台的服务端代码，本身就是运行在 **Node.js** 环境的，我们可以把单元测试直接放在 **Node.js** 环境中进行。不过，这里你可能会有疑问，进行前端代码单元测试的时候，理论上也调用服务端提供的 **HTTP** 接口直接使用，并且进行测试，等于间接测试了服务端，为什么还要单独对服务端代码做单元测试呢？

为什么需要对服务端做单元测试

首先要明确一个观点，无论全栈项目用什么技术开发服务端，服务端单元测试都是必须要做的操作。

因为服务端不像前端那么“方便”。前端代码可以“所见即所得”，直接让代码运行在浏览器里验证前端功能效果，但是，服务端，**很多功能都是“不可见”的内容**，例如 **HTTP** 接口、**HTTP** 请求的状态情况、**TCP** 通信接口，甚至是业务代码的各种 **API**、数据库调用操作等等。

而且，服务端功能在开发过程中，**很多功能不方便直接验证**，通常需要借助一些工具来辅助验证，例如用 **Postman** 来做完整的 **HTTP** 请求服务功能验证。

所以，做服务端开发的程序员，都需要单元测试来做功能的验证和保障。如果你曾经做过 **Java** 服务端开发，应该被要求过写 **Java** 代码的单元测试。

前端和服务端在单元测试中有什么差异

而且目前，我们只依靠前端单元测试，直接使用服务端的 **HTTP** 接口，间接对服务端做单元测试，是不够全面的。

单纯的 **HTTP** 接口调用来渲染前端代码进行测试，覆盖不了服务端的所有功能逻辑，比如服务背后关联的数据库操作测试、业务逻辑分支测试，以及 **HTTP** 请求状态的内容的测试。从本质上来讲，这是前端和服务端单元测试的差异，也带来了各自的局限性。

