

Cypress中文教程--从入门到精通

入门篇

cypress的介绍与安装

Cypress是一款现代化的JavaScript端到端测试框架，它是一款基于Node.js的开源测试工具。它具有易于安装和使用、内置断言库和高效的并行化测试等特点，可以让测试工程师和开发人员轻松地编写和运行测试用例。Cypress提供了丰富的API和命令，可以模拟用户在浏览器中的交互，并对网页元素进行操作和断言。在Cypress中，测试用例会在一个真实的浏览器环境中运行，因此可以更加准确地模拟用户的操作。

安装Cypress非常简单，只需要使用npm安装即可。可以通过以下命令安装最新版的Cypress：

```
1 npm install cypress --save-dev
```

安装完成后，可以使用以下命令打开Cypress测试运行器：

```
1 npx cypress open
```

执行该命令后，Cypress测试运行器会自动打开，并显示可用的测试用例。可以通过在测试运行器中选择要运行的测试用例，然后单击“运行”按钮来运行测试用例。

```
1 describe('My First Test', () => {
2   it('Does not do much!', () => {
3     expect(true).to.equal(true)
4   })
5 })
```

编写第一个测试用例

在Cypress中编写第一个测试用例非常简单。首先，在项目的根目录下创建一个名为 `cypress/integration` 的文件夹，用于存放测试用例文件。然后，创建一个名为 `example.spec.js` 的文件，并将以下代码复制到该文件中：

```
1 describe('My First Test', () => {
2   it('Does not do much!', () => {
3     expect(true).to.equal(true)
4   })
5 })
```

在上面的代码中，我们使用 `describe` 和 `it` 函数来定义一个测试用例。`describe` 函数用于定义一个测试套件，`it` 函数用于定义一个具体的测试用例。在这个例子中，我们定义了一个名为“My First Test”的测试套件，包含了一个名为“Does not do much!”的测试用例。在测试用例中，我们使用 `expect` 函数进行断言，判断 `true` 是否等于 `true`。

接下来，我们可以在Cypress测试运行器中运行这个测试用例。在运行器中，选择 `example.spec.js` 文件，然后单击“运行”按钮。Cypress会在浏览器中打开一个新的窗口，并自动运行该测试用例。在测试运行器中可以看到测试用例的执行结果。

如何使用cypress测试网页元素

在Cypress中，要测试网页元素，首先需要获取到它们的选择器。可以使用Chrome浏览器的开发者工具（或其他浏览器的类似工具）来查找选择器。一旦找到选择器，就可以在测试用例中使用 `cy.get()` 命令获取该元素。

例如，如果要测试一个id为“my-button”的按钮，可以在测试用例中使用以下代码：

```
1 describe('My First Test', () => {
2   it('clicks the button', () => {
3     cy.visit('https://example.com')
4     cy.get('#my-button').click()
5   })
6 })
```

在上面的代码中，我们首先使用 `cy.visit()` 命令访问 `https://example.com` 网页，然后使用 `cy.get()` 命令获取id为“my-button”的按钮，并使用 `click()` 命令模拟点击操作。

除了 `cy.get()` 命令，Cypress还提供了其他命令来测试网页元素，如 `cy.contains()`、`cy.find()` 等。可以查阅Cypress官方文档获取更多命令的详细信息。

cypress的命令式编程模型

Cypress使用命令式编程模型来编写测试用例，也就是说，测试用例中的每一行代码都是一个命令，Cypress会按照代码的顺序依次执行这些命令。这种编程模型非常直观，也很容易理解和调试。

例如，以下代码是一个简单的Cypress测试用例：

```
1 describe('My First Test', () => {
```

```
2  it('visits the homepage', () => {
3    cy.visit('https://example.com')
4    cy.contains('h1', 'Example Domain')
5    cy.url().should('include', 'example.com')
6  })
7 })
8
```

在上面的代码中，我们定义了一个测试用例，名称为“My First Test”，测试内容是访问 <https://example.com> 网页，检查网页中是否包含标题为“Example Domain”的h1元素，并验证当前URL是否包含“example.com”。

需要注意的是，Cypress的命令式编程模型使得测试用例的执行速度非常快。在执行每个命令时，Cypress会自动等待元素或事件的出现，因此不需要手动添加等待代码。同时，Cypress还会自动捕捉错误，并将其显示在测试结果中，方便调试。

断言的使用和实现

在编写测试用例时，我们需要使用断言来验证我们的预期结果是否与实际结果相符。Cypress提供了一系列丰富的断言方法，可以帮助我们轻松地编写和执行断言。

以下是一些常用的Cypress断言方法：

- `should`：用于在元素上执行断言，例如验证元素是否可见、是否包含指定文本等。示例代码：

```
1 cy.get('button').should('be.visible')
2 cy.get('h1').should('have.text', 'Welcome to my website')
```

- `expect`：用于在JavaScript代码中执行断言，例如验证变量的值、函数的返回值等。示例代码：

```
1 expect(true).to.be.trueexpect(42).to.equal(42)
2 expect([1, 2, 3]).to.have.lengthOf(3)
```

- `assert`：用于在JavaScript代码中执行断言，与 `expect` 方法类似，但语法稍有不同。示例代码：

```
1 assert.isTrue(true)
2 assert.strictEqual(42, 42)
3 assert.lengthOf([1, 2, 3], 3)
```

需要注意的是，Cypress的断言方法都是链式调用的，可以在同一个语句中使用多个断言方法，例如：

```
1 cy.get('#myElement')
2   .should('be.visible')
3   .should('have.text', 'Hello, world!')
```

