[1 自动化测试 2](#_Toc204528787)

[1.1 测试框架 2](#_Toc204528788)

[1.1.1 Pytest框架 3](#_Toc204528789)

[1.1.1.1 Pytest原理 3](#_Toc204528790)

[1.1.1.2 Pytest基础 3](#_Toc204528791)

[1.1.1.3 Fixtures与conftest 6](#_Toc204528792)

[1.1.1.4 Marks 14](#_Toc204528793)

[1.1.1.5 数据驱动测试 21](#_Toc204528794)

[1.1.1.6 插件与集成 23](#_Toc204528795)

[1.1.2 Unittest框架 26](#_Toc204528796)

[1.1.2.1 基本用法 26](#_Toc204528797)

[1.1.2.2 断言方法 27](#_Toc204528798)

[1.1.2.3 测试套件（Test Suite） 27](#_Toc204528799)

[1.1.2.4 测试夹具（Test Fixtures） 28](#_Toc204528800)

[1.1.2.5 跳过测试（Skipping Tests） 28](#_Toc204528801)

[1.1.2.6 7. 运行测试 29](#_Toc204528802)

[1.1.2.7 对比Pytest 29](#_Toc204528803)

[1.2 接口自动化 29](#_Toc204528804)

[1.2.1 Requests基础-请求、响应 30](#_Toc204528805)

[1.2.2 Requests Cookie/Session处理方式 30](#_Toc204528806)

[1.2.2.1 Cookie 处理 30](#_Toc204528807)

[1.2.2.2 Session 处理 31](#_Toc204528808)

[1.2.3 Requests不同数据类型的处理方式 33](#_Toc204528809)

[1.2.3.1 表单数据（Form Data） 33](#_Toc204528810)

[1.2.3.2 JSON 数据 33](#_Toc204528811)

[1.2.3.3 URL 查询参数（Query Parameters） 34](#_Toc204528812)

[1.2.3.4 文件上传（Multipart Form Data） 34](#_Toc204528813)

[1.2.3.5 二进制数据（Binary Data） 35](#_Toc204528814)

[1.2.3.6 自定义请求头（Custom Headers） 35](#_Toc204528815)

[1.2.3.7 Cookies 处理 35](#_Toc204528816)

[1.2.3.8 超时设置（Timeout） 35](#_Toc204528817)

[1.2.3.9 重定向与流式传输（Redirect & Streaming） 36](#_Toc204528818)

[1.2.4 Requests多测试环境管理 36](#_Toc204528819)

[1.2.4.1 多测试环境管理概念与目的 36](#_Toc204528820)

[1.2.4.2 Requsests多测试环境管理的实现 37](#_Toc204528821)

[1.2.5 Requests二次封装 41](#_Toc204528822)

[1.2.5.1 二次封装的目的 41](#_Toc204528823)

[1.2.5.2 二次封装 requests 的基本结构 41](#_Toc204528824)

[1.2.5.3 二次封装的增强功能 42](#_Toc204528825)

[1.2.5.4 二次封装的好处 42](#_Toc204528826)

[1.2.6 接口自动化业务流程及流程封装 42](#_Toc204528827)

[1.3 web自动化 42](#_Toc204528828)

[1.3.1 现代浏览器基础 42](#_Toc204528829)

[1.3.1.1 浏览器启动 42](#_Toc204528830)

[1.3.1.2 浏览器结构与工作原理 43](#_Toc204528831)

[1.3.1.3 浏览器的功能 44](#_Toc204528832)

[1.3.1.4 页面资源获取与渲染 44](#_Toc204528833)

[1.3.2 Selenium 45](#_Toc204528834)

[1.3.2.1 Selenium原理 45](#_Toc204528835)

[1.3.2.2 Webdriver 46](#_Toc204528836)

[1.3.2.3 元素定位与操作 51](#_Toc204528837)

[1.3.2.4 ActionChains 53](#_Toc204528838)

[1.3.2.5 等待机制 55](#_Toc204528839)

[1.3.2.6 文件操作 60](#_Toc204528840)

[1.3.2.7 Cookie与会话管理 61](#_Toc204528841)

[1.3.2.8 iframe与弹窗处理 63](#_Toc204528842)

[1.3.3 Playwright 65](#_Toc204528843)

[1.3.3.1 Playwright原理 66](#_Toc204528844)

[1.3.3.2 层次资源管理 66](#_Toc204528845)

[1.3.3.3 元素定位于操作 74](#_Toc204528846)

[1.3.3.4 等待机制 84](#_Toc204528847)

[1.3.3.5 文件操作 86](#_Toc204528848)

[1.3.3.6 异步操作 88](#_Toc204528849)

[1.3.3.7 Playwright的API操作 91](#_Toc204528850)

[1.3.3.8 Playwright Test框架 91](#_Toc204528851)

[1.4 APP自动化 93](#_Toc204528852)

[1.4.1 Appium环境搭建与ADB命令 93](#_Toc204528853)

[1.4.2 Appium启动参数详解及元素定位 93](#_Toc204528854)

[1.4.3 Appium常用方法以及键盘鼠标手势事件 93](#_Toc204528855)

[1.4.4 Appium等待方式与二次封装 93](#_Toc204528856)

[1.4.5 Appium之H5及Toast处理 93](#_Toc204528857)

[1.5 自动化框架 93](#_Toc204528858)

# 自动化测试

## 测试框架

测试框架是为了提高测试过程的效率、可维护性和可扩展性而设计的一套结构化的工具和方法集合。它提供了一个标准化的方式来组织、执行、报告和管理测试。测试框架的目标是自动化测试、提高测试的重复性、减少人为错误，并提供清晰的测试结果反馈。

测试框架的核心组成部分包括：

* **测试用例**：用于验证软件功能是否符合预期，包括功能性测试（如验证功能）和非功能性测试（如性能、负载等）。
* **测试执行**：框架按顺序执行测试用例，并报告执行结果。
* **断言机制**：用于验证实际结果与预期结果的一致性，是框架的重要组成部分。
* **日志和报告**：记录执行过程中的日志，生成测试报告，帮助定位问题。
* **数据驱动**：通过自动化运行不同数据集的测试用例，验证系统在不同输入下的表现。
* **测试数据管理**：管理测试数据的加载、清理和恢复，确保测试环境的一致性。
* **测试环境管理**：自动化设置和恢复测试环境，保证不同环境下测试的一致性。

### Pytest框架

pytest 是一个非常强大的测试框架，能够帮助开发者高效地编写、组织和执行测试。通过灵活的功能，如 fixture、参数化、标记、并行执行等，pytest 能够应对各种测试需求。它简单的语法和丰富的功能使得它不仅适用于初学者，也适合复杂项目的自动化测试。

参考[pytest框架-详解（学习pytest框架这一篇就够了）-CSDN博客](https://blog.csdn.net/kkkkk19980517/article/details/139065687?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522b1043682f7c7602aaeccec2a7c6cab16%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=b1043682f7c7602aaeccec2a7c6cab16&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-139065687-null-null.142%5ev102%5econtrol&utm_term=pytest&spm=1018.2226.3001.4187)

官方文档[pytest documentation](https://docs.pytest.org/en/stable/)

#### Pytest原理

**pytest** 就像一个“自动化测试操作系统”，  
它把整个测试生命周期**分解为多个阶段**，  
并在每个阶段通过“钩子函数”开放接口（这些钩子就是“事件插槽”），  
你只需要按照指定的函数名（规范）去写，  
就能在 **不破坏 pytest 主流程的前提下**，  
注入你自己的逻辑，实现灵活的扩展和定制

#### Pytest基础

##### 基本组成

Pytest框架主要由以下几个核心部分组成：

1. **测试用例**：核心功能，通过编写函数或类测试软件功能，使用assert进行断言验证。
2. **测试执行**：负责执行测试用例并报告结果，可以通过命令行运行测试。
3. **断言机制**：使用assert语句验证测试结果，自动判断测试通过与否。
4. **测试夹具（Fixtures）**：用于设置和清理测试环境，可以在测试前后执行预处理和后处理操作。
5. **测试标记（Markers）**：用于分类和控制测试（如跳过测试、预期失败、参数化等）。
6. **命令行选项**：提供多种选项控制测试执行，例如设置最大失败次数、生成报告等。
7. **测试报告**：生成各种格式的测试报告，如HTML或JUnit格式，帮助分析测试结果。
8. **插件系统**：支持通过插件扩展功能，如并行测试、代码覆盖率等。
9. **钩子（Hooks）**：允许在测试流程的不同阶段插入自定义操作，扩展框架行为。
10. **参数化（Parameterization）**：通过@pytest.mark.parametrize进行测试用例的数据驱动，运行多组数据。

##### 框架约束

Pytest 对测试文件名、目录结构及代码组织有明确的约定，以确保测试用例的自动发现和执行。

###### 测试文件

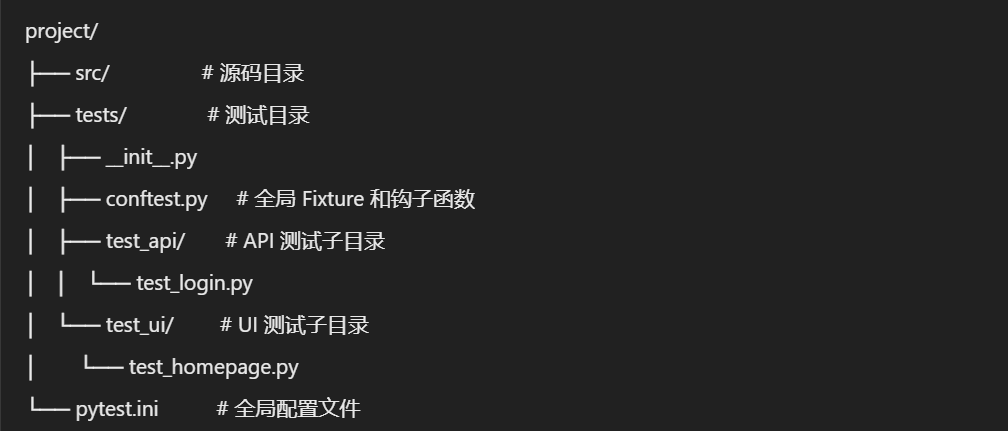
* **测试文件​​：**文件名需以 test\_ 开头（如 test\_login.py）或以 \_test.py 结尾（如 login\_test.py）。
* ​​**测试函数/方法​​：**函数名或类中的方法名需以 test\_ 开头（如 test\_add()）。
* ​​**测试类​​：**类名需以 Test 开头（如 TestCalculator），且类中不能包含 \_\_init\_\_ 构造函数，否则可能导致用例无法被收集

若文件名不符合默认规则（如 demo.py），Pytest 不会自动识别为测试文件，需通过命令行指定文件路径运行。

###### 目录结构

下面是一个推荐的Pytest目录结构：

* **根目录​​：**通常将测试代码与源码分离，推荐使用 tests 目录存放所有测试文件，源码放在 src 目录。
* ​​**子目录​​：**可按模块划分子目录（如 tests/api/、tests/ui/），每个子目录需包含 \_\_init\_\_.py 文件（空文件即可）以支持多层级用例管理。
* ​​**配置文件​​：**根目录下可放置 pytest.ini 定义全局参数，conftest.py 存放共享 Fixture。



##### Pytest.ini测试配置与运行

pytest.ini是Pytest的配置文件，用于设置Pytest的全局配置选项。允许在一个地方集中管理测试框架的行为、日志记录、插件配置等，避免每次执行测试时都使用命令行参数进行设置。

常见配置项

|  |  |  |
| --- | --- | --- |
| **配置项​​** | **​​说明​​** | **​​示例​​** |
| ​**​addopts​**​ | 添加默认命令行参数，简化测试执行命令 | addopts = -v -s --html=report.html --reruns=2 |
| ​**​testpaths​**​ | 指定测试用例搜索目录（多个目录用空格分隔） | testpaths = tests/api tests/db |
| ​**​python\_files​**​ | 定义测试文件命名规则（支持通配符） | python\_files = test\_\*.py \*\_check.py |
| ​**​python\_classes​**​ | 定义测试类命名规则（需以指定前缀开头） | python\_classes = Test\* Check\* |
| ​**​python\_functions​**​ | 定义测试函数/方法命名规则 | python\_functions = test\_\* check\_\* |
| ​**​markers​**​ | 注册自定义标记，用于分类测试用例（如冒烟测试、性能测试） | markers = smoke: Quick smoke tests; slow: Long-running tests |
| ​**​log\_cli​**​ | 启用实时日志输出（默认不显示） | log\_cli = True |
| ​**​log\_cli\_level​**​ | 设置实时日志输出级别（如 DEBUG、INFO） | log\_cli\_level = INFO |
| ​**​log\_cli\_format​**​ | 自定义实时日志格式（使用 logging 模块格式） | log\_cli\_format = %(asctime)s [%(levelname)s] %(message)s |
| ​**​xfail\_strict​**​ | 强制标记为 @pytest.mark.xfail 的测试在意外通过时视为失败 | xfail\_strict = True |
| ​**​norecursedirs​**​ | 排除指定目录（避免扫描无关目录如 venv、build） | norecursedirs = venv build logs |
| ​**​filterwarnings​**​ | 过滤或忽略特定警告（如忽略弃用警告） | filterwarnings = ignore::DeprecationWarning |
| ​**​junit\_suite\_name​**​ | 生成 JUnit XML 报告时自定义测试套件名称 | junit\_suite\_name = API Test Suite |
| ​**​junit\_logging​**​ | 控制 JUnit 报告中日志输出方式（如 system-out 或 no） | junit\_logging = system-out |
| ​**​minversion​**​ | 指定 pytest 最低版本要求（低于此版本会报错） | minversion = 7.0 |

Pytest一般通过命令行运行，命令行运行的配置选项都可以通过ini文件进行配置，一般推荐是通过ini配置好运行选项，然后通过基本命令运行，或者IDE会提供一些更方便的启动运行方式，例如VS code可以开启测试模式通过点击函数、项目进行测试。

**命令行基本命令：**

1. **运行所有测试 直接输入 pytest 命令，Pytest 会自动查找当前目录下所有符合命名规则的测试文件并执行。**

**pytest**

1. **运行特定的测试文件 指定某个文件来执行其中的测试用例。**

**pytest test\_example.py**

1. **运行特定目录的测试 指定一个目录来运行该目录下的所有测试文件。**

**pytest tests/**

1. **运行指定的测试类 使用 :: 符号来指定某个类进行执行。**

**pytest test\_example.py::TestExample**

1. **运行指定的测试函数 使用 :: 符号来指定某个函数进行执行。**

**pytest test\_example.py::test\_addition**

##### 用例断言与报告

Pytest 提供了一个强大的断言机制，扩展和增强了 Python 原生 assert 语句，使得测试结果更加易于理解和调试。Pytest 的断言机制相对于 Python 原生 assert 做了许多增强和改进：