差异，主要体现在测试内容上。

在前端 **Vue.js** 单元测试中，主要内容有三点。

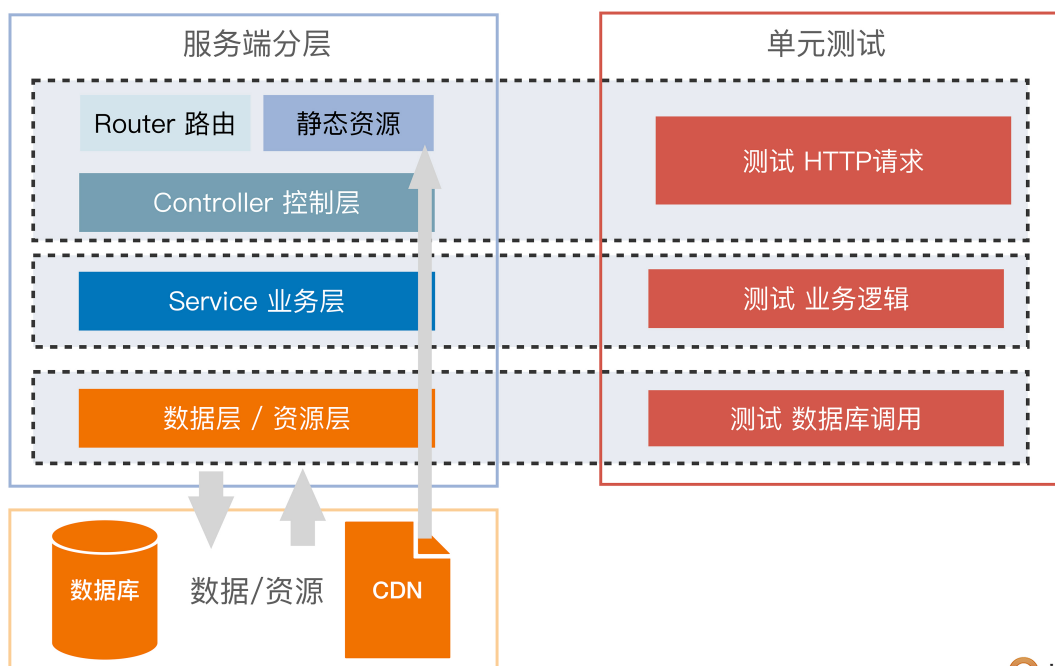
- **DOM** 结构
- **DOM** 事件

- 使用模拟或真实的 HTTP 请求

服务端没有浏览器的 DOM 内容，更多是数据操作和业务流程逻辑。所以，服务端的单元测试侧重内容主要是这三点。

- 数据库调用
- 业务逻辑代码
- HTTP 内容和状态

因为全栈项目服务端的分层结构，从下往上是分别是：数据层（Model）、业务层（Service）、控制层（Controller）、路由层（Router）。我们可以看出，服务端单元测试的三个侧重点，直接对应服务端代码分层结构的一个或者两个层级。



我们根据分层，从下往上看每个测试点的具体操作内容。

第一点，数据库调用的测试，就是验证服务端的数据层代码，具体操作内容就是测试和验证在服务端中，操作数据库的相关代码。这个过程需要特别注意，**数据库不能直接用生产环境的数据库，一般用开发测试环境的数据库进行单元测试。**

第二点，业务逻辑代码，就是验证服务端的业务层代码。具体操作内容就是基于测试环境的业务层的代码，调用数据层内容，进行业务功能逻辑验证，来判断业务逻辑代码的输入和输出，是否符合正确预期。

第三点，HTTP 内容和状态，就是验证服务端的控制层和路由层代码，从 HTTP 网络层面进行功能测试和验证。在这个测试过程中，真实地用测试代码发起 HTTP 请求，去访问和请求服务端提供的 HTTP 服务，最后验证 HTTP 服务响应结果是否符合正确预期。

了解了服务端测试的主要内容，下一步就是进行测试了。

不过，既然前端和服务端的测试内容有差异，那么之前学习的前端测试的工具，是不是就不能用了呢？“工欲善其事，必先利其器”，我们怎么选择工具，进行 Node.js 服务的单元测试呢？

如何选择工具进行 Node.js 服务的单元测试

在 Vue.js 前端单元测试的学习过程中，我们掌握了三个 Node.js 环境下前端单元测试工具。

- Mocha，老牌 Node.js 环境的单元测试工具，服务端功能测试的生态比较齐全。
- Jest，React.js 官方团队维护的前端单元测试工具，基于 Node.js 环境来模拟前端浏览器环境，能测试服务端功能代码。
- Vitest，Vue.js 官方团队维护的前端单元测试工具，同样基于 Node.js 环境来模拟前端浏览器环境，也支持测试服务端功能代码。

综合看，Mocha 比较适用 Node.js 服务端项目，Jest 比较适用 React.js 项目，Vitest 比较适用 Vue.js 项目。那么，我们选择 Mocha 来做 Node.js 服务端测试是不是最优解呢？

这里我们要讨论一个技术观点，单元测试是为了保障代码质量，尽量减少代码迭代过程不必要的问题，所以没有绝对的技术工具要求，只有质量和效率的要求。而且，Node.js 全栈项目的单元测试没有绝对的工具限制，无论是全栈项目的前端，还是服务端代码的单元测试，核心是要验证功能模块输入和输出是否符合预期。

所以，测试工具，你完全可以按照个人开发习惯，或者团队开发规范，或者学习成本考虑等等因素来做选择。

为了跟之前课程的 **Vue.js** 的单元测试保持一致，用同一个单元测试工具。对于课程的全栈项目的服务端单元测试，我们也选择用 **Vitest** 来实现。

这里，你可能会有顾虑，使用了比较新的 **Vitest** 作为测试工具，是不是等于放弃 **Mocha** 和 **Jest** 的技术工具生态？会不会增加服务端测试的工作量？

不用顾虑，**Vitest** 可以复用 **Jest** 等生态工具，而且 **Vitest** 也对 **Jest** 的一些内容做了兼容工作。况且，我们不是讨论和讲解如何使用工具，而是要讲解面对需求场景，如何制定方案来实现功能和解决问题，不能陷入依赖工具的“术”的误区，要尽量提升到解决方案的“道”的视角。