上面的代码中，我们先通过 `get` 方法获取一个ID为 `myElement` 的元素，然后对该元素执行两个断言：首先验证该元素是否可见，然后验证该元素的文本是否为“Hello, world!”。

总之，Cypress的断言方法非常灵活，可以根据测试需求进行选择 and 组合，方便编写和执行各种类型的测试用例。

如何在测试中模拟用户操作

在 Cypress 中，可以使用 `cy.get()` 方法定位页面元素，并使用 `.click()` 方法来模拟点击操作。例如，以下代码段将单击一个具有 `id` 为 `my-button` 的按钮：

```
1 cy.get('#my-button').click()
```

类似地，可以使用 `.type()` 方法模拟用户在页面元素上输入文本。例如，以下代码段将在一个具有 `id` 为 `my-input` 的输入框中输入文本 `Hello, World!`：

```
1 cy.get('#my-input').type('Hello, World!')
```

此外，Cypress 还支持模拟键盘和鼠标事件，如 `cy.type()`、`cy.trigger()` 等方法，可用于更复杂的测试场景。

需要注意的是，由于 Cypress 采用了异步架构，因此在对元素进行操作时需要添加 `.then()` 方法来保证执行顺序。例如：

```
1 cy.get('#my-button')
2   .then(($button) => {
3     if ($button.is(':disabled')) {
4       // Do something
5     } else {
6       $button.click()
7     }
8   })
9
```

以上代码首先使用 `.get()` 方法获取 `id` 为 `my-button` 的按钮，然后使用 `.then()` 方法获取按钮元素的状态，最后执行点击操作。

在测试中使用fixtures和data-driven测试

Cypress 中的 fixture 是指测试用例中使用的静态数据，通常存储在文件中。可以使用 `.fixture()` 方法来加载 fixture 文件，例如：

```
1 cy.fixture('example.json').then((data) => {
2   // do something with data
3 })
4
```

以上代码将加载名为 `example.json` 的 fixture 文件，并在 `.then()` 方法中接收 fixture 文件中的数据。fixture 文件可以是 JSON、YAML、CSV、XML 等格式。

另外，Cypress 还支持 data-driven 测试，即在一个测试用例中执行多个测试数据，以验证同一个测试用例在多个数据下的执行结果。例如：

```
1 describe('My Data-Driven Test Suite', () => {
2   [
3     { name: 'Alice', age: 20 },
4     { name: 'Bob', age: 30 },
5     { name: 'Charlie', age: 40 }
6   ].forEach((data) => {
7     it(`should greet ${data.name} correctly`, () => {
8       cy.visit('/')
9       cy.get('#name-input').type(data.name)
10      cy.get('#age-input').type(data.age)
11      cy.get('#greet-button').click()
12      cy.get('#greeting').should('contain', `Hello, ${data.name}! You are ${data
13      })
14    })
15  })
16
```

以上代码定义了一个数据数组，并在测试套件中使用 `.forEach()` 方法循环执行数据，每次都执行相同的测试用例，以验证不同数据下的测试结果。在测试用例中，可以使用 `data` 对象来访问测试数据。

如何使用环境变量进行测试配置

在 Cypress 中，可以通过环境变量来动态配置测试用例的行为。例如，可以通过环境变量来配置测试用例运行的端口号、测试的 URL、测试用例的数据等。

Cypress 提供了 `Cypress.env()` 方法来访问环境变量。可以在 Cypress 的配置文件中设置默认的环境变量值，例如：

```
1 //cypress.json
2 {"baseUrl": "http://localhost:3000", "env": {"username": "testuser", "password": " "}}
```

以上配置文件中，`baseUrl` 是一个默认的环境变量值，`env` 是一个对象，包含了两个环境变量 `username` 和 `password` 的默认值。

在测试用例中，可以使用 `Cypress.env()` 方法来访问环境变量，例如：

```
1 describe('My Test Suite', () => {
2   it('should log in using environment variables', () => {
3     cy.visit(Cypress.env('baseUrl'))
4     cy.get('#username').type(Cypress.env('username'))
5     cy.get('#password').type(Cypress.env('password'))
6     cy.get('#login-button').click()
7     cy.url().should('eq', `${Cypress.env('baseUrl')}/dashboard`)
8   })
9 })
```

以上测试用例中，使用了 `Cypress.env()` 方法来访问 `baseUrl`、`username` 和 `password` 环境变量，并在测试用例中使用它们来访问网站、输入登录信息、点击登录按钮，并验证登录后的 URL 是否正确。

进阶篇

测试用例的组织与管理

在编写测试用例的过程中，测试用例的组织和管理非常重要，这可以帮助我们快速地维护和更新测试用例，提高测试用例的可读性和可维护性。下面是一些测试用例组织和管理的最佳实践：

将测试用例按功能模块分组，并使用 `describe` 块描述每个功能模块。

- 在每个 `describe` 块中，使用 `before` 和 `after` 钩子函数，分别用于测试用例执行前和执行后的初始化和清理工作。
- 使用 `context` 块描述不同的场景或状态，并在该块内编写测试用例。
- 对于复杂的测试用例，可以将测试用例拆分为多个独立的测试步骤，并使用 `it` 块逐步描述测试步骤。

- 在测试用例中使用注释，以提高测试用例的可读性和可维护性。

以下是示例代码，展示了如何组织和管理测试用例：

```
1 describe('Login functionality', () => {
2   before(() => {
3     cy.visit('/login')
4   })
5
6   after(() => {
7     // Clean up user session after each test
8     cy.clearCookies()
9   })
10
11  context('When user enters valid login credentials', () => {
12    it('Should log user in successfully', () => {
13      // Test steps for successful login
14    })
15  })
16
17  context('When user enters invalid login credentials', () => {
18    it('Should show an error message', () => {
19      // Test steps for error message display
20    })
21  })
22 })
```