1. **详细的失败信息**：自动展示失败的断言表达式、预期值、实际值等。
2. **自定义错误消息**：可以为失败的断言添加有意义的错误消息。
3. **异常断言**：使用 pytest.raises() 来捕获和验证特定异常。
4. **浮动数值比较**：通过 pytest.approx() 进行浮动数值的近似比较。
5. **容器断言**：支持检查列表、字典、字符串等容器中是否包含特定元素或符合某个条件

**常见的断言类型：**

* **基本断言：判断表达式是否为真，如果为假则测试失败。**

def test\_addition():

    assert 1 + 1 == 2  # 断言通过

* **比较断言：检查值是否等于预期。**

def test\_string\_equality():

    assert "hello" == "hello"  # 断言通过

* **包含断言：判断容器类型数据（如列表、字典、字符串）是否包含某个元素。**

def test\_in\_list():

    assert 5 in [1, 2, 3, 4, 5]  # 断言通过

* **异常断言：验证代码是否抛出特定的异常。**

import pytest

def test\_zero\_division():

    with pytest.raises(ZeroDivisionError):

        1 / 0  # 期待抛出 ZeroDivisionError

* **浮动数据比较：通过 pytest.approx() 进行浮动数据的近似比较。**

import pytest

def test\_float\_addition():

    assert 0.1 + 0.2 == pytest.approx(0.3, rel=1e-9)  # 允许误差范围比较

Pytest通过增强断言，使得错误的捕获更加强大，让用例报告有着更详尽的错误信息和报告。运行后会根据运行配置、插件配置自动生成测试报告。

#### Fixtures与conftest

**Fixtures** 是 Pytest 中用于为测试提供共享资源、状态或设置的功能。它们可以被用来初始化环境、准备测试数据、清理资源等。Fixtures 提供了一种灵活的方式来为多个测试用例提供所需的前置条件，避免在每个测试中重复相同的初始化代码。

**conftest.py** 是一个特殊的配置文件，用于存放共享的 Fixtures、钩子函数以及其他测试配置。它通常用于项目级的设置，避免在每个测试文件中重复定义相同的 Fixtures。

##### 基本功能与结构

**定义Fixture**：使用 @pytest.fixture 装饰器来定义一个 Fixture。Fixture 是一个普通的 Python 函数，它的返回值会被作为参数传递给测试用例。

**基本形式：**

@pytest.fixture(scope = "function",params=None,autouse=False,ids=None,name=None)

**常见Fixtures函数的写法：**

@pytest.fixture

def api\_client():

    with ApiClient() as client:

        yield client

使用夹具时一般会同yield结合使用，以实现前置条件的实现和后置资源的清理。

##### 嵌套

Fixtures可以实现嵌套的前置操作管理，以减少代码重复、模块化测试前置条件、提高灵活性。例如前置条件A、B都依赖C，而函数X、Y分别依赖A、B时，就可以使用嵌套的fixtures来模块化这些前置条件。

**嵌套的执行顺序：**

* pytest 会根据 fixture 的依赖关系来决定执行的顺序。它会先执行依赖的 fixture，然后执行依赖它的 fixture。
* 每个 fixture 的返回值会被自动传递到需要它的测试用例中。
* 嵌套 fixture 的执行顺序是从内到外（即外层依赖内层 fixture）。

**多嵌套示例**

import pytest

@pytest.fixture

def config():

    return {"database": "localhost", "user": "admin"}

@pytest.fixture

def database\_connection(config):

    print(f"Connecting to {config['database']} with user {config['user']}")

    return "DB connection"

@pytest.fixture

def user\_data(database\_connection):

    print("Setting up user data")

    return {"user": "test\_user", "connection": database\_connection}

def test\_user(user\_data):

    assert user\_data["user"] == "test\_user"

    assert user\_data["connection"] == "DB connection"

在这个例子中，user\_data 依赖于 database\_connection，而 database\_connection 又依赖于 config。测试用例 test\_user 需要 user\_data，而 user\_data 又需要其他 fixture。

##### 多注入

**多注入**（Multiple Fixture Injection）指的是在一个测试函数中注入多个 fixture。这是 pytest 强大的功能之一，可以帮助你更灵活地管理测试环境，避免重复代码，同时保持测试代码的简洁和模块化。

**多注入示例：**

import pytest

@pytest.fixture

def fixture\_a():

    return "Fixture A"

@pytest.fixture

def fixture\_b():

    return "Fixture B"

def test\_multiple\_injection(fixture\_a, fixture\_b):

    assert fixture\_a == "Fixture A"

    assert fixture\_b == "Fixture B"

**优势：**

1. **简化代码：**多个 fixture 可以在测试函数中被注入并传递，避免重复代码。
2. **提高灵活性：**你可以通过组合不同的 fixture 来创建复杂的测试环境，同时保持模块化。
3. **支持多种组合方式：**通过多注入，支持更灵活的测试场景，比如结合参数化、作用域等。
4. **自动化管理：**结合 autouse=True 等功能，可以自动管理多个 fixture，减少冗余代码。

##### Scope

fixture 的 scope 参数用于定义 fixture 的生命周期（即创建和销毁的时机）。通过设置 scope，你可以控制每个 fixture 在测试过程中活跃的范围，避免重复初始化资源，节省测试运行时间，并确保测试环境的正确性。

控制范围的排序为：session > module > class > function

|  |  |
| --- | --- |
| **取值** | **范围说明** |
| function | 函数级 每一个函数或方法都会调用 |
| class | 类级 每一个.py文件测试类内调用一次 |
| module | 模块级 每一个.py文件调用一次 |
| session | 会话级 整个测试会话只需要运行一次，会话内所有方法及类，模块都共享这个方法 |

* **scope的作用**

1. **减少资源初始化：**使用较长的作用域（如 module 或 session）避免重复创建和销毁资源。
2. **控制资源生命周期：**根据需求控制资源的共享范围，避免不必要的资源浪费。
3. **提高效率：**共享资源（如数据库连接）可以提升测试运行效率，尤其是大规模测试时。

* **注意事项**

1. **作用域顺序：**作用域较小的 fixture 会在作用域较大的 fixture 之前创建和销毁，例如 function 级别的 fixture 在 module 级别的 fixture 之前创建。
2. **依赖关系：**pytest 会根据 Fixture 的依赖关系和作用域确定执行顺序，确保小作用域的 fixture 先于大作用域的 fixture 执行。
3. **资源共享：**避免在长作用域的 fixture 中使用不可重复初始化的资源（如数据库连接），确保这些资源能正确共享和管理。

* **作用域与依赖关系**

当多个 fixture 存在依赖关系时，pytest 会遵循以下规则：

* 作用域较小的 fixture 必须先创建，且在作用域较大的fixture之前销毁。
* 如果一个较小作用域的 fixture 依赖于一个较大作用域的 fixture，pytest 会确保较大作用域的 fixture 在较小作用域的 fixture 创建之前就已经创建好。

##### Autouse

**autouse** 参数用于控制 fixture 是否**自动**注入到测试函数中，而无需在测试函数的参数列表中显式声明。它是一个布尔值参数，默认为 False。当 autouse=True 时，pytest 会自动将该 fixture 注入到所有测试函数中，而不需要手动传递或声明。

* **示例：**

import pytest

@pytest.fixture(autouse=True)

def setup\_db():

    print("\nSetting up the database")

    return "Database Connection"

def test\_1(setup\_db):

    print("Running test\_1")

def test\_2(setup\_db):

    print("Running test\_2")

**输出：**

Setting up the database

Running test\_1

Setting up the database

Running test\_2

* **作用范围：**

当autouse配合scope使用，可以实现控制自动注入的范围

* **模块级别**：autouse=True 和 scope="module" 结合时，整个模块的测试函数会自动使用该 Fixture。
* **类级别**：autouse=True 和 scope="class" 结合时，整个测试类的测试方法会自动使用该 Fixture。
* **函数级别**：autouse=True 和 scope="function" 结合时，每个测试函数都会自动使用该 Fixture
* **适用场景：**

1. **全局初始化和清理**

自动初始化每个测试需要的资源（如数据库连接、网络服务），避免每个测试函数中显式声明。

1. **日志记录、性能监控等跨测试功能**

在每个测试执行前自动进行操作，如记录日志、启动监控等，无需显式调用。

1. **资源共享和清理**

自动启动和清理跨测试共享的资源，例如 Web 服务、数据库连接等。

1. **跨测试类或模块共享资源**

在整个模块或类中自动注入共享资源（如全局配置），无需每个测试显式声明。

##### params

params 参数是 pytest 中 fixture 的一个强大特性，它允许你为同一个 fixture 提供多个不同的参数值，从而让一个测试函数可以多次运行，每次使用不同的参数，避免重复编写多个相似的测试。每个参数值都会触发一个新的测试用例。

* **基本用法：**

在 fixture 中使用 params 参数时，你可以传递一个列表、元组或其他可迭代对象。每个元素都会作为参数传递给测试函数，然后 pytest 会自动执行测试，分别为每个参数值运行一次。

import pytest

@pytest.fixture(params=[(1, 2), (3, 4), (5, 6)])

def add\_numbers(request):

    return sum(request.param)

def test\_addition(add\_numbers):

    assert add\_numbers in [3, 7, 11]

例中，params 参数传递了一个包含三组数字的列表，pytest 会对每一组数字执行一次测试：第一次add\_numbers函数返回1+2=3，第二次7，第三次11，也就是完成了三次测试用例的执行！

* **params 与 request.param 的关系**

params 会通过 request 对象传递给 fixture，request.param 是当前正在使用的参数值。你可以通过 request.param 访问每次测试执行时传递的参数。

* **多参数组合**

@pytest.fixture(params=[(1, 2, 3), (4, 5, 9), (7, 8, 15)])

def add\_numbers(request):

    return request.param[0] + request.param[1], request.param[2]

def test\_addition(add\_numbers):

    assert add\_numbers[0] == add\_numbers[1]

这里 params 传递了一个元组，每个元组有两个加数和期望结果，add\_numbers 会分别返回加法结果和期望结果，最终检查它们是否匹配。

##### Ids

**ids** 参数是 pytest 中用于给 fixture 中的每个参数值指定一个易读的名称。它通常用于参数化测试（如使用 params 参数时），为每次测试运行指定一个清晰的名称，以便在测试输出中区分和识别不同的测试用例。

* **基本用法**

ids可以是一个列表，它的长度必须与 params 中的元素数目相同，每个ids元素都会对应一个 params中的元素。如果不提供 ids，pytest 会自动生成名称，通常是参数的字符串表示形式。

* **示例 1：使用 ids 参数**

假设你有一个测试，它需要对不同的加法输入执行验证：

import pytest

@pytest.fixture(params=[(1, 2), (3, 4), (5, 6)], ids=["test\_case\_1", "test\_case\_2", "test\_case\_3"])

def add\_numbers(request):

    return sum(request.param)

def test\_addition(add\_numbers):

    assert add\_numbers in [3, 7, 11]

* params 提供了三个输入元组：(1, 2)、(3, 4)、(5, 6)。
* ids 给每个测试用例提供了对应的可读名称：test\_case\_1、test\_case\_2、test\_case\_3。

**在 pytest 输出中，你将看到类似 test\_case\_1、test\_case\_2、test\_case\_3 的名称，而不是 (1, 2)、(3, 4)、(5, 6) 这样的元组值。这样可以让你更容易地理解每个测试用例的目的。**

* **示例 2：使用 ids 生成动态名称**

你还可以动态生成 ids，例如根据输入值生成易读的名称：

import pytest

def generate\_ids(param):

    return f"add\_{param[0]}\_and\_{param[1]}"

@pytest.fixture(params=[(1, 2), (3, 4), (5, 6)], ids=lambda param: generate\_ids(param))

def add\_numbers(request):

    return sum(request.param)

def test\_addition(add\_numbers):

    assert add\_numbers in [3, 7, 11]

在这里，generate\_ids 函数生成了基于输入的自定义名称：

* 对于 (1, 2)，生成的 ID 是 add\_1\_and\_2；
* 对于 (3, 4)，生成的 ID 是 add\_3\_and\_4；
* 对于 (5, 6)，生成的 ID 是 add\_5\_and\_6。
* **示例 3：部分指定 ids**

如果你只想为某些参数值指定 ID，而其他参数使用默认 ID，可以部分指定 ids 参数。

import pytest

@pytest.fixture(params=[(1, 2), (3, 4), (5, 6)], ids=["sum\_1\_and\_2", None, "sum\_5\_and\_6"])

def add\_numbers(request):

    return sum(request.param)

def test\_addition(add\_numbers):

    assert add\_numbers in [3, 7, 11]

在这个例子中，第二个参数 (3, 4) 没有指定 ID，因此 pytest 会自动为它生成 ID（通常是 (3, 4) 的字符串表示）。

##### name

通过name参数，你可以为 fixture 自定义名称，从而让测试用例中更易于识别和理解。如果使用了name,那只能将name传如，函数名不再生效。

* **基本用法：**

通过name参数重命名fixture函数，十七通过重命名的函数名调用

* **示例 1：命名 fixture**

import pytest

@pytest.fixture(name="database\_connection")

def setup\_db():

    print("\nConnecting to the database...")

    yield "DB Connection"

    print("\nDisconnecting from the database...")

def test\_insert(database\_connection):

    print(f"Using {database\_connection} for insert test.")

def test\_delete(database\_connection):

    print(f"Using {database\_connection} for delete test.")