好，现在我们已经确定继续沿用 **Vitest** 这个测试工具，那么如何来设计和实现 **Node.js** 服务的单元测试呢？

如何设计和实现 Node.js 服务的单元测试

按照我们前面的归纳，服务端测试场景可以分成数据库、业务代码 **API** 和 **HTTP** 请求，进行三个层面的单元测试。

首先，**数据库层面的单元测试**，主要是用来验证数据操作逻辑是否符合预期。在这个测试过程中，要特别注意，必须用测试环境的数据库来测试，我们可以复用之前提到的发布流程中，测试节点的环境。

实现具体就是引用对应的数据操作代码，在单元测试代码中进行。

 复制代码

```
1 // packages/work-server/__tests__/database.test.ts
2 import { describe, test, expect } from 'vitest';
3 import md5 from 'md5';
4 import {
5   findUserByUsernameAndPassword,
6   checkUserIsUsernameExist
7 } from '../src/model/user';
8
9 describe('work-server: database', () => {
10
11   // 验证查询用户数据方法
12   test('model/user.ts findUserByUsernameAndPassword', async () => {
13     const result = await findUserByUsernameAndPassword({
14       username: 'admin001',
15       password: md5('88888888')
```

```

16     });
17     expect(result).toStrictEqual({
18       id: 1,
19       uuid: '00000000-aaaa-bbbb-cccc-ddddeeee0001',
20       username: 'admin001',
21       password: '1f22f6ce6e58a7326c5b5dd197973105',
22       status: 1,
23       info: '{}',
24       extend: null,
25       createTime: '2023-01-18T08:33:18.000Z',
26       modifyTime: '2023-01-18T08:33:18.000Z'
27     });
28   });
29
30   })

```

在例子中，我引用了查询用户的数据库操作代码，进行单独的功能模块测试。在服务端单元测试过程中，要保持数据库服务的开启状态，才能进行调用测试。

第二个层面，**业务代码的 API 单元测试**，就是直接测试业务逻辑的运行结果是否符合预期。

这个层面的单元测试，大多数跟数据层面的测试会有重叠。这是因为业务代码大部分都会涉及数据库内容的操作，所以等于间接测试了数据库层面的内容。

 复制代码

```

1  // packages/work-server/__tests__/service.test.ts
2  import { describe, test, expect } from 'vitest';
3  import md5 from 'md5';
4  import { queryAccount } from '../src/service/user';
5
6  describe('work-server: database', () => {
7    test('service/user.ts queryAccount', async () => {
8      const result = await queryAccount({
9        username: 'admin001',
10       password: md5('88888888')
11     });
12     expect(result).toStrictEqual({
13       data: {
14         allow: true,
15         username: 'admin001',
16         uuid: '00000000-aaaa-bbbb-cccc-ddddeeee0001'
17       },
18       success: true,
19       message: '登录成功'
20     });
21   });
22 });

```

在这段测试代码中，我引用了用户登录过程中查询用户的业务代码 **API**，进行测试。用户登录逻辑需要先根据账号和密码查询数据库，判断用户数据是否已存在，如果存在，就可以得到允许登录的业务数据。

这个用户登录业务代码逻辑中，使用了上一个单元测试中验证的数据库操作代码，也就间接验证了数据库层面的代码。

最后一个层面，**HTTP 请求测试**，主要是基于 **Node.js** 项目提供的 **HTTP 服务**，发起 **HTTP 请求进行功能测试**，验证路由和控制层的功能是否符合预期。