在上面的示例中，我们使用了describe块和context块来组织和管理测试用例。每个describe块描述了一个功能模块，使用before和after钩子函数进行初始化和清理工作。在每个context块中，我们编写了不同的测试用例。

定位网页元素的最佳实践

在测试中，定位网页元素是非常重要的一步。在使用 Cypress 进行测试时，使用正确的定位方式可以让你的测试用例更加稳定和可靠。下面是一些定位网页元素的最佳实践：

- 避免使用绝对路径，而是使用相对路径。使用绝对路径会让你的测试用例更加脆弱，因为任何 HTML 或 CSS 的更改都可能导致测试用例失败。
- 在选择器中避免使用标签名。对于一个给定的网站，标签名可能会发生变化，但是选择器的 ID 或类名通常不会变化。
- 避免使用复杂的 CSS 选择器。CSS 选择器可以非常强大，但是也很容易出错。尽可能地使用简单的选择器。
- 使用 data-* 属性来定位元素。这种方式不依赖于网页的结构和样式，而是依赖于数据属性。

下面是一个例子，展示了如何使用相对路径和 `data-*` 属性来定位网页元素：

```
1 // 通过 data-* 属性定位元素
2 cy.get('[data-testid="submit-button"]').click()
3
4 // 通过类名定位元素
5 cy.get('.form-input').type('username')
6
7 // 通过 ID 定位元素
8 cy.get('#password-input').type('password')
9
10 // 通过相对路径定位元素
11 cy.get('.form-group').find('.form-label').should('have.text', 'Username')
```

使用这些最佳实践，你可以编写更加健壮的测试用例，而且这些测试用例更加易于维护和更新。

在测试中使用自定义命令

在 Cypress 中，自定义命令（Custom Commands）可以帮助我们更好地组织测试代码，避免测试用例变得冗长复杂。自定义命令可以用来封装重复使用的代码逻辑，从而提高代码的可读性和可维护性。

下面我们来看一个具体的示例。假设我们需要在多个测试用例中反复访问某个特定的网页，并进行一些共同的操作。在这种情况下，我们可以使用 `cy.visit` 命令和 `cy.intercept` 命令来模拟访问和拦截网络请求，并将它们封装成一个自定义命令：

```
1 Cypress.Commands.add('login', () => {
2   cy.visit('/login');
3   cy.get('#username').type('myusername');
4   cy.get('#password').type('mypassword');
5   cy.get('#login-button').click();
6   cy.intercept('GET', '/api/user').as('getUser');
7   cy.wait('@getUser');
8 });
9
```

在这个自定义命令中，我们首先使用 `cy.visit` 命令访问登录页面，然后使用 `cy.get` 命令获取用户名、密码和登录按钮的元素，并模拟用户输入用户名和密码、点击登录按钮进行登录。最后，我们使用 `cy.intercept` 命令拦截请求 `/api/user`，并使用 `cy.wait` 命令等待该请求完成。

有了这个自定义命令后，我们可以在多个测试用例中直接调用它来模拟登录：

```

1 describe('My test suite', () => {
2   beforeEach(() => {
3     cy.login();
4   });
5
6   it('Test case 1', () => {
7     // ...
8   });
9
10  it('Test case 2', () => {
11    // ...
12  });
13 });
14

```

在这个测试套件中，我们使用 `beforeEach` 钩子函数在每个测试用例运行前都调用了 `cy.login` 命令，从而实现了登录操作的重复使用。

自定义命令可以大大简化测试代码，提高测试代码的可读性和可维护性。但需要注意的是，自定义命令也可能会导致测试代码的可读性和可维护性变差，特别是在自定义命令变得越来越复杂、过于抽象或者用途过于广泛时。因此，在使用自定义命令时，需要权衡利弊，仔细考虑自定义命令的设计和Usage方式。

如何使用cypress进行API测试

在Cypress中进行API测试是一个很常见的需求。Cypress提供了许多可用于测试API的命令和方法。下面是一些示例：

- 发送GET请求并断言响应状态码

```

1 cy.request('GET', '/api/users').then((response) => {
2   expect(response.status).to.eq(200)
3 })
4

```

- 发送POST请求并断言响应中的数据

```

1 cy.request('POST', '/api/users', {
2   name: 'John Doe',
3   email: 'johndoe@test.com'
4 }).then((response) => {
5   expect(response.body).to.have.property('name', 'John Doe')
6 })

```

- 发送PUT请求并断言响应中的数据

```
1 cy.request('PUT', '/api/users/123', {
2   name: 'John Doe',
3   email: 'johndoe@test.com'
4 }).then((response) => {
5   expect(response.body).to.have.property('email', 'johndoe@test.com')
6 })
7
```

- 发送DELETE请求并断言响应状态码

```
1 cy.request('DELETE', '/api/users/123').then((response) => {
2   expect(response.status).to.eq(204)
3 })
```

在进行API测试时，建议将API的URL和数据定义为常量或使用fixtures进行组织，以提高代码的可维护性。此外，还可以使用Cypress的拦截功能来模拟API响应，以避免对真实的API进行测试。

如何使用cypress进行移动端测试

移动端测试是Web前端测试中的一个重要部分。cypress也提供了相应的工具和插件，可以轻松地在移动端进行测试。

- 安装插件：

cypress提供了多个插件来帮助我们在移动端进行测试，比如[cypress-plugin-snapshots](#)、[cypress-react-native](#)等等。我们可以根据项目需要，选择对应的插件进行安装。