* name="database\_connection"：这里 database\_connection 是 fixture 的名称，表示在每个测试函数中都可以通过这个名字来访问数据库连接。
* 测试函数 test\_insert 和 test\_delete 都会自动接收到名为 database\_connection 的 fixture。
* **示例 2：为多个 fixture 使用相同名称**

如果你希望通过同一个名字共享不同的 fixture，可以将 name 应用到多个 fixture。

import pytest

@pytest.fixture(name="shared\_resource")

def resource\_one():

    return "Resource One"

@pytest.fixture(name="shared\_resource")

def resource\_two():

    return "Resource Two"

def test\_resource(shared\_resource):

    print(f"Using {shared\_resource}")

两个不同的 fixture 函数有相同的 name（shared\_resource），但pytest 会选择最后定义的 fixture，作为 test\_resource 测试用例的输入。**这会导致 test\_resource 总是接收到 "Resource Two"。**

* **示例 3：动态命名 fixture**

可以在一些复杂的场景中使用动态命名，通过一个工厂函数来决定 fixture 的名称。

import pytest

def generate\_fixture\_name(param):

    return f"fixture\_{param}"

@pytest.fixture(name=lambda param: generate\_fixture\_name(param))

def dynamic\_fixture(request):

    return f"Data for {request.param}"

@pytest.mark.parametrize("dynamic\_fixture", [1, 2, 3], indirect=True)

def test\_dynamic\_fixture(dynamic\_fixture):

    print(f"Using {dynamic\_fixture}")

在这个例子中，name 通过一个 lambda 函数动态生成名称，根据 param 参数生成不同的 fixture 名称。

##### Confest管理Fixtures

pytest 中，**conftest.py** 文件是一个特殊的配置文件，用于集中管理 fixture 和插件。它让你能够共享 fixture，配置钩子函数，和配置测试的行为，而无需将这些内容写入每个测试文件中。conftest.py 使得大型项目的测试结构更加清晰和模块化。

* conftest.py 用于集中管理测试项目中的 fixture 和配置，避免了在每个测试文件中重复定义相同的 fixture。
* 在 conftest.py 中定义的 fixture 可以通过 scope 参数控制其生命周期，并在项目中的任何测试文件中使用。
* 除了 fixture 外，conftest.py 还支持添加自定义命令行选项、钩子函数等，用于扩展 pytest 的功能。
* conftest.py 可以放在项目的根目录或测试目录中，支持层次化配置。
* **conftest.py 的作用域（Scope）**

在 conftest.py 中定义的 fixture 仍然可以像其他 fixture 一样设置 scope 参数来控制其生命周期。

* **conftest.py 中的其他配置**

除了定义 fixture 之外，conftest.py 还可以用于配置 pytest 的其他行为，包括钩子函数、命令行选项等

* **conftest.py 的层次结构**

conftest.py 文件可以存在于多个层级：

* **根目录：**如果放在项目的根目录，里面定义的 fixture 可以在所有测试文件中使用。
* **子目录：**如果放在某个特定测试子目录中，里面定义的 fixture 仅对该子目录及其子目录中的测试文件有效。pytest 会从根目录到当前目录逐层查找 conftest.py 文件。

这种层次结构让你可以有选择性地共享 fixture。例如，你可以在根目录的 conftest.py 中定义一些通用的 fixture，并在子目录中定义与该子目录相关的特定 fixture。

#### Marks

在 pytest 中，marks 是用来标记测试用例的一种方式，能够为测试用例添加额外的信息或进行分类。通过标记，pytest 允许你根据特定的标记来选择运行特定的测试，或执行一些自定义操作。

##### pytest.mark.parametrize

pytest.mark.parametrize 是 pytest 中的一个非常强大的功能，用于参数化测试。通过它，你可以一次性为一个测试函数提供多个输入值，pytest 会自动地使用这些输入值多次执行该测试函数，从而提高测试的覆盖范围和效率。

###### 基本结构

**pytest.mark.parametrize 的基本结构是：**

@pytest.mark.parametrize(argnames, argvalues)

* **argnames：**参数的名称，通常是一个字符串（如果是多个参数，可以是一个由逗号分隔的字符串）。这些参数会被传递给测试函数。
* **argvalues：**为每个测试用例提供的参数值。它可以是一个列表、元组或其他可迭代对象，包含多组参数。

**示例：**

**假设你有一个简单的加法函数，想对它进行参数化测试，确保它对多个输入值正确工作。**

import pytest

def add(x, y):

    return x + y

@pytest.mark.parametrize("x, y, expected", [(1, 2, 3), (3, 4, 7), (5, 5, 10)])

def test\_add(x, y, expected):

    assert add(x, y) == expected

**在上面的例子中：**

* x, y, 和 expected 是测试函数 test\_add 的参数名。
* [(1, 2, 3), (3, 4, 7), (5, 5, 10)] 是每一组输入数据及预期结果。
* pytest 会为每一组参数运行一次 test\_add 测试函数。

**运行结果：**

test\_example.py::test\_add[1-2-3] PASSED

test\_example.py::test\_add[3-4-7] PASSED

test\_example.py::test\_add[5-5-10] PASSED

**这表示 test\_add 测试函数会执行 3 次，分别使用不同的输入和预期结果。**

###### 支持多个参数

如果你想为一个测试函数提供多个参数，可以通过逗号分隔参数名。例如：

import pytest

def multiply(x, y):

    return x \* y

@pytest.mark.parametrize("x, y", [(1, 2), (3, 4), (5, 6)])

def test\_multiply(x, y):

    assert multiply(x, y) == x \* y

在这个例子中，x 和 y 是测试函数 test\_multiply 的两个参数，pytest 会针对每一对 (x, y) 的值运行一次 test\_multiply。

###### 参数化和组合

pytest.mark.parametrize 支持多个参数进行组合测试，pytest 会为所有组合执行测试。这对于需要多组数据组合来执行测试非常有用。

**示例：**

import pytest

@pytest.mark.parametrize("a", [1, 2])

@pytest.mark.parametrize("b", [3, 4])

def test\_combinations(a, b):

    print(f"Testing combination: a={a}, b={b}")

    assert a + b > 0

在这个例子中，a 和 b 都是参数化的，并且每个参数值会和另一个参数的每个值进行组合，从而生成所有可能的组合。pytest 会运行如下的组合：

**(1, 3)、(1, 4)、(2, 3)、(2, 4)**

###### 使用 ids 参数定制测试名称

类似于fixtures，pytest.mark.parametrize 支持通过 ids 参数来定制每个测试的名称（测试标识）。这可以帮助你更清晰地看到每个测试用例的具体内容，尤其在生成测试报告时。

**示例：**

import pytest

@pytest.mark.parametrize("x, y, expected", [(1, 2, 3), (3, 4, 7), (5, 5, 10)], ids=["test\_case\_1", "test\_case\_2", "test\_case\_3"])

def test\_add(x, y, expected):

    assert add(x, y) == expected

这里我们为每一组参数设置了自定义的测试名称。

###### 6. 使用 request 访问参数

还可以通过 request 来访问参数化的数据，这通常用于更复杂的用例或调试。

**示例：**

import pytest

@pytest.mark.parametrize("x", [1, 2, 3])

def test\_dynamic\_param(request, x):

    print(f"Testing with: {request.param}")

    assert x > 0

###### pytest.mark.parametrize 和 fixtures参数params

pytest.mark.parametrize 和 fixtures 的 params 都用于参数化测试pytest.mark.parametrize 主要用于测试数据的参数化而fixtures 的 params 主要用于测试夹具（资源）的参数化。

* **pytest.mark.parametrize：**
* 用于直接给测试函数提供多个参数，测试不同的输入组合。
* 每个参数化的组合都会运行一次测试函数。
* 适用于没有复杂资源依赖的场景。

**示例：**多个数据集下测试一个函数的不同输入输出。

* **fixtures 的 params：**
* 用于参数化测试夹具，提供多个不同的资源或数据集。
* 每次使用夹具时，pytest 会为每组参数化的数据初始化资源并传递给测试函数。
* 适用于需要共享资源或环境设置的复杂场景。

**示例：**多个数据库连接、配置文件等资源传递给测试函数。

##### pytest.mark.skip与skipif

pytest.mark.skip 和 pytest.mark.skipif 是 pytest 提供的两个标记（marks），用于跳过某些测试。它们可以根据不同的条件来决定是否跳过某个测试，常用于一些特定场景，如条件不满足时跳过测试，或者在某些环境下测试无法执行时跳过。

* **pytest.mark.skip** 用于无条件跳过测试。
* **pytest.mark.skipif** 用于基于特定条件跳过测试，适合于环境依赖、平台依赖的场景。

###### pytest.mark.skip

pytest.mark.skip 用于无条件跳过某个测试。被标记为 skip 的测试将不会执行，直接标记为跳过。

* **用法：**

import pytest

@pytest.mark.skip(reason="This test is skipped")

def test\_not\_implemented():

    assert 1 == 2

reason：跳过测试的原因，可以是一个字符串，用来解释为什么跳过此测试。

输出：**test\_example.py::test\_not\_implemented SKIPPED [0%]**

* **特点：**
* **无条件跳过：**每次运行测试时，都会跳过这个测试。
* **适用场景：**用于跳过当前不需要运行的测试，如某个功能未实现或需要排除某些无法运行的测试。

###### 2. pytest.mark.skipif

pytest.mark.skipif 用于根据条件跳过测试。通过传入一个表达式作为条件，只有当条件为 True 时，测试才会被跳过。

* **用法：**

import pytest

import sys

@pytest.mark.skipif(sys.platform != "linux", reason="Test only on Linux")

def test\_only\_on\_linux():

    assert 1 == 1

* 条件：第一个参数是一个条件表达式，只有条件为 True 时，测试才会被跳过。
* reason：与 skip 一样，可以为跳过的测试提供一个理由，帮助理解为什么跳过。

**输出：**

* 如果条件满足，跳过测试的输出会类似：

**test\_example.py::test\_only\_on\_linux SKIPPED [0%]**

* 如果条件不满足，测试会执行：

**test\_example.py::test\_only\_on\_linux PASSED [100%]**

* **特点：**
* **条件跳过**：只有当条件为 True 时，才跳过测试。否则测试会正常执行。
* **适用场景：**当测试的环境或条件特定时（如特定平台、特定库版本），可以跳过不符合条件的测试。

###### 常见应用场景：

* 跳过未实现的功能：
* 平台相关的跳过（例如，只在 Linux 上运行的测试）：
* 跳过有特定依赖的测试（例如某些外部库或硬件资源不可用时）：

##### pytest.mark.xfail与xpass

pytest.mark.xfail 和 pytest.mark.xpass 是 pytest 中的两个标记（marks），其中的x表示expected，**用于处理预期会失败的测试**，特别是在测试中出现了已知的问题或缺陷时。这些标记帮助在测试报告中明确地指出哪些测试是预期失败的，并允许在报告中跟踪这些测试的状态。

* **pytest.mark.xfail：**标记一个预期会失败的测试，失败时报告为“预期失败”（XPASS），不会影响整体结果。
* **pytest.mark.xpass：**标记一个本应失败的测试意外通过，会被报告为“意外通过”（XPASS），并发出警告。

###### pytest.mark.xfail

pytest.mark.xfail（“expected fail”）标记用于标记那些**已知会失败**的测试。当被标记的测试失败时，pytest 会记录为“预期失败”并跳过该测试，但不会影响测试的整体结果（测试依然会被视为通过）。通常用于跟踪已知缺陷、未修复的问题或尚未实现的功能。

* **用法：**

import pytest

@pytest.mark.xfail(reason="Known bug, fixed in future version")

def test\_known\_bug():

    assert 1 == 2  # 这将失败

* **condition (默认为 None)：**
* 用于指定 xfail 标记的条件。当 condition 为 True 时，xfail 标记生效，表示测试会预期失败。
* 你可以为 condition 提供一个表达式或一个布尔值。如果是 True，测试将被视为预期失败。
* **reason (默认为 None)：**
* 提供对为什么预期测试会失败的描述信息。
* 这是一个字符串，通常用于解释为何该测试被标记为预期失败。
* **raises (默认为 None)：**
* 你可以指定一个异常类，如果测试抛出了指定的异常，则视为预期失败。
* 比如，可以指定某个测试预期会抛出 ValueError，如果该异常抛出，则测试视为预期失败。
* **run (默认为 True)：**
* 用来控制测试是否实际运行。当 run=False 时，即使测试被标记为 xfail，它也不会执行。
* 如果你希望测试仅用于标记，而不执行某些代码，可以使用 run=False。
* **strict (默认为 False)：**
* 如果为 True，当测试通过时，它会被视为失败。即测试的实际通过不符合预期时会触发失败。
* 默认为 False，测试通过不会导致标记为 xfail 的测试失败。

当 xfail 测试失败时，pytest 会显示如下输出：

**test\_example.py::test\_known\_bug xfailed [100%]**

* **特点：**
* **预期失败：**xfail 标记的测试被视为预期会失败的测试。
* **失败不会导致失败：**当测试失败时，pytest 认为这并不是一个错误，因此测试结果会被标记为“通过”，但会标记为“预期失败”（XPASS）。
* **跳过修复**：适用于那些已知的 bug 或待修复的问题，不影响 CI 或报告中的通过率。

###### pytest.mark.xpass

pytest.mark.xpass（“expected pass”）标记用于标记那些 预期应该失败但意外通过 的测试。如果某个 xfail 标记的测试意外地通过了（即，实际结果与预期不符），则 pytest 会将其视为 “意外通过”（XPASS）。这种情况通常**表示问题已经被修复，或者存在不应通过的情况**。

* **用法：**

import pytest

@pytest.mark.xfail(reason="Known bug, fixed in future version")

def test\_bug\_fixed():

    assert 1 == 1  # 这将通过，但应该预期为失败

当 xfail 测试意外通过时，pytest 会显示如下输出：

**test\_example.py::test\_bug\_fixed xpassed [100%]**

* **特点：**
* **意外通过**：当 xfail 测试的实际结果是通过时，pytest 会将其标记为 XPASS，即“预期失败，实际通过”。
* **警告**：如果 xfail 测试意外通过，pytest 会在报告中标记为警告，提醒开发者可能存在已修复的 bug。

###### 使用场景

* **预期失败的场景**

当你知道某个功能存在缺陷，暂时无法修复时，你可以使用 xfail 标记该测试。

* **预期失败的测试意外通过**

当原本标记为 xfail 的测试意外通过时，可能表示 bug 已经被修复，或者功能得到了改善。你需要注意到这种情况，并考虑将该测试移除 xfail 标记。

* **在持续集成中处理已知问题**

在持续集成（CI）环境中，xfail 和 xpass 可帮助团队标记已知的缺陷和修复进度。这样，当这些测试失败时，你可以清楚地知道是预期中的问题，并且不会干扰整体测试结果。

##### pytest.mark.filterwarnings

pytest.mark.filterwarnings 是 pytest 提供的一个标记，用于 **过滤或捕获** 测试过程中产生的警告信息。它可以用来指定某些警告类型在测试执行期间应当被忽略、记录或直接处理，从而避免不必要的警告干扰测试结果，或确保特定的警告被记录和处理。

###### pytest.mark.filterwarnings 的基本用法

filterwarnings 允许你在特定的测试函数、类或模块级别设置警告过滤规则，控制哪些警告需要被捕获、忽略或记录。它提供了更精细的控制，让你在运行测试时能够更好地管理警告。