具体可以按照功能视角的颗粒度进行测试。我们看个例子，比如，用户登录功能维度的 **HTTP 请求内容**，具体测试步骤。

第一步，测试用户登录的 **HTTP 页面**，进行登录页面的 **HTTP 请求测试**，验证一下用户登录页面是否能正常访问。参考相关测试用例代码。

 复制代码

```
1 // packages/work-server/__tests__/http.test.ts
2 // ...
3 describe('work-server', () => {
4   //...
5   test('page /page/sign-in', async () => {
6     const url = `http://${workServerHost}:${workServerPort}/page/sign-in`;
7     const res = await nodeFetch(url);
8     const html = await res.text();
9     const expectHtml = `<html>
10    <head>
11      <meta charset="utf-8" />
12      <script type="importmap">
13        {
14          "imports": {
15            "vue": "/public/cdn/pkg/vue/3.2.45/dist/vue.runtime.esm-browser.js"
16          }
17        }
18      </script>
19      <link href="/public/dist/page/sign-in.css" rel="stylesheet" />
20      <script src="/public/dist/lib/vue.js"></script>
21      <script src="/public/dist/lib/vue-router.js"></script>
22    </head>
23    <body>
24      <div id="app"></div>
```



```

25     </body>
26     <script src="/public/dist/page/sign-in.js"></script>
27 </html>`;
28     expect(html).toStrictEqual(expectHtml);
29   });
30
31   //...
32 }):

```

第二步，测试登录的 HTTP 接口，用户登录操作是用异步的 HTTP 请求进行的，所以需要测试用户登录相关的 HTTP API。参考相关测试用例代码。

 复制代码

```

1 // packages/work-server/__tests__/http.test.ts
2 // ...
3 describe('work-server', () => {
4   //...
5   test('login action /api/post/account/sign-in', async () => {
6     const url = `http://${workServerHost}:${workServerPort}/api/post/account/si
7     const res = await nodeFetch(url, {
8       body: JSON.stringify({
9         username: 'admin001',
10        password: md5('123456')
11      }),
12      headers: {
13        'content-type': 'application/json'
14      },
15      method: 'POST'
16    });
17    const json = await res.json();
18    expect(json).toStrictEqual({
19      data: { allow: false },
20      success: true,
21      message: '登录成功'
22    });
23  });
24
25   //...
26 });

```

第三步，测试登录成功的 HTTP 状态。在用户登录成功后，会在服务端设置 HTTP Cookie 等状态数据，所以需要基于 HTTP 的状态验证登录是否成功。参考相关测试用例代码。

 复制代码

```

1 // packages/work-server/__tests__/http.test.ts

```



```

2 // ...
3 describe('work-server', () => {
4   //...
5   test('login action', async () => {
6     const loginStatusUrl = `http://${workServerHost}:${workServerPort}/api/get/
7     const noLoginRes = await nodeFetch(loginStatusUrl);
8     const noLoginjson = await noLoginRes.json();
9     expect(noLoginjson).toStrictEqual({ username: null, uuid: null });
10
11    // 进行登录
12    const signInUrl = `http://${workServerHost}:${workServerPort}/api/post/acco
13    const signInRes = await nodeFetch(signInUrl, {
14      body: JSON.stringify({
15        username: 'admin001',
16        password: md5('88888888')
17      }),
18      // credentials: 'same-origin',
19      headers: {
20        'content-type': 'application/json'
21      },
22      method: 'POST'
23    });
24    const signInHeaders = signInRes.headers;
25    const signInJson = await signInRes.json();
26    expect(signInJson).toStrictEqual({
27      data: {
28        allow: true,
29        username: 'admin001',
30        uuid: '00000000-aaaa-bbbb-cccc-ddddeeee0001'
31      },
32      success: true,
33      message: '登录成功'
34    });
35
36    // 登录后再判断登录态
37    const cookie = signInHeaders.get('set-cookie') || '';
38    const hasLoginRes = await nodeFetch(loginStatusUrl, {
39      headers: {
40        credentials: 'same-origin',
41        cookie
42      }
43    });
44    const hasLoginJson = await hasLoginRes.json();
45    expect(hasLoginJson).toStrictEqual({
46      allow: true,
47      username: 'admin001',
48      uuid: '00000000-aaaa-bbbb-cccc-ddddeeee0001'
49    });
50  });
51
52  //...
53 });