以cypress-plugin-snapshots为例，安装方式如下：

```
1 npm install -D cypress-plugin-snapshots
```

- 配置cypress.json

在cypress.json文件中添加以下内容，以允许在移动设备上测试：

```
1 {
```

```
2  "viewportWidth": 375,  
3  "viewportHeight": 812,  
4  "cy": {  
5    "viewport": {  
6      "name": "iphone-x"  
7    }  
8  }  
9 }  
10
```

其中，viewportWidth和viewportHeight表示浏览器窗口的大小，cy.viewport.name表示使用的移动设备类型，这里使用了iPhone X的尺寸。

- 编写测试用例

在测试用例中，我们可以使用cypress提供的命令来模拟移动设备上的操作，比如点击、滑动等等。以下是一个简单的测试用例：

```
1 describe('Mobile Test', () => {  
2   it('should open google.com on iPhone X', () => {  
3     cy.visit('https://www.google.com')  
4     cy.get('input[type="text"]').type('cypress').type('{enter}')  
5     cy.screenshot()  
6     cy.contains('cypress.io').click()  
7     cy.url().should('include', 'https://www.cypress.io/')  
8   })  
9 })  
10
```

在这个测试用例中，我们首先访问Google网站，然后在搜索框中输入关键字“cypress”，按下回车键，进入搜索结果页面。然后我们对搜索结果进行截图，点击第一个结果中包含“cypress.io”的链接，并验证当前URL是否包含“<https://www.cypress.io/>”。

- 运行测试用例

为了在移动设备上测试，我们需要通过命令行参数 `--config` 来指定我们希望在哪个设备上运行测试。

假设我们希望在iPhone X上运行测试，可以通过以下命令来运行测试：

```
1 npx cypress run --config viewportWidth=375,viewportHeight=812,cy.viewport.name=i
```

在运行测试之前，我们需要确保我们的设备已连接到电脑，并且已启动了相应的模拟器或浏览器。

在测试中使用插件和第三方库

Cypress具有强大的插件系统，允许你扩展其功能，自定义命令、添加报告等等。在本章节中，我们将讨论如何使用Cypress插件和第三方库

安装和使用插件

Cypress插件可以通过npm包管理器安装，例如，如果你想安装一个Cypress插件，你可以在你的项目根目录下运行以下命令：

```
1 npm install cypress-plugin-name --save-dev
```

安装完毕后，在你的 `cypress/plugins/index.js` 文件中引入该插件即可，例如：

```
1 const pluginName = require('cypress-plugin-name');
2
3 module.exports = (on, config) => {
4   // 注册插件
5   on('task', pluginName());
6
7   // 在这里可以对config进行修改
8   config.baseUrl = 'https://example.com';
9
10  return config;
11 };
12
```

- 使用第三方库

Cypress默认包含了许多功能强大的库，如jQuery和Lodash等，但你也可以使用其他第三方库。要使用其他库，你需要先将其安装为npm依赖项，例如：

```
1 npm install moment --save-dev
```

然后，你可以在你的测试代码中引入该库并使用它，例如：

```
1 import moment from 'moment';
2
3 describe('My test suite', () => {
4   it('should use moment library', () => {
5     const today = moment().format('YYYY-MM-DD');
```

```
6     expect(today).to.equal('2023-04-18');
7   });
8 });
9
```

需要注意的是，Cypress测试代码运行在浏览器环境中，而不是Node.js环境中，因此有些库可能无法在Cypress中正常工作。在使用第三方库时，应该先测试它们是否与Cypress兼容。

Cypress如何生成测试报告

在cypress中生成测试报告是很重要的，它能够帮助测试工程师更好地跟踪测试结果、定位问题，以及向团队和利益相关者汇报测试成果。cypress本身并不提供测试报告生成功能，但是可以使用第三方库或插件来生成测试报告。

以下是一些常见的测试报告生成库或插件：

- Mochawesome

这是一个非常受欢迎的测试报告生成工具，它可以为Mocha测试框架生成漂亮的HTML测试报告。要使用它，你需要在你的项目中安装 `mochawesome` 和 `mochawesome-merge` 两个npm包。安装完成后，你需要在cypress.json文件中配置测试报告生成器，指定报告的输出路径和格式，示例配置如下：

```
1 {
2   "reporter": "mochawesome",
3   "reporterOptions": {
4     "reportDir": "cypress/reports",
5     "overwrite": false,
6     "html": false,
7     "json": true
8   }
9 }
10
```

- Cypress Mochawesome Reporter

这是一个专门为cypress设计的测试报告生成插件，它基于Mochawesome，并且添加了一些特定于cypress的功能。要使用它，你需要在你的项目中安装 `cypress-mochawesome-reporter` npm包。安装完成后，你需要在cypress/plugins/index.js文件中配置插件，示例配置如下：

```
1 const { cypressMochawesomeReporter } = require('cypress-mochawesome-reporter');
2
3 module.exports = (on, config) => {
4   on('after:run', (results) => {
5     return cypressMochawesomeReporter(results, {
```

```
6     reportDir: 'cypress/reports',
7     overwrite: false,
8     html: false,
9     json: true
10  });
11  });
12  };
13
```

以上是两个常见的测试报告生成工具，还有其他很多选择，你可以根据自己的需求和偏好选择一个适合自己的测试报告生成器。无论你使用哪个测试报告生成器，记得在测试运行完成后查看测试报告，并及时修复测试中发现的问题。