**语法：**

**pytest.mark.filterwarnings(warning\_expression)**

* warning\_expression：这个表达式指定了如何过滤警告。它通常是一个字符串，描述警告的类型、来源和行为。
* **常见警告过滤表达式：**可以通过以下参数来指定过滤规则：
* "ignore"：忽略指定类型的警告。
* "always"：总是显示指定类型的警告。
* "module"：只在当前模块中显示警告。
* "once"：仅在一次测试运行中显示一次警告。

示例：

import pytest

# 过滤警告，只忽略DeprecationWarning类型的警告

@pytest.mark.filterwarnings("ignore::DeprecationWarning")

def test\_deprecated\_function():

    # 测试代码，可能会触发一个 DeprecationWarning

    pass

###### 使用 pytest.mark.filterwarnings 的场景

pytest.mark.filterwarnings 适用于以下几种情况：

* **忽略某些警告**

如果你不希望某些特定的警告干扰测试结果，可以使用 filterwarnings 来忽略它们。例如，你可能会遇到一些 弃用警告（DeprecationWarnings），但这些警告不影响测试逻辑，可以选择忽略它们。

* **只显示特定的警告**

如果你只希望看到某些类型的警告，比如 资源警告（ResourceWarnings），可以设置 filterwarnings 规则，只显示这类警告：

* **捕获并记录警告**

你也可以使用 filterwarnings 来确保某些警告在测试过程中被记录下来，从而能够在后续的测试报告中查看和分析。这对于排查某些潜在问题非常有用。

* **针对模块或类设置警告过滤规则**

如果希望对整个模块或测试类应用警告过滤规则，可以在模块级别或类级别使用 filterwarnings。例如，你可以只在某个测试类中忽略警告：

##### 自定义mark

在pytest中，mark 是一种标签，用于为测试添加元数据。通过自定义mark，用户可以对测试进行分类、筛选和控制执行。@pytest.mark 装饰器允许我们为测试添加自定义标签（即mark），以便根据需求选择执行特定的测试。

###### 目的与作用

自定义mark的作用主要包括：

* **测试分类**：例如，将测试分为smoke（冒烟测试）、regression（回归测试）、integration（集成测试）等。
* **筛选执行：**用户可以通过命令行选项（pytest -m）仅运行标记了特定标签的测试。
* **提高可维护性**：通过自定义标记，可以更方便地对不同类别的测试进行管理和维护，增强测试的可读性和可控性。

###### 基本使用

* **定义自定义mark**

可以通过 @pytest.mark 装饰器定义自己的标记。自定义标记通常使用一种符合Python标识符规范的名称。

import pytest

# 自定义mark - 标记为smoke测试

@pytest.mark.smoke

def test\_smoke\_1():

    assert 1 == 1

# 自定义mark - 标记为regression测试

@pytest.mark.regression

def test\_regression\_1():

    assert 2 == 2

* **在命令行中使用自定义mark筛选测试**

执行时，可以通过-m选项来筛选执行特定标记的测试。例如，执行所有标记为smoke的测试：

pytest -m smoke

要执行多个标记，可以使用逻辑运算符（如and、or）来组合多个条件：

pytest -m "smoke and regression"  # 只运行既标记为smoke又标记为regression的测试

* **自定义mark添加描述**

为避免pytest发出警告，建议在pytest.ini中声明允许的自定义mark。这样可以让pytest知道哪些标记是你自己定义的。

pytest.ini 文件示例：

[pytest]

markers =

    smoke: 快速验证关键功能

    regression: 回归测试

    integration: 集成测试

#### 数据驱动测试

在 Pytest 中，数据驱动测试通常是通过**参数化（parametrize）**来实现的。参数化使得同一个测试用例可以使用不同的输入数据进行多次执行。Pytest 提供了 **@pytest.mark.parametrize** 装饰器（基本用法见前面）来实现DDT，使得我们能够方便地传入不同的数据集，执行相同的测试逻辑。它支持从内部硬编码数据、外部文件读取数据，甚至与 **Fixture** 配合使用来处理前置条件和资源管理。

##### pytest.mark.parametrize内部数据DDT

最常见的数据驱动测试实现是通过 **内部数据** 实现的，即数据直接在测试代码中定义。这种方式的优势是简单且直观，适用于数据较少、固定的情况。

**例1：基本的内部数据 DDT：**

import pytest

# 使用 pytest.mark.parametrize 装饰器传递数据集

@pytest.mark.parametrize("num1, num2, expected\_sum", [

    (1, 2, 3),  # 第一组数据

    (4, 5, 9),  # 第二组数据

    (7, 8, 15)  # 第三组数据

])

def test\_addition(num1, num2, expected\_sum):

    assert num1 + num2 == expected\_sum

* @pytest.mark.parametrize 装饰器接收两个参数：第一个是需要传递给测试函数的参数名称（即 "num1", "num2", "expected\_sum"），第二个是一个数据列表，其中每一项代表一组数据。
* 每次 Pytest 执行测试时，会分别用 (1, 2, 3)、(4, 5, 9) 和 (7, 8, 15) 来调用 test\_addition 测试函数。
* 这种方式非常直观，代码清晰，但适用于数据不多、变化不大的场景。

**例2：生成器加载数据**

import pytest

def generate\_data():

    # 这里可以从数据库、API 或其他地方获取动态数据

    yield (1, 2, 3)

    yield (2, 3, 5)

    yield (3, 4, 7)

@pytest.mark.parametrize("a, b, expected", generate\_data())

def test\_addition(a, b, expected):

    result = a + b

    assert result == expected

generate\_data 函数使用了 yield 来逐条返回数据，这对于需要处理大数据集或动态数据源的场景非常有用。

**使用场景：**

* 测试**数据量较小且数据已知**，且不需要频繁更新。
* 常用于**小范围的验证和单元测试**。

##### pytest.mark.parametrize外部数据DDT

当测试数据较多或会动态变化时，将数据存储在外部文件中（如 **JSON**、**CSV**、**数据库** 等）更为高效。通过外部数据源，可以使测试数据更易于维护和更新，同时避免硬编码的麻烦。

**文件读取数据测试：**

假设你有一个 CSV 文件，包含多个测试用例数据，可以将这些数据传递到测试函数中。

import pytest

import csv

def load\_data\_from\_csv(file\_path):

    data = []

    with open(file\_path, mode='r') as file:

        reader = csv.reader(file)

        next(reader)  # 跳过表头

        for row in reader:

            # 假设 CSV 的格式为 a, b, expected

            data.append((int(row[0]), int(row[1]), int(row[2])))

    return data

@pytest.mark.parametrize("a, b, expected", load\_data\_from\_csv('test\_data.csv'))

def test\_addition(a, b, expected):

    result = a + b

    assert result == expected

在这个例子中，load\_data\_from\_csv 函数读取 CSV 文件并返回数据。然后，@pytest.mark.parametrize 将 CSV 中的每一行数据传递给 test\_addition 函数进行测试。

**优势：**

* 数据来源灵活，可以通过不同的数据源（如 CSV、数据库）加载。
* 数据可以单独管理和更新，避免硬编码在测试代码中。
* 测试用例与数据分离，提高了可维护性和可扩展性。

**使用场景：**

* 测试数据量较大，或者测试数据需要频繁变更。
* 适用于从外部系统获取数据或配置的情况，如从数据库查询、读取配置文件等。

##### 实际生产中的DDT

规范化的数据驱动测试（DDT）减少硬编码、增强灵活性、确保可维护性，基本原则：

1. **数据与逻辑分离**

* **原则：**将测试数据和测试逻辑分开，确保测试逻辑专注于执行测试而不涉及具体的数据内容。
* **做法：**使用参数化方法（如 pytest.mark.parametrize）或外部化配置文件（如 JSON、CSV、YAML 等）来加载测试数据。

1. **规范外部数据格式**

* **原则**：统一测试数据的存储格式，确保数据一致性和可扩展性。
* **做法：**根据需要选择合适的数据格式（如 JSON、CSV、YAML），确保数据结构清晰、易于理解，便于后期维护和扩展。

1. **外部数据源与动态加载**

* **原则：**避免硬编码测试数据，确保数据源独立且可以动态加载。
* **做法：**通过配置文件或数据库来存储和加载测试数据，确保每次测试都使用最新的数据集，而不需要修改测试代码。

1. **统一数据管理**

* **原则：**集中管理测试数据的加载与格式化，避免在多个地方重复编写相同的加载逻辑。
* **做法：**创建数据管理模块，将数据加载、验证、格式化等功能集中处理，确保所有测试用例使用统一的标准和方法。

1. **灵活的数据维护**

* **原则：**测试数据可以随着业务变化进行灵活调整，减少修改测试代码的频率。
* **做法**：使用结构化的配置文件（如 JSON、CSV）和自动化工具来处理数据更新和验证，确保测试数据的可维护性和适应性。

#### 插件与集成

##### 常用插件

* **核心测试增强​​**

|  |  |  |
| --- | --- | --- |
| **插件名** | **功能描述** | **安装命令** |
| ​**​pytest-xdist​**​ | 多进程/多线程并行运行测试，大幅加速测试执行 | pip install pytest-xdist |
| ​**​pytest-cov​**​ | 生成代码覆盖率报告，支持多种格式（HTML、XML） | pip install pytest-cov |
| ​**​pytest-mock​**​ | 简化 Mock 对象使用，集成 unittest.mock | pip install pytest-mock |
| ​**​pytest-asyncio​**​ | 支持异步测试（async/await），简化协程测试编写 | pip install pytest-asyncio |
| ​**​pytest-timeout​**​ | 设置超时机制，防止测试卡死 | pip install pytest-timeout |

* **测试框架集成​​**

这些插件相比于原生可以更好的适应pytest框架，整体使用上基本一致，但是会提供适应pytest的一些参数化或者fixture功能

|  |  |
| --- | --- |
| ​**插件名** | **功能描述** |
| ​**​pytest-django​**​ | 为 Django 项目提供测试支持（数据库事务、请求客户端） |
| ​**​pytest-flask​**​ | 简化 Flask 应用测试（客户端、上下文、路由） |
| ​**​pytest-tornado​**​ | 支持 Tornado 异步 HTTP 服务器的测试 |
| ​**​pytest-selenium​**​ | 集成 Selenium 进行浏览器自动化测试 |
| ​**​pytest-requests​**​ | 快速测试 HTTP API（封装 requests 库） |

* **报告与输出优化​​**

|  |  |
| --- | --- |
| **插件名** | **功能描述** |
| ​**​pytest-html​**​ | 生成美观的 HTML 测试报告，包含详细用例和日志 |
| ​**​pytest-sugar​**​ | 美化控制台输出（彩色进度条、即时失败显示） |
| ​**​pytest-verbose​**​ | 更详细的测试输出（显示每个用例名称） |
| ​**​pytest-instafail​**​ | 实时显示失败用例的错误信息（无需等待测试结束） |

* ​​**参数化与数据驱动​​**

|  |  |
| --- | --- |
| **插件名** | **功能描述** |
| ​**​pytest-datadir​**​ | 管理测试数据文件（自动加载项目目录中的测试数据） |
| ​**​pytest-cases​**​ | 高级参数化工具，支持更灵活的数据组合和标记 |
| ​**​pytest-bdd​**​ | 支持行为驱动开发（BDD），集成 Gherkin 语法（类似 Cucumber） |

* ​​**依赖与顺序控制​​**

|  |  |
| --- | --- |
| **插件名** | **功能描述** |
| ​**​pytest-dependency​**​ | 管理测试用例依赖关系（确保某些用例在其他用例成功后运行） |
| ​**​pytest-ordering​**​ | 控制测试用例执行顺序（通过装饰器标记） |

* ​​**性能与资源管理​​**

|  |  |
| --- | --- |
| **插件名** | **功能描述** |
| ​**​pytest-benchmark​**​ | 性能基准测试（统计代码执行时间、对比历史结果） |
| ​**​pytest-freezegun​**​ | 模拟时间（冻结系统时间，测试时间相关逻辑） |
| ​**​pytest-rerunfailures​**​ | 自动重试失败用例（适合处理偶发性失败） |

​​

##### 测试框架集成

框架集成指的是将pytest与其他工具、库、框架等结合，以提高测试的效率、扩展测试的能力，或者更好地与其他部分协作。框架集成通常包括与持续集成（CI）工具、代码覆盖率工具、Web框架（如Django或Flask）、数据库、Mock工具等的集成。

其实就是使用前面提供的插件，针对一些特定的功能或者方法进行框架功能的拓展。

###### 常见集成做法

集成pytest与其他工具时，一般采用以下方法：

* **持续集成（CI）工具的集成：**
* 将 pytest 测试框架与持续集成工具（如 Jenkins、GitLab CI、Travis CI、GitHub Actions 等）结合，使得每次代码变更后自动执行测试，并提供反馈。
* 配置 CI 文件（如 .gitlab-ci.yml 或 .github/workflows）来执行 pytest 测试，获取测试结果，并可通过邮件、Slack 等通知开发者。
* **与 Web 框架的集成：**
* 在使用如 Django、Flask、FastAPI 等 Web 框架时，可以通过相应的 pytest 插件（如 pytest-django、pytest-flask）进行集成，简化 Web 应用的测试。
* 这些插件可以自动创建数据库、模拟 HTTP 请求，或者提供与 Web 框架相关的上下文和支持。
* **与 Mock 库的集成：**

pytest 与 Mock 工具（如 unittest.mock 或 pytest-mock）结合时，能够模拟外部 API、函数或服务的行为，减少对实际外部依赖的依赖，保证测试环境的稳定性。

* **与数据库的集成：**
* pytest 与数据库的集成通常通过插件实现，帮助自动化数据库的设置、清理及事务回滚。
* pytest-django 自动创建测试数据库并清理数据，避免对生产环境的影响。
* **与代码覆盖率工具的集成：**

pytest 可以与代码覆盖率工具（如 coverage.py）集成，通过插件如 pytest-cov 生成详细的代码覆盖率报告。

* **与报告工具的集成：**
* pytest 可以与报告生成工具集成，如 pytest-html 生成 HTML 格式的测试报告。
* 通过集成，开发人员可以获取更友好的测试结果，方便与团队分享。
* **与任务调度工具的集成：**

pytest 可以与任务调度工具（如 Celery）进行集成，进行异步任务的测试，验证任务队列和异步任务的执行情况。

###### 集成的优势

集成 pytest 与其他框架和工具后，可以获得多方面的好处：

1. **提升自动化程度：**

通过集成持续集成（CI）工具、自动化报告生成和数据库管理等，可以让测试变得更加自动化，减少人工干预，确保每次提交或代码变更都能触发自动化测试。

1. **提高测试效率：**

pytest 与并行测试插件（如 pytest-xdist）集成，可以显著提高测试的执行速度，尤其是在有大量测试用例时。通过分布式运行测试，可以减少测试总时间。

1. **优化测试覆盖率：**

集成代码覆盖率工具（如 pytest-cov）可以帮助开发者全面了解哪些代码部分被测试了，哪些没有，进而提高测试的全面性和准确性。

1. **增强测试可视化：**

集成测试报告生成工具（如 pytest-html）可以生成结构化的、易于阅读的 HTML 测试报告，使团队成员能够清晰地查看测试结果。

1. **简化复杂应用的测试：**

对于 Web 应用、数据库、外部 API 等复杂的系统，集成插件如 pytest-django、pytest-flask、pytest-mock 等，能够简化测试的编写，模拟外部环境，确保测试过程中的依赖能够正常工作。