```

这段代码，我们在调用登录的 HTTP 接口后，验证了登录成功数据，最后再验证 HTTP 响应中是否存在用户的登录 Cookie 数据。

Node.js 服务测试到这里，还要进行测试覆盖率的统计，这个配置跟前端单元测试类似，我们就不重复讲了。

现在，我们做了服务端的单元测试，不过，全栈项目的“开发自测”工作还“任重道远”，还有很多额外的测试工作。

我们举一个例子，假设开发了个全栈项目，用户的手机性能不足或者浏览器版本太低，导致页面遇到性能瓶颈卡顿了，这时候，我们可以告知用户，优先换个手机或者浏览器，来解决性能问题，那如果是服务端遇到性能瓶颈问题了，导致用户使用不了功能，要怎么快速解决问题呢？

这个服务端瓶颈是很难快速解决的，但是我们可以在开发自测阶段，尽量排查出瓶颈问题，也就是要对服务端代码的性能指标做测试，也叫基准测试。

如何给 Node.js 服务端做基准测试

基准测试，英文是 Benchmarking 或 Benchmark，也可以简称 Bench。

“基准测试，指通过设计科学的测试方法、测试工具和测试系统，实现对一类测试对象的某项性能指标进行定量的和可对比的测试”。

基准测试的测试过程，是面向“某项性能指标”，而且突出“定量”和“可对比”的特性。比如在服务端中，最直接的性能指标是“单位时间能支持的请求数量”。

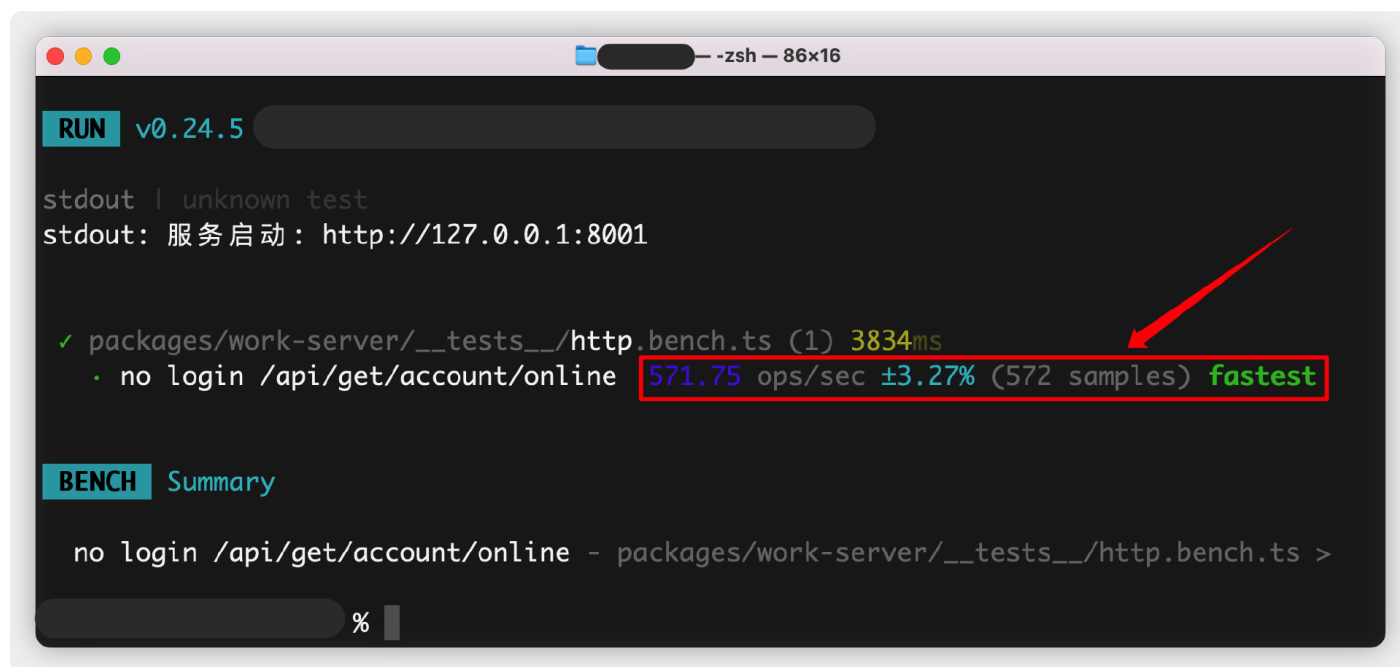
举个最简单的例子，就是服务端提供的 HTTP 接口，能在一秒内支撑多少次请求处理。我们用课程的登录态验证接口，基于 Vitest 做一次基准测试，来测试一秒内的请求数量情况。用 Vitest 的代码测试参考。