如何进行性能测试

Cypress可以通过 `cy.clock()` 和 `cy.tick()` 命令来模拟延迟操作和异步操作，从而实现性能测试。

在Cypress中，可以使用以下几个方法进行性能测试：

- 通过mock API请求来模拟网络请求

可以使用 `cy.route()` 和 `cy.server()` 命令来模拟API请求和响应。这样可以测试应用程序在不同网络条件下的性能表现。示例代码：

```
1  it('should test API performance', function() {
2    cy.server()
3    cy.route('GET', '/api/data', 'fixture:api-response.json').as('getData')
4
5    cy.clock()
6    cy.visit('/dashboard')
7
8    cy.tick(1000)
9    cy.wait('@getData')
10   cy.tick(500)
11  })
12
```

- 通过模拟用户操作来测试性能

可以使用 `cy.intercept()` 命令来拦截浏览器的请求和响应，从而模拟用户的操作行为，测试页面的性能表现。示例代码：

```
1  it('should test page loading performance', function() {
```

```

2   cy.intercept('/api/data', (req) => {
3     req.continue((res) => {
4       res.body = {
5         "data": "some data"
6       }
7     })
8   })
9
10  cy.clock()
11  cy.visit('/dashboard')
12
13  cy.tick(1000)
14  cy.get('#dashboard-page').should('be.visible')
15  cy.tick(2000)
16  cy.get('#dashboard-data').should('be.visible')
17 })
18

```

- 通过性能分析工具来测试性能

可以使用Cypress内置的性能分析工具或其他第三方工具，来测试页面的加载时间、性能瓶颈等性能指标。示例代码：

```

1  it('should test page performance using lighthouse', function() {
2    cy.lighthouse()
3      .then((lighthouseReport) => {
4        const performanceScore = lighthouseReport.categories.performance.score
5
6        expect(performanceScore).to.be.greaterThan(0.8)
7      })
8  })
9

```

以上是一些简单的性能测试示例，可以根据实际需要进行扩展和修改。

cypress与CI/CD持续集成

在本章节中，我们将介绍如何将Cypress与持续集成（Continuous Integration, CI）工具集成，以实现持续集成和自动化测试。

持续集成是一种软件开发实践，旨在通过频繁地将代码集成到主干代码库中，并通过自动化测试和构建等工具自动检查和构建代码，从而快速发现和解决问题。这种方式可以让团队更快地交付代码，减少出错的风险，并增强代码的质量和稳定性。

Cypress 可以与许多流行的 CI 工具集成，例如 Travis CI、CircleCI、Jenkins、GitLab CI、GitHub Actions 等。这些工具都提供了可以轻松配置 Cypress 测试运行的插件或集成。

下面以 Travis CI 和 CircleCI 为例，介绍如何将 Cypress 与这两个 CI 工具进行集成。

- Travis CI

Travis CI 是一个流行的云端持续集成服务，可以与 GitHub 等代码托管平台集成。Travis CI 可以在每次代码提交时自动运行构建和测试，以确保代码的质量和稳定性。

以下是在 Travis CI 上运行 Cypress 测试的基本步骤：

- 在 GitHub 上创建一个新的仓库，并将代码推送到该仓库。
- 在 Travis CI 上创建一个新的项目，选择要构建和测试的代码仓库。
- 在 `.travis.yml` 文件中添加以下配置：

```
1 language: node_js
2 node_js:
3   - "14"
4 addons:
5   chrome: stable
6 cache:
7   directories:
8     - ~/.npm
9     - ~/.cache
10 install:
11   - npm ci
12 script:
13   - npm run cy:run
14
```

在上述配置中，我们使用了 Node.js 14 版本，安装了 Chrome 浏览器，并缓存了 npm 和 cypress 的目录。在 `script` 部分，我们运行 `npm run cy:run` 命令来运行 Cypress 测试。

- 将 `.travis.yml` 文件提交到 GitHub 仓库中，此时 Travis CI 将会自动运行测试。

- CircleCI

CircleCI 是一个云端持续集成服务，支持多种语言和技术栈，包括 Node.js、Ruby、Python、Go、Docker 等。与 Travis CI 类似，CircleCI 也可以与 GitHub 等代码托管平台集成。

以下是在 CircleCI 上运行 Cypress 测试的基本步骤：

- 在 GitHub 上创建一个新的仓库，并将代码推送到该仓库。
- 在 CircleCI 上创建一个新的项目，选择要构建和测试的代码仓库。
- 在 `.circleci/config.yml` 文件中添加以下配置：

```
1 version: 2.1
2 jobs:
3   build:
4     docker:
5       - image: cypress/browsers:node14.17.0-chrome94-ff91
6     working_directory: /
7
```

精通篇

在cypress中使用自动化测试框架

Cypress是一个功能强大的自动化测试工具，但是它并不是一个完整的测试框架。尽管Cypress可以满足大部分Web应用程序的测试需求，但是如果你需要进行更高级的测试，例如功能测试、集成测试和端到端测试，那么你可能需要使用自动化测试框架。

自动化测试框架可以帮助你扩展Cypress的功能，并提供更多的测试工具和库，以便更好地管理测试用例和测试数据。本章将介绍如何在Cypress中使用自动化测试框架。

- 什么是自动化测试框架？

自动化测试框架是一个基于编程语言的软件框架，用于编写、运行和管理自动化测试脚本。这些框架通常包括一组库、工具和技术，以简化测试代码的编写和维护，并提高测试的可靠性和效率。

一些常见的自动化测试框架包括JUnit、TestNG、Selenium、Protractor和Robot Framework等。

- 如何在Cypress中使用自动化测试框架？

在Cypress中使用自动化测试框架通常需要两个步骤：

- 安装测试框架

首先，你需要在项目中安装自动化测试框架的npm包。这可以通过npm或yarn包管理工具来完成，例如：

```
1 npm install mocha chai --save-dev
```