1. **减少测试的维护成本：**

通过集成和自动化工具，可以减少手动测试和配置环境的频繁操作，减少人工错误，提升测试的可重复性和可维护性。

1. **增强团队协作：**

自动化和集成使得测试能够随时随地运行，团队成员可以实时了解测试结果，及时发现问题并进行修复，提升开发和测试的协作效率。

1. **支持多环境和多平台测试：**

与 CI/CD 系统结合，pytest 可以在不同的环境和平台（如不同的操作系统或云服务）上运行，确保代码在多种环境下的兼容性。

1. **提高代码质量：**

自动化测试能够迅速发现代码中的缺陷和问题，早期修复错误，避免缺陷流入生产环境，从而提高软件的质量。

### Unittest框架

unittest 是 Python 标准库中的一个单元测试框架，它基于 xUnit 测试模型，提供了一组用于编写和运行测试的工具和功能。unittest 通常用于单元测试，但也可以进行功能测试和集成测试。

#### 基本用法

在 unittest 中，通常通过继承 unittest.TestCase 类来定义测试用例。每个测试方法的名称必须以 test\_ 开头，这样 unittest 能识别它是一个测试方法。

**基本代码结构：**

import unittest

class MyTestCase(unittest.TestCase):

    # 测试方法

    def test\_addition(self):

        self.assertEqual(1 + 1, 2)  # 断言1+1应该等于2

    def test\_subtraction(self):

        self.assertEqual(2 - 1, 1)  # 断言2-1应该等于1

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()  # 运行所有测试用例

* MyTestCase 是一个继承自 unittest.TestCase 的类，表示一个测试用例。
* test\_addition 和 test\_subtraction 是测试方法，测试了加法和减法操作。
* self.assertEqual(a, b) 是 unittest 提供的断言方法，用于验证 a 和 b 是否相等。
* unittest.main() 会自动发现和运行当前模块中的所有测试方法。

#### 断言方法

unittest.TestCase 提供了多种断言方法，用来验证测试结果是否符合预期。

* **assertEqual(a, b)**：检查 a == b，如果不相等则测试失败。
* **assertNotEqual(a, b)**：检查 a != b，如果相等则测试失败。
* **assertTrue(x)**：检查 x 是否为 True，如果不是则测试失败。
* **assertFalse(x)**：检查 x 是否为 False，如果不是则测试失败。
* **assertIsNone(x)**：检查 x 是否为 None。
* **assertIsNotNone(x)**：检查 x 是否不为 None。
* **assertIn(a, b)**：检查 a 是否在 b 中（如列表、字典等）。
* **assertNotIn(a, b)**：检查 a 是否不在 b 中。
* **assertRaises(exception, callable)**：检查 callable 执行时是否抛出指定的异常。

#### 测试套件（Test Suite）

TestSuite 用于**将多个测试用例组织成一个集合**，方便一起执行。你可以通过 addTest() 方法将多个测试用例添加到一个测试套件中，并通过 TextTestRunner 执行这个套件。

**示例：**

import unittest

class TestAddition(unittest.TestCase):

    def test\_add(self):

        self.assertEqual(1 + 1, 2)

class TestSubtraction(unittest.TestCase):

    def test\_subtract(self):

        self.assertEqual(2 - 1, 1)

# 创建测试套件

suite = unittest.TestSuite()

suite.addTest(TestAddition('test\_add'))

suite.addTest(TestSubtraction('test\_subtract'))

# 运行测试套件

runner = unittest.TextTestRunner()

runner.run(suite)

#### 测试夹具（Test Fixtures）

测试夹具用于在测试执行前后做一些初始化和清理工作。setUp() 和 tearDown() 是 unittest 提供的两个重要方法，用于在每个测试方法**前后执行特定的操作**。

* **setUp()：**每个测试方法执行之前调用，用于准备测试环境。
* **tearDown()：**每个测试方法执行之后调用，用于清理测试环境。

示例：

import unittest

class MyTestCase(unittest.TestCase):

    def setUp(self):

        self.data = [1, 2, 3]  # 初始化数据

    def tearDown(self):

        self.data = None  # 清理数据

    def test\_sum(self):

        self.assertEqual(sum(self.data), 6)

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

#### 跳过测试（Skipping Tests）

有时，我们可能不希望某些测试在某些情况下执行。unittest 提供了装饰器来跳过测试：

* **@unittest.skip(reason)：**跳过当前测试。
* **@unittest.skipIf(condition, reason)：**如果条件为 True，则跳过测试。
* **@unittest.skipUnless(condition, reason)：**如果条件为 False，则跳过测试。
* **@unittest.expectedFailure：**标记为预期失败的测试，如果测试失败，不会被认为是错误。

**示例：**

import unittest

class MyTestCase(unittest.TestCase):

    @unittest.skip("This test is skipped")

    def test\_skip(self):

        self.assertEqual(1 + 1, 3)

    @unittest.expectedFailure

    def test\_expected\_failure(self):

        self.assertEqual(1 + 1, 3)

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

#### 7. 运行测试

unittest 提供了几种方式来运行测试：

* 直接在命令行运行：可以通过命令行运行脚本，unittest 会自动识别以 test\_ 开头的方法并执行它们。

**python -m unittest my\_test\_script.py**

* 使用 unittest.main()：在测试文件中调用 unittest.main()，它会自动查找当前模块中的所有测试方法并执行。
* 使用测试套件：可以手动创建测试套件，并通过 TextTestRunner 来执行套件中的测试。

#### 对比Pytest

* **优点对比：**
* unittest：
* 作为 Python 标准库的一部分，不需要额外安装，适合大型项目和团队开发。
* 更加结构化和规范，适合需要严格组织和管理测试的项目。
* 支持测试用例组织、测试夹具、跳过测试等标准化功能。
* pytest：
* 更加简洁易用，适合快速编写和调试测试，支持函数式测试。
* 丰富的断言信息和简洁的语法使得测试更加直观。
* 强大的插件系统，支持并行测试、报告生成等功能，适合多种扩展需求。
* **缺点对比：**
* unittest：
* 语法较为繁琐，特别是在需要编写大量测试用例时，代码冗长。
* 配置和扩展相对较为复杂。
* pytest：
* 需要额外安装，虽然 pytest 本身安装简单，但并不是 Python 标准库的一部分。
* 对初学者来说，可能不如 unittest 那样规范和结构化，容易出现不统一的写法。

## 接口自动化

接口自动化测试是软件测试领域中的重要组成部分，主要用于验证应用程序接口（API）的功能、性能及可靠性。它通过编写脚本或使用工具模拟客户端与服务器之间的交互，自动执行测试用例并生成报告。

最灵活、最常用的接口自动化方法是Requests+Python

### Requests基础-请求、响应

详细可见[Requests](#_Requests)，此处精简介绍

requests 是一个简单易用的 Python HTTP 库，用于发送 HTTP 请求，处理响应。它设计简洁，封装了底层细节，适合用来与 Web 服务、API 交互。

* **请求**
* **发送请求**：支持常见的 HTTP 方法（GET、POST、PUT、DELETE、HEAD 等）。
* **请求参数**：可以方便地传递 URL 参数、表单数据、JSON 数据等，支持自定义请求头（headers）和 cookies。
* **会话管理**：通过 Session 对象，支持保持会话状态，适合处理登录认证等需要保持连接的情况。
* **响应**
* **响应对象**：返回的响应对象包含了状态码、响应头、响应体等信息。
* **状态码**：通过 .status\_code 获取响应状态码。
* **响应体**：支持获取文本内容（.text），JSON 数据（.json()），或二进制数据（.content）。
* **错误处理**：内置错误处理机制，通过 .raise\_for\_status() 方法检查请求是否成功。

### Requests Cookie/Session处理方式

Requests库提供对Cookie和Session的管理以实现持续会话。

* **Cookie 处理**：requests 允许直接在请求中发送和接收 Cookie，支持通过 cookies 参数发送 Cookie，并且可以自动管理和携带服务器返回的 Cookie。
* **Session 处理**：requests.Session() 对象提供了持久化的 HTTP 会话管理，能够自动管理请求的 Cookie 和请求头，复用 TCP 连接，提升性能。使用 Session 对象发起多个请求时，可以保证这些请求共享同一份 Cookie 和请求头。

可见[Session与持久化配置](#_Session与持久化配置)

#### Cookie 处理

Cookie 是由服务器发送给客户端的小块数据，客户端会在每次向同一个服务器发起请求时携带该 Cookie，服务器则通过 Cookie 来维持会话和跟踪用户。

##### 发送请求时携带 Cookie

requests 可以通过 cookies 参数直接传递 Cookie，或者通过 Session 对象自动管理 Cookie。

**示例：直接携带 Cookie 发送请求**

import requests

import requests

url = "https://httpbin.org/cookies"

cookies = {

    "session\_id": "abc123",

    "user": "john\_doe"

}

response = requests.get(url, cookies=cookies)

print(response.json())

在这个例子中，我们通过 cookies 参数在请求头中发送了一个 Cookie。服务器会返回带有 Cookie 的响应，客户端可以进一步使用这些信息。

##### 接收和保存 Cookie

服务器在响应中通常会设置一些 Cookie，requests 会自动接收并保存这些 Cookie，下一次发起请求时会自动携带这些 Cookie。

import requests

# 第一次请求，服务器返回 Cookie

response = requests.get("https://httpbin.org/cookies/set?name=value")

print(response.cookies)

# 第二次请求，requests 会自动带上上次请求的 Cookie

response = requests.get("https://httpbin.org/cookies")

print(response.json())

在上面的代码中，第一个请求设置了一个 Cookie（name=value），requests 会在第二次请求时自动将 Cookie 包含在请求头中。通过 response.cookies 可以访问返回的 Cookie。

##### 操作 Cookie（获取和删除）

如果你想对 requests 自动管理的 Cookie 进行操作，可以直接访问 response.cookies 属性，它是一个 RequestsCookieJar 类型的对象。

# 获取指定 Cookie

print(response.cookies.get('name'))

# 删除指定 Cookie

response.cookies.clear('name')

#### Session 处理

requests 的 Session 对象用于在多个请求之间保持连接、共享 Cookie 和其他参数（如请求头）。使用 Session 对象能够显著提高性能，尤其是在进行多次请求时，因为它可以重用连接。

##### Session 基本用法

requests.Session() 创建一个会话对象，通过该对象发出的所有请求将**自动携带同一个 Cookie**。

import requests

# 创建一个 Session 对象

session = requests.Session()

# 发送请求，Session 会自动管理 Cookie

session.get("https://httpbin.org/cookies/set?name=value")

response = session.get("https://httpbin.org/cookies")

print(response.json())

在这个例子中，session.get() 会自动保存并携带 Cookie。因此，第二次请求时，requests 会自动带上服务器返回的 Cookie。

##### Session 自定义请求头

除了 Cookie，Session 还可以自动保持其他请求参数，如请求头（headers）。你可以在 Session 对象上设置全局请求头，所有通过这个 Session 发出的请求都会使用这些头。

session = requests.Session()

session.headers.update({"User-Agent": "MyCustomUserAgent/1.0"})

# 使用自定义的 User-Agent 发送请求

response = session.get("https://httpbin.org/user-agent")

print(response.json())

在这个例子中，通过 session.headers.update() 方法，我们为 Session 设置了一个全局的 User-Agent，这个设置会自动应用到所有通过该 Session 发送的请求中。

##### Session 与持久连接

requests.Session 的另一个优势是它支持持久连接。当你使用 Session 对象时，TCP 连接会被保持复用，这样就避免了每次请求都需要重新建立连接的性能开销。特别是在频繁发起请求时，性能提升明显。

session = requests.Session()

# 发起多个请求时，Session 会复用连接

for \_ in range(5):

    response = session.get("https://httpbin.org/get")

    print(response.status\_code)

##### Session 管理 Cookie

Session 会自动管理和存储服务器发送的 Cookie，在整个 Session 生命周期内，Cookie 会被持续传递到后续的请求中。

session = requests.Session()

response = session.get("https://httpbin.org/cookies/set?name=value")

# 后续请求会自动携带该 Cookie

response = session.get("https://httpbin.org/cookies")

print(response.json())

在上述代码中，Session 会自动管理 Cookie，你无需手动添加。

##### Session 的关闭与清理

当你完成所有请求后，应该显式地关闭 Session，以便释放资源。可以通过调用 session.close() 来关闭会话。或者使用with自动管理上下文。

session = requests.Session()

response = session.get("https://httpbin.org/get")

print(response.status\_code)

# 关闭 Session，释放资源

session.close()

关闭 Session 后，无法再使用该 Session 对象发送请求。如果你需要继续使用，应该重新创建一个新的 Session 对象。

### Requests不同数据类型的处理方式

requests 库通过灵活的参数和接口，使得在发送 HTTP 请求时处理不同类型的数据变得非常简单。无论是**表单数据**、**JSON 数据**、**文件上传**，还是**二进制数据**、**查询参数**等，requests 都提供了直观、易用的方法来处理它们。同时，requests 也能处理**超时、重定向、身份验证**等高级功能，非常适合与 Web 服务、API 进行交互，参考[2.8.3.3请求参数](#_请求参数)

#### 表单数据（Form Data）

提交 HTML 表单数据时，通常使用 application/x-www-form-urlencoded 格式，这是浏览器表单提交的默认格式。

* **处理方式**：使用 data 参数将表单数据作为键值对传递。requests 会自动将数据编码为 URL 编码的格式。

import requests

url = 'https://example.com/submit'

data = {

    'username': 'john\_doe',

    'password': 'securepassword'

}

response = requests.post(url, data=data)

# 输出服务器返回的响应内容

print(response.text)

在这个例子中，requests 会将 data 字典编码为 application/x-www-form-urlencoded 格式并发送给服务器。

* **多部分表单数据：**在上传文件时，表单数据会以 multipart/form-data 格式发送。

files = {'file': open('example.txt', 'rb')}

response = requests.post('https://example.com/upload', files=files)

files['file'].close()