 复制代码

```
1 // packages/work-server/__tests__/http.bench.ts
2 import { bench, expect, afterAll, beforeAll } from 'vitest';
3 // ...
4 bench(
5   'no login /api/get/account/online',
```

```
6   async () => {
7     const url = `http://${workServerHost}:${workServerPort}/api/get/account/onl
8     const res = await nodeFetch(url);
9     const json = await res.json();
10    expect(json).toStrictEqual({ username: null, uuid: null });
11  },
12  {
13    time: 1000
14  }
15 );
```

用 Vitest 执行基准测试代码后，看结果截图。



```
RUN v0.24.5

stdout | unknown test
stdout: 服务启动: http://127.0.0.1:8001

✓ packages/work-server/__tests__/http.bench.ts (1) 3834ms
  · no login /api/get/account/online 571.75 ops/sec ±3.27% (572 samples) fastest

BENCH Summary

no login /api/get/account/online - packages/work-server/__tests__/http.bench.ts >
% |
```

 复制代码

```
1 571.75 ops/sec ±3.27% (572 samples) fastest
```

在基准测试结果中，描述性能指标数据，就是截图中红框标出的内容。

- “571.75 ops/sec”，其中“ops/sec”是单位，表示每秒钟执行测试代码的次数，这个数值越大，表示这个代码每秒能执行更多次数，性能就越好。
- “±3.27%”是测试过程中所有数据的方差，你可以通俗理解成性能数据的统计误差。
- (572 samples)，表示取样的内容。

需要注意，目前 Vitest 的基准测试（benchmark）功能还是属于“实验”阶段，后面可能有大改动。比如，这个性能结果，我们用的是 Vitest 的 0.24.5 版本，现在最新版本有一些变化。

如果你不想用 Vitest 进行基准测试，也可以考虑用 Node.js 的其他基准测试工具，例如 autocannon（[🔗https://www.npmjs.com/package/autocannon](https://www.npmjs.com/package/autocannon)）。autocannon 严格上算是一种压力测试工具，换个角度看，基准测试，也可以算是某种层度上的“压力测试”，就是在指定“维度”下，针对某个“性能指标”，测试能支持住多大的“压力”。

总结

我们围绕 Node.js 服务端的单元测试展开学习，首先归纳了前端和服务端单元测试的差异点。

- 前端的单元测试，内容主要是验证 DOM 结构、DOM 事件和 HTTP 请求的使用。
- 服务端的单元测试，内容主要是验证数据库操作、业务逻辑代码和 HTTP 请求，其中 HTTP 请求相关内容包括 HTTP 页面、HTTP 接口和 HTTP 状态数据。

服务端的性能指标测试，也就是服务端的基准测试，主要是基于“某项性能指标”来“定量”测试和对比程序运行情况，也可以算是一种程序的“压力测试”。

要记得，在开发过程中，测试是为了保障代码质量，用什么框架工具都不是重点。真正的重点是如何根据个人或者团队的情况，设计低成本、高性价比的测试设计方案，基于设计方案，选择趁手或者熟悉的技术工具。

思考题

我们提到了服务端做基准测试的必要性，那为什么前端项目代码很少见到做基准测试？

欢迎留言参与讨论，我们下节课见。

[🔗完整的代码在这里](#)

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 加餐 | 实战篇思考题答疑（下）

下一篇 33 | 页面功能扩展：如何对Vue.js全栈项目做优雅扩展？

精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。