这个命令会在项目中安装Mocha和Chai测试框架的npm包。

- 编写测试代码

然后，你需要编写测试用例代码，并将其集成到Cypress测试代码中。这可以通过创建自定义命令或使用Cypress插件来完成。

下面是一个使用Mocha和Chai测试框架的示例测试用例代码：

```
1 describe('Login Tests', () => {
```

```
2  it('should log in successfully', () => {
3    cy.visit('https://www.example.com/login')
4
5    cy.get('#username').type('testuser')
6    cy.get('#password').type('password')
7    cy.get('#login-button').click()
8
9    cy.url().should('eq', 'https://www.example.com/dashboard')
10 })
11
12 it('should display an error message for invalid credentials', () => {
13   cy.visit('https://www.example.com/login')
14
15   cy.get('#username').type('testuser')
16   cy.get('#password').type('invalidpassword')
17   cy.get('#login-button').click()
18
19   cy.get('#error-message').should('be.visible')
20 })
21 })
22
```

在这个示例中，我们使用Mocha和Chai测试框架编写了两个测试用例，分别测试了成功登录和输入无效凭据的情况。我们还使用Cypress命令来模拟用户操作，并使用断言验证了测试结果。

使用cypress进行全链路测试

在软件开发生命周期中，全链路测试是一个重要的测试类型，它涉及到整个应用程序的各个组件和流程。全链路测试能够验证应用程序在生产环境中的稳定性和可靠性。在本章节中，我们将介绍如何使用Cypress进行全链路测试。

什么是全链路测试？

全链路测试是一种测试方法，旨在测试整个应用程序的所有部分和功能。这包括前端、后端、数据库以及各种服务和第三方API等。全链路测试可以确保应用程序的各个组件之间的集成以及整个应用程序的功能正常运行。它是一个非常综合的测试方法，需要使用多种技术和工具。

Cypress的全链路测试

Cypress可以轻松进行全链路测试，因为它可以模拟用户的交互，从而测试整个应用程序的各个组件和功能。在Cypress中，你可以使用多个插件和库来模拟后端服务和第三方API。这些插件和库包括Cypress-graphql-mock、Cypress-faker、Cypress-socket-io和Cypress-xhr。

编写全链路测试

下面我们将介绍如何编写全链路测试。

1. 创建测试文件

首先，创建一个测试文件，并在文件中添加以下代码：

```
1 describe('Full end-to-end test', () => {
2   it('should visit the home page and navigate to other pages', () => {
3     cy.visit('/');
4     cy.contains('Sign In').click();
5     cy.url().should('include', '/login');
6     cy.get('#username').type('testuser');
7     cy.get('#password').type('testpassword');
8     cy.get('form').submit();
9     cy.url().should('include', '/dashboard');
10  });
11 });
```

2. 编写测试用例

接下来，编写测试用例。在本例中，我们访问网站首页，然后单击“Sign In”按钮，输入用户名和密码，然后登录。最后，我们检查 URL 是否包含“/dashboard”字符串。

3. 运行测试

现在我们可以运行测试，只需在终端中输入以下命令：

```
1 npx cypress run
```

Cypress 将会运行测试，并在终端中显示测试结果。

4. 运行测试覆盖率

你还可以使用 Cypress 运行测试覆盖率。要生成测试覆盖率报告，只需在终端中输入以下命令：

```
1 npx cypress run --env coverage=true
```

Cypress 将会生成一个覆盖率报告，并在浏览器中打开该报告。

如何进行多浏览器测试

在cypress中进行多浏览器测试可以帮助我们确保应用程序在各种浏览器和操作系统上的兼容性，因此在多浏览器测试时，需要考虑以下几个方面：

1. 安装浏览器：Cypress支持Chrome、Firefox和Electron浏览器，你需要安装并配置所需的浏览器，以便Cypress可以使用它们来运行测试。
2. 编写测试用例：你需要为每个浏览器编写测试用例，并在测试运行时指定要运行测试的浏览器。
3. 平台差异：不同浏览器在不同的平台上可能有所不同，如不同的操作系统或设备，因此在多浏览器测试时，需要测试不同的平台和设备组合，以确保应用程序在所有环境中都能正常运行。

以下是在cypress中进行多浏览器测试的步骤：

1. 安装所需的浏览器：Cypress支持Chrome、Firefox和Electron浏览器，你需要先安装并配置这些浏览器。
2. 在cypress.json文件中配置浏览器：在cypress.json文件中，你可以为每个浏览器设置一个别名，以便在测试运行时引用它们。以下是一个cypress.json文件的示例：

```
1 {"chrome": {"name": "Chrome", "browser": "chrome"}, "firefox": {"name":  
  "Firefox", "browser": "firefox"}, "electron": {"name": "Electron", "browser":  
  "electron"}}
```

3. 编写测试用例：你需要为每个浏览器编写测试用例，并在测试运行时指定要运行测试的浏览器。以下是一个基本的多浏览器测试示例：

```
1 describe('My App', function() {  
2   it('should load in Chrome', function() {  
3     cy.visit('/', { browser: 'chrome' })  
4     cy.get('h1').should('contain', 'Welcome')  
5   })  
6  
7   it('should load in Firefox', function() {  
8     cy.visit('/', { browser: 'firefox' })  
9     cy.get('h1').should('contain', 'Welcome')  
10  })  
11  
12  it('should load in Electron', function() {  
13    cy.visit('/', { browser: 'electron' })  
14    cy.get('h1').should('contain', 'Welcome')  
15  })  
16 })
```