#### JSON 数据

当与 API 进行交互时，常常需要发送 JSON 格式的数据。大多数现代 API 都使用 application/json 内容类型。

* **处理方式：**使用 json 参数传递数据，requests 会自动将 Python 字典转换为 JSON 格式，并设置请求头为 Content-Type: application/json。

import requests

url = 'https://example.com/api'

data = {

    'name': 'John',

    'age': 30

}

response = requests.post(url, json=data)  # 自动转换为 JSON 格式

print(response.json())  # 解析 JSON 响应

例子中，requests 自动将字典 data 转换为 JSON 格式，并发送到服务器。你可以使用 .json() 方法解析 JSON 格式的响应。

#### URL 查询参数（Query Parameters）

在发送 GET 请求时，查询参数通常作为 URL 的一部分附加在请求中，常见于搜索、筛选等操作。

* **处理方式：**通过 params 参数传递查询参数，requests 会自动将它们转换为 URL 编码的查询字符串。

import requests

url = 'https://example.com/search'

params = {'query': 'python', 'page': 2}

response = requests.get(url, params=params)

print(response.url)  # 打印带有查询参数的 URL

print(response.text)  # 输出响应内容

这个请求会生成类似以下的 URL：

https://example.com/search?query=python&page=2

#### 文件上传（Multipart Form Data）

文件上传通常使用 multipart/form-data 编码方式，特别是在通过表单上传文件时。

* **处理方式**：使用 files 参数传递文件，requests 会自动将文件以适当的格式编码，并设置 Content-Type 为 multipart/form-data。

import requests

url = 'https://example.com/upload'

files = {'file': open('file.txt', 'rb')}

response = requests.post(url, files=files)

files['file'].close()  # 确保文件被关闭

print(response.text)  # 打印响应内容

在这个例子中，requests 会将 file.txt 文件作为二进制文件上传到服务器。

如果需要上传多个文件，可以将它们包装在一个字典中：

files = {

    'file1': open('file1.txt', 'rb'),

    'file2': open('file2.jpg', 'rb')

}

response = requests.post(url, files=files)

#### 二进制数据（Binary Data）

当需要发送图片、音频或其他二进制文件时，通常需要通过 POST 请求将其作为原始数据发送。

* **处理方式：**使用 data 参数传递二进制数据。如果是上传文件，则通常使用 files 参数。对于原始二进制数据，使用 data 参数。

import requests

url = 'https://example.com/upload'

with open('image.jpg', 'rb') as f:

    response = requests.post(url, data=f)

print(response.status\_code)

此代码将 image.jpg 文件的二进制（‘rb‘）内容发送到服务器。

#### 自定义请求头（Custom Headers）

在发送 HTTP 请求时，某些应用或 API 可能要求传递特定的 HTTP 头部（如身份验证、内容类型等）。

* **处理方式**：通过 headers 参数传递自定义的 HTTP 头部。

import requests

url = 'https://example.com/api'

headers = {'Authorization': 'Bearer your\_token'}

response = requests.get(url, headers=headers)

print(response.json())  # 打印响应的 JSON 数据

在此示例中，我们为请求添加了一个 Authorization 头部，用于 API 的身份验证。

#### Cookies 处理

有时需要在 HTTP 请求中携带 cookies，通常是为了维持会话（如模拟登录状态）。

* **处理方式：**通过 cookies 参数传递 cookies，requests 会自动处理 cookies 的发送和接收。

import requests

url = 'https://example.com/dashboard'

cookies = {'session\_id': 'your\_session\_id'}

response = requests.get(url, cookies=cookies)

print(response.text)  # 打印响应内容

#### 超时设置（Timeout）

当请求外部资源时，可能会因为网络问题导致请求长时间没有响应。为了防止这种情况，通常需要设置超时。

* **处理方式**：通过 timeout 参数设置请求超时时间，单位为秒。

import requests

url = 'https://example.com/api'

try:

    response = requests.get(url, timeout=5)  # 设置超时为 5 秒

    print(response.text)

except requests.Timeout:

    print('请求超时')

如果请求超过 5 秒未得到响应，requests 将引发 **Timeout** 异常。

#### 重定向与流式传输（Redirect & Streaming）

有些请求会进行重定向，或者需要处理大量数据流（如大文件下载）。

* **处理方式：**
* **重定向**：requests 会默认自动跟随重定向，但可以通过 allow\_redirects 参数控制是否允许重定向。
* **流式传输**：对于大文件下载，使用 stream=True 来避免将整个文件内容加载到内存中。
* **禁用重定向：**

response = requests.get('https://example.com/redirect', allow\_redirects=False)

print(response.status\_code)  # 返回 3xx 状态码表示重定向

* **流式文件下载：**

with requests.get('https://example.com/largefile', stream=True) as r:

    with open('largefile', 'wb') as f:

        for chunk in r.iter\_content(chunk\_size=8192):

            f.write(chunk)

### Requests多测试环境管理

#### 多测试环境管理概念与目的

多测试环境管理指在同一代码库或测试框架中，灵活地切**换并管理不同的测试环境配置。核心目标是让同一套测试代码灵活适配不同环境（如开发、测试、生产）​**​，而无需修改代码逻辑

**例:**可能会有不同的环境变量或配置文件用于开发环境、测试环境和生产环境的 API 地址、认证信息、数据库连接等。合理地管理这些不同环境的配置能够确保测试脚本在不同的环境中运行时能够自动选择合适的配置。

* **多测试环境管理的原因与目的：**

**以配置驱动解耦环境依赖，保障测试安全、高效、可复用。**

* **​​安全隔离​​：**隔离不同环境（开发/测试/生产），防止测试污染生产数据或泄露敏感信息，规避误操作风险。
* **​​灵活适配​​：**通过统一配置切换环境参数（如URL、鉴权），实现一套代码适配多环境差异。
* **​​效率提升​​：**减少人工配置调整，依托自动化快速验证全环境功能兼容性。
* **​​持续交付​​：**支持CI/CD流程自动触发多环境测试，确保版本迭代前后流程连贯。

#### Requsests多测试环境管理的实现

尽管 requests 库本身并没有直接提供多测试环境管理的功能，但可以通过不同的方式来管理多环境配置，并与 requests 结合使用。

实现 requests 的多测试环境管理，关键在于能够根据不同的环境切换 API 地址、认证信息、请求头等配置。常见的方式包括：

* 使用 **环境变量** 来动态选择环境。
* 使用 **配置文件（JSON, YAML 等）** 来存储不同环境的配置信息。
* 使用 **pytest** 的命令行参数来动态选择环境。
* 使用 **Docker** 等容器化技术模拟不同环境。
* 使用 **配置管理库**（如 Dynaconf）来管理多环境配置。

##### 环境变量切换

环境变量是最常见的用于管理不同环境配置的方法之一。你可以通过设置不同的环境变量来指定当前运行的环境（如开发、测试、生产），并根据环境选择不同的配置。

* **实现步骤**：
* 使用 os.environ 来获取当前环境。
* 根据环境加载不同的 API URL、认证信息等。

**示例代码：**

import os

import requests

# 获取环境变量，默认为 'development'

env = os.getenv('ENV', 'development')

# 根据环境选择不同的配置

if env == 'development':

    base\_url = 'https://dev.example.com/api'

    api\_key = 'dev-api-key'

elif env == 'testing':

    base\_url = 'https://test.example.com/api'

    api\_key = 'test-api-key'

else:

    base\_url = 'https://prod.example.com/api'

    api\_key = 'prod-api-key'

# 使用配置发送请求

response = requests.get(f"{base\_url}/endpoint", headers={"Authorization": api\_key})

print(f"Response Status: {response.status\_code}")

**配置环境：**

在运行时，通过环境变量设置当前环境。（命令行运行）例如：

export ENV=development

python test\_script.py  # 运行测试脚本

切换到其他环境时：

export ENV=testing

python test\_script.py

这样可以根据环境变量自动切换不同的 API 配置。

##### 使用 配置文件（JSON, YAML 等）

另一种方式是将不同环境的配置存储在配置文件中。常见的格式有 JSON、YAML、INI 等。你可以读取这些配置文件，并根据当前环境加载相应的设置。

**示例代码：**

假设有一个 config.json 文件，内容如下：

{

  "development": {

    "base\_url": "https://dev.example.com/api",

    "api\_key": "dev-api-key"

  },

  "testing": {

    "base\_url": "https://test.example.com/api",

    "api\_key": "test-api-key"

  },

  "production": {

    "base\_url": "https://prod.example.com/api",

    "api\_key": "prod-api-key"

  }

}

然后，使用 json 库读取该配置文件并根据环境加载相应的配置：

import json

import requests

# 获取环境（例如从命令行或环境变量）

env = 'testing'  # 在实际场景中，可以动态获取

# 加载配置文件

with open('config.json') as f:

    config = json.load(f)

# 获取当前环境的配置

current\_config = config.get(env)

# 使用当前环境配置发起请求

response = requests.get(f"{current\_config['base\_url']}/endpoint", headers={"Authorization": current\_config["api\_key"]})

print(f"Response Status: {response.status\_code}")

**配置不同环境的配置：**

1. 在 config.json 中，可以定义多个环境的 API 地址、认证信息等。
2. 运行时，只需选择适当的环境，代码会自动加载相应的配置。

##### 使用 pytest 管理测试环境

如果使用的是 pytest 进行自动化测试，pytest 提供了丰富的插件和功能，允许你在不同的环境中运行测试。

**示例：使用 pytest 的命令行参数切换环境**

pytest 支持通过命令行参数来传递不同的配置，通常通过 --env 参数来指定当前测试的环境。

1. **创建 pytest.ini 配置文件（可选）**

可以在 pytest.ini 文件中设置默认的环境配置。

1. **编写测试代码**

然后，在测试代码中使用 pytest 的 request 来获取配置：

import pytest

import requests

# pytest fixture 用来获取环境变量

@pytest.fixture

def env(request):

    return request.config.getoption('--env')

# 测试代码，根据环境选择不同的 URL

def test\_api(env):

    if env == 'development':

        base\_url = 'https://dev.example.com/api'

    elif env == 'testing':

        base\_url = 'https://test.example.com/api'

    else:

        base\_url = 'https://prod.example.com/api'

    response = requests.get(f'{base\_url}/endpoint')

    assert response.status\_code == 200

1. **运行测试**

通过 --env 参数传递环境类型：

pytest --env=development

pytest --env=testing

pytest --env=production

在运行测试时，pytest 会根据传入的 --env 参数动态加载不同的配置，保证测试在不同环境中运行时使用正确的 API 地址、认证信息等。

##### Docker或虚拟化

在某些复杂的测试环境中，可能需要同时模拟不同的服务、数据库等。在这种情况下，使用 Docker 容器化管理不同的测试环境是一个非常好的解决方案。你可以为每个环境配置不同的 Docker 容器，确保测试能够在独立的环境中运行。

**示例：使用 Docker Compose 配置多环境**

在 docker-compose.yml 文件中，你可以为开发、测试、生产等环境配置不同的服务：

yaml：

version: '3'

services:

  dev:

    image: myapp:dev

    environment:

      - ENV=development

    ports:

      - "5000:5000"

  test:

    image: myapp:test

    environment:

      - ENV=testing

    ports:

      - "5001:5000"

  prod:

    image: myapp:prod

    environment:

      - ENV=production

    ports:

      - "5002:5000"

在测试中，你可以通过容器间通信来访问不同环境下的服务，确保测试数据与实际环境隔离。

##### 配置管理库（如 Dynaconf, ConfigParser）

如果你的项目需求更加复杂，可以使用一些专门的配置管理库，如 Dynaconf，它支持多环境配置和动态加载配置。你可以根据不同的环境读取并加载相应的配置。

##### 综合使用环境变量和配置文件

在实际项目中，可能会结合使用环境变量与配置文件的方式来管理不同环境。你可以通过环境变量选择当前环境，并使用配置文件加载详细的环境配置。

**示例代码：**

import os

import json

import requests

# 获取当前环境，默认为 development

env = os.getenv('ENV', 'development')

# 加载配置文件

with open('config.json') as f:

    config = json.load(f)

# 获取当前环境的配置

current\_config = config.get(env)

# 使用当前配置发送请求

response = requests.get(f"{current\_config['base\_url']}/endpoint", headers={"Authorization": current\_config["api\_key"]})

print(f"Response Status: {response.status\_code}")

### Requests二次封装

requests 的二次封装主要是将常见的请求逻辑进行抽象和封装，以提高代码的可维护性、可扩展性和复用性。通过封装，可以在一个地方管理请求头、认证、超时、重试等配置，从而使得每次发送 HTTP 请求的代码更加简洁和一致。二次封装可以增加更多功能，如自动重试、认证、日志记录等，提升测试和开发的效率。

#### 二次封装的目的

* **简化请求的调用：**对于常见的请求方法（如 GET、POST、PUT 等），封装后调用更加简单。
* **统一配置：**将请求的基础配置（如请求头、超时设置、代理等）统一管理，避免在每次请求时重复配置。
* **增加错误处理和日志记录：**对请求中的错误进行统一的处理和记录，避免每个请求都需要单独处理异常。
* **扩展功能：**可以集成更多的功能，例如请求重试、身份验证、自动解析响应等。
* **提高可维护性**：在封装层修改功能时，底层代码无需修改，减少维护成本。

#### 二次封装 requests 的基本结构

通常，我们会封装一个请求类，类中提供常见的 HTTP 请求方法（如 GET、POST），并在类中处理一些通用逻辑（如请求头、认证、超时等）。

**一个简单的请求资源类封装：**

class ApiClient:

    def \_\_init\_\_(self, base\_url="https://jsonplaceholder.typicode.com"):

        self.base\_url = base\_url

        self.session = requests.Session()

    def \_\_enter\_\_(self):

        return self

    def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

        self.session.close()

    def send\_request(self, endpoint, method="GET", params=None, data=None):

        url = f"{self.base\_url}/{endpoint}"

        response = self.session.request(method, url, params=params, json=data)

        return response

#### 二次封装的增强功能

在基本封装的基础上，我们可以添加更多功能，如重试机制、自动解析响应、日志记录、认证等。

**1. 自动重试功能**

为了提高请求的稳定性，我们可以为 requests 封装添加重试机制。如果请求失败，自动重试请求一定次数。

**2. 自动认证处理**

许多 API 需要身份验证，常见的认证方式包括 Basic Authentication、Bearer Token 等。我们可以在封装中添加统一的认证处理。

**3. 日志记录功能**

对请求和响应的日志进行记录，便于后期调试和审计。

**4. 响应解析**

对于 API 返回的数据，常常需要根据不同的返回格式（如 JSON、XML）进行解析。我们可以在封装时进行自动解析。

#### 二次封装的好处

* **简化接口调用**：使用二次封装后的 API，可以更加简洁地调用不同的 HTTP 方法（如 GET、POST、PUT、DELETE），无需每次都写大量的重复代码。
* **统一管理配置**：通过二次封装，可以将认证信息、请求头、代理设置等统一管理，避免每次请求时都重复配置。
* **增强功能**：通过封装，可以很方便地加入重试机制、日志记录、认证管理等额外功能。
* **提高可维护性**：业务逻辑和请求的细节处理从调用代码中分离，使得代码更加模块化，便于维护和修改。

### 接口自动化业务流程及流程封装

## web自动化

### 现代浏览器基础

#### 浏览器启动

浏览器启动是浏览器从关闭状态转为可用状态的过程，涉及多个步骤：

1. **初始化和配置加载：**

* 当浏览器被启动时，它首先会加载默认的用户配置，包括浏览器的设置、插件、缓存等。
* 浏览器的主进程会启动，并加载一些底层服务，例如网络服务、UI 组件等。

例如我们认为指定端口启动浏览器为：

"C:\Program Files\Google\Chrome\Application\chrome.exe" --remote-debugging-port=9222 --user-data-dir="C:\Users\29639\AppData\Local\Google\Chrome\User Data"

其中的exe为浏览器程序，但不仅仅是启动浏览器，还需要读取用户的配置、缓存等信息，也就是User Data文件夹，一般来说我们通过点击快捷方式会自动加载一个默认的用户数据。

1. **浏览器进程与渲染进程：**

* **主进程：**浏览器的主进程负责管理浏览器窗口、UI 和后台任务。它会管理各个标签页，处理用户的输入和操作。
* **渲染进程：**每个标签页通常会启动一个单独的渲染进程（也叫工作进程），负责渲染该页面。不同的浏览器可能有不同的进程管理模型（如 Chrome 的多进程架构）。
* **JavaScript 进程**：独立的 JavaScript 线程负责执行页面上的脚本，处理与 DOM 相关的操作。

1. **浏览器初始化过程：**

* 浏览器在启动时加载**核心文件**（如用户配置、缓存、浏览历史等，也就是User Data）并准备启动渲染进程。
* 一旦渲染进程启动，浏览器就准备好加载网页内容。

#### 浏览器结构与工作原理

浏览器的工作原理涉及多个重要的组件，它们相互协作来完成页面加载和渲染任务：

1. **渲染引擎（Rendering Engine）：**

* 渲染引擎是浏览器的**核心组件**，负责**解析 HTML、CSS、JavaScript**，并将页面渲染到用户的屏幕上。
* 渲染引擎的职责包括：解析 HTML 文件并生成 DOM 树，解析 CSS 文件并生成 CSSOM 树，合并 DOM 和 CSSOM 构建渲染树，计算布局、绘制页面。

1. **JavaScript 引擎：**

* 现代浏览器内置的 JavaScript 引擎（如 Chrome 的 V8 引擎、Firefox 的 SpiderMonkey）负责**执行网页中的 JavaScript 代码**。它会处理 DOM 操作、事件监听、Ajax 请求等任务。
* JavaScript 引擎通过解释执行或者 JIT（即时编译）技术将 JavaScript 代码转换为机器语言，以便执行和优化。

1. **UI 组件：**

* 浏览器的**用户界面部分**，包括地址栏、菜单栏、工具栏、开发者工具等。UI 组件负责与用户交互，展示浏览器的控制元素。
* 用户的输入（如地址栏的 URL 输入、鼠标点击等）会被主进程处理，并传递给渲染进程。

1. **网络组件：**

* 网络组件负责**页面资源的请求和响应**，如加载 HTML 文件、CSS 样式表、JavaScript 文件、图片等。它会通过 HTTP/HTTPS 协议与服务器通信。
* 网络组件会处理请求的优化（如请求的合并、缓存管理、异步加载等）。

1. **数据存储：**

* 数据存储模块管理**浏览器的缓存、cookies、localStorage、sessionStorage** 等。本地存储使得浏览器能够在不同会话之间持久化用户数据，提高性能和用户体验。
* Cache API、IndexedDB、localStorage 都是浏览器提供的存储机制，它们允许开发者存储少量或大量的本地数据。

1. **多进程架构：**

* 现代浏览器通常采用多进程架构，**每个标签页、插件或者扩展通常运行在独立的进程中**。这种架构提高了稳定性和安全性，因为一个标签页或插件的崩溃不会影响整个浏览器的运行。

#### 浏览器的功能

现代浏览器具备丰富的功能，能够支持动态和复杂的 Web 应用。主要功能包括：

* **页面渲染：**
* 浏览器将接收到的 HTML、CSS 和 JavaScript 解析为 DOM 树和 CSSOM 树，合并成渲染树，再进行布局和绘制，最终将页面呈现给用户。
* 渲染过程包括获取资源、构建页面结构、布局计算和页面绘制。
* **JavaScript 执行：**
* 浏览器的 JavaScript 引擎负责执行页面上的脚本。它通过事件循环、异步执行、Promise 等机制处理页面交互、动态效果和数据加载。
* **网络请求和响应：**
* 浏览器通过网络组件向服务器发送 HTTP 请求，获取网页资源（如 HTML、CSS、JavaScript、图片等）。它支持 HTTP/HTTPS 协议，并具备缓存策略、压缩技术（如 gzip）等功能。
* 浏览器还可以通过异步请求（AJAX）获取数据，支持 WebSockets 等长连接技术。
* **安全性功能：**
* 同源策略：浏览器的同源策略（Same-Origin Policy）限制了不同域名的脚本和数据访问，保障了 Web 应用的安全性。
* 跨站脚本攻击（XSS）防护：浏览器对脚本注入攻击进行防范，确保网页内容的安全展示。
* 内容安全策略（CSP）：通过 CSP，网站可以控制哪些资源可以加载，进一步提高网页的安全性。
* **用户体验和交互：**
* 浏览器支持各种用户交互，包括鼠标点击、滚动、键盘输入、语音识别、触摸屏操作等。现代浏览器还支持动态效果、动画、媒体播放等多种功能。
* 支持 Web Workers 和 Service Workers，能够提供离线应用、后台任务等功能。
* **开发者工具：**
* 浏览器通常内置开发者工具（如 Chrome DevTools、Firefox Developer Tools），用于调试、分析和优化网页。开发者工具包括：元素检查器、JavaScript 控制台、网络监控、性能分析等功能。

#### 页面资源获取与渲染

浏览器的页面资源获取与渲染过程包括多个关键步骤：

1. **请求资源**：
   * 浏览器启动时，向 Web 服务器发送 HTTP 请求，获取 HTML 页面。页面中的资源（如 CSS 文件、JavaScript 文件、图片等）会通过后续的请求一一加载。浏览器根据页面的内容（如 <link>、<script>、<img> 标签）向服务器请求相应的资源。
2. **解析 HTML 和构建 DOM**：
   * 浏览器解析收到的 HTML 文件，构建 **DOM 树**（Document Object Model）。DOM 是网页的结构化表示，每个 HTML 元素和属性都成为 DOM 树的一个节点。
3. **解析 CSS 和构建 CSSOM**：
   * 浏览器加载和解析 CSS 文件，并构建 **CSSOM 树**（CSS Object Model）。CSSOM 描述了样式和规则，负责网页元素的外观和布局。
4. **合并 DOM 和 CSSOM 构建渲染树**：
   * DOM 树和 CSSOM 树合并形成 **渲染树**。渲染树包含了每个页面元素的可见部分，并对其进行样式计算（如颜色、尺寸、位置）。
5. **计算布局**：
   * 渲染树构建完成后，浏览器计算页面元素的 **布局**，即每个元素在屏幕上的确切位置和大小。
6. **绘制页面**：
   * 一旦布局完成，浏览器会执行 **绘制** 操作，将渲染树的每个节点绘制到屏幕上，最终形成用户可见的页面。
7. **执行 JavaScript**：
   * 如果页面包含 JavaScript，浏览器会启动 JavaScript 引擎执行脚本。JavaScript 可能会动态修改 DOM 或 CSS，导致页面重新渲染。这是现代 Web 应用动态交互的基础。
8. **异步资源加载**：
   * 页面中的异步资源（如通过 <script> 标签加载的 JavaScript 文件，或者通过 AJAX 请求的动态数据）可能会在页面加载的过程中被请求和执行。浏览器会在加载这些资源时通过异步加载的方式保持页面渲染的流畅性。
9. **页面优化与重绘**：
   * 浏览器还会进行性能优化，例如合并 CSS 文件、延迟加载非关键资源、缓存静态资源等，以提升页面加载速度和响应性能。

### Selenium

简单介绍见[Selenium](#_Selenium)，下面为深入具体功能介绍

#### Selenium原理

Selenium操作浏览器的原理如下：

* 用户通过 from selenium import webdriver **使用 WebDriver 客户端**，编写代码，**调用API**（比如 get()、click() 等）发出浏览器操作指令。
* 执行程序时，**WebDriver 客户端会通过 HTTP 协议，发送指令请求到本地的 webdriver 驱动程序**（比如 chromedriver.exe，由 executable\_path 指定）。
* webdriver.exe（本地驱动程序）接收到指令后，会通过浏览器的**底层接口**（比如 Chrome DevTools Protocol、Gecko Remote Protocol 等）**真正去控制浏览器**执行操作，比如跳转页面、点击按钮、输入文本等。
* 执行完操作后，浏览器驱动把执行结果（成功/失败、元素数据等）通过 HTTP 响应返回给 WebDriver 客户端。

**一句话概括就是：**

用户用 WebDriver 客户端库编写代码，客户端库通过 HTTP 请求把命令发给本地下载的浏览器驱动（如 chromedriver.exe），浏览器驱动再通过原生接口操作浏览器。但是由于三个层次间的轮询会导致效率较低。

**Selenium 对浏览器的操作，主要手段有三种：**

|  |  |  |
| --- | --- | --- |
| **方式** | **说明** | **主要用途** |
| **WebDriver API** | 操作浏览器本身。直接通过 driver.get()、driver.refresh()、driver.back()、driver.forward()、driver.execute\_script() 等方法来控制浏览器。 | 控制 **浏览器整体行为**，如打开页面、刷新、跳转、执行 JS 脚本 |
| **WebElement API** | 定位到单个元素后，通过 .click()、.send\_keys()、.clear()、.get\_attribute() 等方法进行具体操作。 | 控制 **页面中的具体元素**，比如点击按钮、输入内容、提取信息 |
| **ActionChains API** | 使用 ActionChains(driver) 创建动作链，组合鼠标和键盘的高级操作，如 .move\_to\_element()（悬停）、.drag\_and\_drop()（拖拽）、.context\_click()（右键）。 | 模拟 **复杂的人机交互**，如拖拽、悬停、组合键（Ctrl+A） |

* **WebDriver** 是全局操作。
* **WebElement** 是局部单元素操作。
* **ActionChains** 是组合动作链，可以跨元素、多步操作。

#### Webdriver

Selenium WebDriver 是 **Selenium** 中最核心的组件，主要用于与浏览器进行交互，模拟用户行为进行自动化测试。它通过直接与浏览器进行通信，控制浏览器的动作（如点击按钮、输入文本、选择下拉框等）。

##### Webdriver架构

WebDriver 的架构基于客户端/服务器模型，其中客户端是测试脚本，服务器是浏览器驱动。每种浏览器都有特定的 WebDriver 实现，这些驱动与浏览器的不同实现通过浏览器的原生接口进行通信。

WebDriver 架构大致如下：

* **客户端（WebDriver API）**：由编写测试脚本的开发者使用的 API
* **浏览器驱动（Browser Driver）**：每种浏览器都有一个专门的驱动，负责与浏览器通信。**WebDriver 客户端通过 HTTP 请求**与这些浏览器驱动通信。
* **浏览器（Browser**）：真实的浏览器（Chrome、Firefox、Edge 等），它执行 WebDriver 指令来操作网页。

##### Webdriver使用

WebDriver 启动的基本步骤包括：

1. 创建一个 WebDriver 实例。
2. 启动浏览器并访问指定的 URL。
3. 对网页进行交互（如查找元素、点击按钮、提交表单等）。
4. 执行完操作后，关闭浏览器。

##### Webdriver初始化

driver由下面规范化写法定义（Edge）：

driver = webdriver.Chrome(service=service, options=options, keep\_alive=True)

* **service（类） 参数**：驱动服务配置​

常见配置项

* + executable\_path：驱动路径
  + port：服务端口（默认自动分配）
  + log\_path：日志文件路径

service = Service(executable\_path="chromedriver.exe", port=9515)

* **options（类） 参数**：浏览器启动配置​

常见配置项和方法

* binary\_location：指定可执行文件路径
* add\_argument():将命令行参数添加到启动配置
* add\_experimental\_option():添加实验性选项
* **keep\_alive 参数**：连接保持机制​
* True: 保持驱动服务进程存活，允许多次调用 driver.quit() 后复用会话（默认值）
* False: 关闭浏览器时立即终止驱动进程

##### Options类配置

###### add\_argument()

添加命令行参数（启动参数），这是直接加启动时的浏览器指令，常用来改变浏览器行为。

|  |  |  |
| --- | --- | --- |
| **参数** | **说明** | **示例** |
| --headless | 无头模式，不打开界面 | options.add\_argument('--headless') |
| --disable-gpu | 禁用GPU加速（有些无头模式必须加） | options.add\_argument('--disable-gpu') |
| --start-maximized | 浏览器启动时最大化 | options.add\_argument('--start-maximized') |
| --window-size=宽,高 | 指定窗口大小 | options.add\_argument('--window-size=1920,1080') |
| --incognito | 无痕模式（隐身模式） | options.add\_argument('--incognito') |
| --disable-extensions | 禁用扩展插件 | options.add\_argument('--disable-extensions') |
| --proxy-server=ip:port | 设置代理服务器 | options.add\_argument('--proxy-server=http://127.0.0.1:8080') |
| --no-sandbox | 取消沙盒模式（Linux常用） | options.add\_argument('--no-sandbox') |
| --disable-dev-shm-usage | 解决Linux共享内存不足问题 | options.add\_argument('--disable-dev-shm-usage') |
| --remote-debugging-port=9222 | 开启远程调试端口 | options.add\_argument('--remote-debugging-port=9222') |
| --disable-blink-features=AutomationControlled | 避免被检测是自动化脚本（反爬用） | options.add\_argument('--disable-blink-features=AutomationControlled') |
| --user-agent=xxx | 修改 User-Agent | options.add\_argument('--user-agent=自定义UA') |
| --lang=xx | 设置浏览器语言 | options.add\_argument('--lang=zh-CN') |