4. 运行测试：在运行测试时，你需要指定要运行测试的浏览器。可以在命令行中使用--browser标志来指定要运行测试的浏览器，例如：

```
1 cypress run --browser chrome
```

以上是在cypress中进行多浏览器测试的基本步骤，你可以根据你的具体需求进行修改和扩展。

如何进行无头浏览器测试

无头浏览器是指没有用户界面的浏览器。在自动化测试中，无头浏览器常常用于模拟浏览器行为，以便测试网站或应用程序的功能。Cypress默认使用有头浏览器进行测试，但是也可以使用无头浏览器进行测试。

下面是使用无头浏览器进行测试的步骤：

1. 安装无头浏览器

Cypress支持多种无头浏览器，包括Chrome，Chromium和Electron。安装方法可以在Cypress官方文档中找到。

2. 配置Cypress

在配置文件cypress.json中配置无头浏览器，示例如下：

```
1 {"baseUrl": "http://localhost:3000", "chromeWebSecurity": false, "viewportWidth":
  1920, "viewportHeight": 1080, "testFiles":
  "**/*_headless_spec.js", "chromeWebSecurity": false, "env": {"NODE_ENV":
  "production", "HEADLESS": "true"}, "chromeExecutablePath":
  "/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome", "video":
  true, "videoUploadOnPasses": true, "videoCompression": 32, "numTestsKeptInMemory":
  5, "supportFile": "cypress/support/index.js", "pluginsFile":
  "cypress/plugins/index.js"}
```

其中，重要的配置项包括：

- `chromeWebSecurity`：是否启用Chrome的网页安全功能，默认为true，如果要使用无头浏览器测试，需要将其设置为false。
- `testFiles`：测试文件的路径，只有路径中包含"`_headless_spec.js`"的文件才会使用无头浏览器进行测试。
- `env`：环境变量，这里设置了一个名为"`HEADLESS`"的环境变量，用于判断是否使用无头浏览器进行测试。
- `chromeExecutablePath`：Chrome可执行文件的路径，Cypress默认会自动寻找可执行文件，但是如果需要使用自定义的Chrome可执行文件，可以使用这个配置项指定路径。
- `video`：是否录制视频。
- `supportFile`和`pluginsFile`：支持文件和插件文件的路径。

3. 编写测试用例

编写测试用例时，可以在文件名中加入"`_headless_spec.js`"，以便让Cypress使用无头浏览器进行测试。同时，在测试用例中也需要对环境变量进行判断，示例如下：

```
1 describe('Headless Testing', () => {
2   it('should display the correct title', () => {
3     cy.visit('/')
4     cy.get('h1').should('have.text', 'Hello, World!')
5     if (Cypress.env('HEADLESS')) {
6       cy.screenshot('headless-testing-title')
7     }
8   })
9 })
```

在这个测试用例中，如果环境变量"`HEADLESS`"的值为`true`，就会在测试过程中使用无头浏览器。

在使用Cypress进行无头浏览器测试时，需要使用Cypress提供的Xvfb插件或使用Headless Chrome。使用Xvfb插件时，需要先安装该插件并配置环境变量，然后在运行Cypress时指定该插件。使用Headless Chrome时，需要在Cypress配置文件中设置Chrome的启动参数，启用无头模式。这样就可以在无图形界面的环境下进行浏览器测试。同时，由于无头浏览器没有图形界面，测试执行的速度通常比有界面的浏览器要快，而且不会影响其他正在运行的程序。

使用无头浏览器测试还需要注意以下几点：

1. 无头浏览器缺少界面，测试过程中无法手动干预，因此需要编写更加完善的测试用例，确保测试用例的覆盖率足够高。
2. 无头浏览器的执行环境可能有界面的浏览器不一样，需要进行额外的测试和验证，确保测试结果的准确性和可靠性。
3. 无头浏览器可能会出现一些和有界面浏览器不同的问题，需要对这些问题进行额外的处理和解决。例如，无头浏览器可能会出现性能问题，需要进行调优和优化。

综上所述，使用无头浏览器测试可以提高测试的效率和可靠性，但需要注意一些额外的细节和问题。在实际应用中，需要根据具体的情况选择是否使用无头浏览器测试，并根据实际情况进行调整和优化。

使用cypress进行可靠性测试

在软件开发过程中，可靠性测试是必不可少的一部分，以确保系统在长时间运行时，不会出现问题。Cypress是一个非常适合进行可靠性测试的工具，以下是几个使用Cypress进行可靠性测试的示例。

1. 资源泄漏测试

在长时间运行的情况下，资源泄漏是一种常见的问题。可以使用Cypress编写测试来检测资源泄漏。

```

1 describe('Memory leak test', () => {
2   let oldMemoryUsage;
3   let newMemoryUsage;
4
5   before(() => {
6     oldMemoryUsage = process.memoryUsage().heapUsed;
7   });
8
9   after(() => {
10    newMemoryUsage = process.memoryUsage().heapUsed;
11    expect(newMemoryUsage).to.be.closeTo(oldMemoryUsage, 100000);
12  });
13
14  it('Should not leak memory', () => {
15    // your test code here
16  });
17 });

```

在这个测试中，我们使用 `process.memoryUsage()` 方法来检测内存使用情况，并使用 `expect()` 断言来检测内存使用是否超过了100k。在测试运行之前和之后，我们分别检查了内存使用情况。如果内存使用量超过了我们设置的阈值，这个测试就会失败。

2. 重复测试

重复测试是可靠性测试的重要部分。Cypress允许我们使用 `.retry()` 方法来自动重试失败的测试。

```

1 describe('Retry test', () => {
2   it('Should pass eventually', () => {
3     cy.visit('/url')
4       .get('#element')
5       .should('have.text', 'Expected text')
6       .retry(3);
7   });
8 });

```

在这个测试中，我们使用 `.retry()` 方法来重试获取元素的断言，最多重试3次。如果在3次尝试之后，元素仍然不具备我们预期的文本，测试将失败。

3. 网络断开测试

Cypress允许我们轻松模拟网络断开情况，以测试应用程序在断开网络时的行为。

```

1 describe('Offline test', () => {
2   it('Should display offline message', () => {

```

```
3     cy.visit('/url')
4     .window()
5     .then((win) => {
6         win.navigator.onLine = false;
7         cy.visit('/url')
8             .contains('You are offline');
9     });
10 });
11 });
```

在这个测试中，我们模拟了浏览器处于离线状态，并访问了一个网页。然后，我们检查页面上是否显示了离线消息。如果没有，则测试失败。

以上是使用Cypress进行可靠性测试的几个示例。当然，还有很多其他的测试方式，可以根据具体情况进行选择。

如何进行可访问性测试

在cypress中进行可访问性测试是非常重要的，这有助于确保您的应用程序对所有用户都可访问，包括具有残疾或使用辅助技术的用户。Cypress提供了一些内置的工具来帮助您进行可访问性测试。

1. 安装和使用cypress-axe插件

cypress-axe是一个Cypress插件，可以帮助您检测Web应用程序中的可访问性问题。要使用cypress-axe，请执行以下步骤：

第一步：安装cypress-axe插件

您可以使用以下命令安装cypress-axe插件：

```
1 npm install --save-dev cypress-axe
```

第二步：导入cypress-axe插件

在cypress/support/index.js文件中导入cypress-axe插件：

```
1 import 'cypress-axe'
```

第三步：使用cypress-axe插件

您可以使用以下命令在您的测试中运行cypress-axe插件：

```
1 cy.injectAxe()
```

```
2 cy.checkA11y()
```

`cy.injectAxe()` 命令将axe-core注入到页面中。`cy.checkA11y()` 命令将使用注入的axe-core来检测页面中的可访问性问题。

2. 了解Axe-core

Axe-core是一个自动化可访问性测试引擎，它可以帮助您检测Web应用程序中的可访问性问题。Cypress-axe插件是基于Axe-core构建的。如果您想深入了解Axe-core，请查看以下链接：

- Axe-core官方网站：<https://github.com/dequelabs/axe-core>
- Axe-core文档：<https://dequeuniversity.com/rules/axe/4.3>

3. 检测可访问性问题

运行cypress-axe插件后，您将看到有关您的Web应用程序中的可访问性问题的报告。报告将指出问题的严重性级别以及如何解决它们。您可以使用该报告来识别和修复您的应用程序中的可访问性问题。

下面是一个示例测试，演示如何使用cypress-axe插件检测可访问性问题：

```
1 describe('Accessibility Test', () => {
2   beforeEach(() => {
3     cy.visit('/')
4     cy.injectAxe()
5   })
6
7   it('Has no detectable a11y violations on load', () => {
8     cy.checkA11y()
9   })
10 })
```

此测试将检测您的应用程序中的可访问性问题，并在控制台中输出问题的数量和详细信息。

cypress的优化和调优技巧

当使用 Cypress 进行测试时，你可能会遇到一些性能方面的问题，例如测试用例运行缓慢、测试中断、内存溢出等。这些问题都可以通过优化和调优 Cypress 测试来解决。下面是一些 Cypress 优化和调优的技巧：

1. 减少等待时间：

在测试用例中，你需要等待网页加载完成或元素出现在页面上。Cypress 提供了 `cy.wait()` 命令来实现等待时间。然而，等待时间过长会使测试运行速度变慢。因此，你可以通过调整等待时间来减少测试时间。可以使用 `defaultCommandTimeout` 来设置默认的命令等待时间，使用 `pageLoadTimeout` 来设置页面加载时间，使用 `requestTimeout` 来设置请求等待时间。

```
1 javascriptCopy code
2 // 设置默认命令等待时间为5秒Cypress.config('defaultCommandTimeout', 5000)
3
4 // 设置页面加载等待时间为10秒Cypress.config('pageLoadTimeout', 10000)
5
6 // 设置请求等待时间为10秒Cypress.config('requestTimeout', 10000)
```

2. 最小化测试数据：

在测试用例中，只需要使用最少的测试数据来测试你的应用程序。使用大量的测试数据会导致测试运行时间变慢。因此，可以考虑只使用必要的测试数据进行测试。

3. 选择合适的定位元素方式：

Cypress 提供了多种定位元素的方式，例如 `cy.get()`、`cy.contains()`、`cy.find()` 等。你可以选择最适合你的测试场景的方式来定位元素。比如，使用 `cy.get()` 来定位单个元素，使用 `cy.contains()` 来定位包含特定文本的元素，使用 `cy.find()` 来定位特定父元素下的子元素。

4. 最小化测试用例的依赖：

测试用例中的依赖关系可能会导致测试失败或测试运行缓慢。因此，你需要最小化测试用例的依赖关系。你可以使用 `cy.clock()` 命令来模拟时间，使用 `cy.stub()` 命令来模拟函数调用，以减少测试用例之间的依赖关系。

5. 使用 Cypress 的并行测试：

Cypress 支持并行测试，可以同时运行多个测试用例。使用并行测试可以显著提高测试速度。可以使用 Cypress Dashboard 或者第三方工具来实现并行测试。