###### add\_experimental\_option()

添加实验性参数（特殊控制），这种是控制一些浏览器“隐藏设置”、“内部开关”。

|  |  |  |
| --- | --- | --- |
| **键（key）** | **说明** | **示例** |
| "excludeSwitches" | 排除特定启动开关 | options.add\_experimental\_option('excludeSwitches', ['enable-automation']) |
| "useAutomationExtension" | 禁用自动化扩展（也用于反检测） | options.add\_experimental\_option('useAutomationExtension', False) |
| **"prefs"** | 配置浏览器偏好设置 | options.add\_experimental\_option('prefs', prefs字典) |

其中 "prefs" 常见配置：

|  |  |  |  |
| --- | --- | --- | --- |
| **类别** | **配置项 Key** | **示例值** | **功能说明** |
| 下载 | "download.default\_directory" | "C:\\Downloads" | 设置默认下载目录（需为绝对路径） |
| "download.prompt\_for\_download" | False | 取消每次下载弹窗，自动保存 |
| "download.directory\_upgrade" | True | 允许下载目录“自动升级”（保持设置） |
| "safebrowsing.enabled" | True / False | 是否启用安全浏览（防恶意文件） |
| "safebrowsing.disable\_download\_protection" | True | 禁用Chrome对潜在危险文件的拦截 |
| "profile.default\_content\_setting\_values.automatic\_downloads" | 1 | 允许多文件自动下载 |
| "plugins.always\_open\_pdf\_externally" | True | 下载PDF而不是在线预览 |
| 弹窗 | "profile.default\_content\_setting\_values.popups" | 0 | 禁止网站弹窗（默认是1允许） |
| "profile.default\_content\_setting\_values.notifications" | 2 | 禁用网页通知（1允许，2禁止） |
| "profile.default\_content\_setting\_values.media\_stream" | 2 | 禁用网页请求摄像头/麦克风权限 |
| 多媒体 | "profile.managed\_default\_content\_settings.images" | 2 | 禁用网页图片加载，加快加载速度 |
| "profile.default\_content\_setting\_values.javascript" | 2 | 禁用 JavaScript（不建议关闭） |
| 隐私 | "profile.default\_content\_setting\_values.geolocation" | 2 | 禁用定位权限 |
| "credentials\_enable\_service" | False | 禁用保存密码提示 |
| "profile.password\_manager\_enabled" | False | 关闭密码管理器 |
| 国际化 | "intl.accept\_languages" | "zh-CN,zh" | 设置浏览器首选语言 |
| 外观 | "webkit.webprefs.default\_font\_size" | 16 | 默认字体大小 |
| "webkit.webprefs.default\_fixed\_font\_size" | 14 | 固定宽度字体大小 |

###### 其他 Options 提供的方法

直接调用，不用add\_argument()等函数设置

|  |  |  |
| --- | --- | --- |
| **方法** | **说明** | **示例** |
| options.binary\_location = "浏览器路径" | 指定浏览器可执行文件的位置 | options.binary\_location = "C:\\Program Files\\Chrome\\Application\\chrome.exe" |
| options.add\_extension('扩展.crx') | 加载插件扩展（CRX文件） | options.add\_extension('path/to/extension.crx') |
| options.add\_encoded\_extension(编码扩展) | 加载已经Base64编码过的插件 | 用得少 |
| options.set\_capability(名字,值) | 直接设置某些capabilities，属于更高级别操作 | options.set\_capability('acceptInsecureCerts', True) |

##### Webdriver常用操作

下面是driver直接相关的常见操作

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类别** | **操作** | **方法** | **说明** | **示例** |
| **浏览器控制** | 打开网页 | driver.get(url) | 打开指定 URL 页面 | driver.get("https://www.google.com") |
| 获取页面标题 | driver.title | 获取当前页面的标题 | print(driver.title) |
| 获取当前 URL | driver.current\_url | 获取当前浏览器页面的 URL | print(driver.current\_url) |
| 获取页面源代码 | driver.page\_source | 获取当前页面的 HTML 源代码 | print(driver.page\_source) |
| **窗口操作** | 切换窗口 | driver.switch\_to.window(window\_handle) | 切换到指定的浏览器窗口 | driver.switch\_to.window(driver.window\_handles[1]) |
| 获取所有窗口句柄 | driver.window\_handles | 获取所有打开的浏览器窗口句柄 | handles = driver.window\_handles |
| 后退浏览器 | driver.back() | 模拟浏览器的后退操作 | driver.back() |
| 前进浏览器 | driver.forward() | 模拟浏览器的前进操作 | driver.forward() |
| 刷新页面 | driver.refresh() | 刷新当前页面 | driver.refresh() |
| **弹窗操作** | 切换到警告框 | driver.switch\_to.alert | 切换到当前的警告框（alert、confirm等） | alert = driver.switch\_to.alert |
| 接受警告框 | alert.accept() | 确认警告框（类似点击“确定”） | alert.accept() |
| 拒绝警告框 | alert.dismiss() | 拒绝警告框（类似点击“取消”） | alert.dismiss() |
| 在警告框输入文本 | alert.send\_keys(keys) | 在警告框输入文本（适用于 prompt 类型的警告框） | alert.send\_keys("Some input") |
| **JavaScript 执行** | 执行 JavaScript 代码 | driver.execute\_script(script, \*args) | 执行一段 JavaScript 代码 | driver.execute\_script("return document.title") |
| 执行 JavaScript 并返回值 | driver.execute\_async\_script(script, \*args) | 执行异步 JavaScript 并返回值 | driver.execute\_async\_script("return 2 + 2") |
| **截图操作** | 截取屏幕并保存为文件 | driver.get\_screenshot\_as\_file(filename) | 截取当前页面的屏幕截图并保存到指定路径 | driver.get\_screenshot\_as\_file("screenshot.png") |
| **等待操作** | 设置隐式等待 | driver.implicitly\_wait(time\_to\_wait) | 设置查找元素的最大等待时间 | driver.implicitly\_wait(10) |
| **页面滚动** | 滚动到页面底部 | driver.execute\_script("window.scrollTo(0, document.body.scrollHeight);") | 滚动到页面底部 | driver.execute\_script("window.scrollTo(0, document.body.scrollHeight);") |
|  | 滚动到页面顶部 | driver.execute\_script("window.scrollTo(0, 0);") | 滚动到页面顶部 | driver.execute\_script("window.scrollTo(0, 0);") |
| **处理Cookies** | 获取所有 Cookies | driver.get\_cookies() | 获取当前浏览器中的所有 Cookies | cookies = driver.get\_cookies() |
| 获取指定 Cookie | driver.get\_cookie(name) | 获取指定名称的 Cookie | cookie = driver.get\_cookie("session\_id") |
| 添加 Cookie | driver.add\_cookie(cookie\_dict) | 添加一个新的 Cookie | driver.add\_cookie({"name": "test", "value": "123"}) |
| 删除指定 Cookie | driver.delete\_cookie(name) | 删除指定名称的 Cookie | driver.delete\_cookie("test") |
| 删除所有 Cookies | driver.delete\_all\_cookies() | 删除所有的 Cookies | driver.delete\_all\_cookies() |
| **管理浏览器窗口** | 设置窗口大小 | driver.set\_window\_size(width, height) | 设置浏览器窗口的大小 | driver.set\_window\_size(1024, 768) |
| 获取窗口大小 | driver.get\_window\_size() | 获取浏览器当前窗口的大小 | print(driver.get\_window\_size()) |
| 设置窗口位置 | driver.set\_window\_position(x, y) | 设置浏览器窗口的位置 | driver.set\_window\_position(0, 0) |
| 获取窗口位置 | driver.get\_window\_position() | 获取浏览器当前窗口的位置 | print(driver.get\_window\_position()) |

#### 元素定位与操作

元素定位与操作是自动化测试的核心部分。通过定位方法，我们可以找到网页上的元素，然后通过各种操作函数与这些元素进行交互。

##### 八大定位法（表达式）

Selenium 提供了八种常见的元素定位方法，通过By类来枚举策略。它们用于通过不同的方式查找网页中的元素。每种方法的适用场景不同，根据实际需求选择合适的定位方法。

|  |  |  |  |
| --- | --- | --- | --- |
| **定位方式** | **定位方法** | **说明** | **示例代码** |
| **ID** | By.ID | 通过元素的 id 属性定位，id 是唯一的，效率高。 | driver.find\_element(By.ID, "element\_id") |
| **Class Name** | By.CLASS\_NAME | 通过元素的 class 属性定位，多个元素可能有相同的 class。 | driver.find\_element(By.CLASS\_NAME, "class\_name") |
| **Name** | By.NAME | 通过元素的 name 属性定位，通常用于表单元素。 | driver.find\_element(By.NAME, "element\_name") |
| **Tag Name** | By.TAG\_NAME | 通过元素的 HTML 标签定位，适合查找相同标签的多个元素。 | driver.find\_element(By.TAG\_NAME, "button") |
| **Link Text** | By.LINK\_TEXT | 通过 <a> 标签的完整链接文本定位，要求文本完全匹配。 | driver.find\_element(By.LINK\_TEXT, "Click Here") |
| **Partial Link Text** | By.PARTIAL\_LINK\_TEXT | 通过 <a> 标签的部分链接文本定位，支持部分文本匹配。 | driver.find\_element(By.PARTIAL\_LINK\_TEXT, "Click") |
| **CSS Selector** | By.CSS\_SELECTOR | 通过 CSS 选择器定位，灵活且功能强大，支持多种属性与层级。 | driver.find\_element(By.CSS\_SELECTOR, ".class\_name > button") |
| **XPath** | By.XPATH | 通过 XPath 路径定位，支持根据节点、属性、层级等多重条件查找。 | driver.find\_element(By.XPATH, "//div[@id='element\_id']") |

##### 选择器函数

选择器函数是通过不同的定位方法实现元素查找的另一种方式，适用于更复杂的查询条件。find\_element 和 find\_elements 是常用的函数，用于定位单个或多个元素。

* **find\_element()：定位单个元素**
* 用法：find\_element(By.LOCATOR\_TYPE, "value")
* 返回：返回 WebElement 对象，代表定位到的单个元素。如果没有找到元素，会抛出 NoSuchElementException 异常。

element = driver.find\_element(By.ID, "login\_button")

* **find\_elements()：定位多个元素**
* 用法：find\_elements(By.LOCATOR\_TYPE, "value")
* 返回：返回一个 WebElement 列表，包含所有匹配的元素。如果没有找到任何元素，返回空列表。

elements = driver.find\_elements(By.CLASS\_NAME, "button\_class")

##### 元素操作

Selenium的**元素操作是通过WebElement实现**的，WebElement 是 Selenium 中网页元素的对象化表示，通过 WebElement，可以对元素进行各种操作，比如点击、输入、读取内容、检查状态等等。

**网页上的一个元素，被Selenium“抓下来”，变成了可以操作的对象。**

通过WebElement，你可以实现下面的元素操作：

|  |  |  |  |
| --- | --- | --- | --- |
| **类别** | **方法** | **说明** | **示例** |
| 元素交互 | click() | 单击元素（按钮、链接等） | element.click() |
| send\_keys(\*value) | 向输入框发送文字 | element.send\_keys("hello") |
| clear() | 清空输入框内容 | element.clear() |
| submit() | 提交表单（等价于点击提交按钮） | element.submit() |
| 属性操作 | get\_attribute(name) | 获取元素属性值 | href = element.get\_attribute("href") |
| get\_property(name) | 获取元素属性（更偏原生JS属性） | value = element.get\_property("value") |
| 状态检查 | is\_displayed() | 元素是否可见 | element.is\_displayed() |
| is\_enabled() | 元素是否可操作（启用） | element.is\_enabled() |
| is\_selected() | 元素是否被选中（用于复选框、单选框） | element.is\_selected() |
| 文本内容 | text | 获取元素内部可见文本 | print(element.text) |
| 尺寸位置 | size | 返回元素尺寸，如{'height':20, 'width':100} | print(element.size) |
| location | 返回元素在页面上的位置（x,y坐标） | print(element.location) |
| rect | 返回元素的位置和大小综合信息（x,y,width,height） | print(element.rect) |
| 元素截图 | screenshot(filename) | 截取当前元素的截图 | element.screenshot("element.png") |
| screenshot\_as\_base64 | 获取元素截图的 base64 编码 | img\_base64 = element.screenshot\_as\_base64 |

#### ActionChains

ActionChains 是 Selenium 中用于处理复杂用户交互的一个工具。通过 ActionChains，你可以模拟鼠标和键盘的高级操作。

* **ActionChains** 是 Selenium 提供的一个高级 API。
* 它允许你**模拟复杂的人机交互动作**，比如：
* 鼠标移动（悬停）
* 点击（单击、双击、右击）
* 拖拽（drag and drop）
* 键盘按键（输入、组合键）
* 特点是：**一连串动作可以链式调用、一次性执行**。

##### 基本使用

1. ​**创建动作链对象​​**

​​ 初始化一个动作链，用于记录后续的连续操作。

​​ 方法​​：将 driver 对象传入 ActionChains 构造函数。

from selenium.webdriver.common.action\_chains import ActionChains

# 假设 driver 已初始化

actions = ActionChains(driver)

1. **添加操作到动作链​​**

​​ 通过方法链（Method Chaining）依次添加鼠标、键盘等操作（支持连续调用）。

​​ 方法​​：使用如 click()、send\_keys()、move\_to\_element() 等方法。

# 示例1：鼠标悬停到元素并点击

actions.move\_to\_element(element).click()

# 示例2：输入文本组合操作

actions.send\_keys("Hello").send\_keys(Keys.ENTER)

1. **执行动作链​​**

​​ 触发之前添加的所有操作（操作需在此方法调用后才会实际执行）。

​​ 方法​​：调用 perform() 方法。

##### 方法函数

下面是全面的action chain方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **说明** | **示例** |
| **perform()** | 执行所有已排队的动作 | actions.perform() |
| **reset\_actions()** | 清空已排队的动作（重置队列） | actions.reset\_actions() |
| **click(on\_element=None)** | 点击元素；若不指定，则点击当前鼠标位置 | actions.click(element) |
| **click\_and\_hold(on\_element=None)** | 在元素上按住鼠标左键 | actions.click\_and\_hold(element) |
| **context\_click(on\_element=None)** | 在元素上执行右键点击 | actions.context\_click(element) |
| **double\_click(on\_element=None)** | 在元素上双击 | actions.double\_click(element) |
| **drag\_and\_drop(source, target)** | 从 source 元素拖拽到 target 元素 | actions.drag\_and\_drop(src, tgt) |
| **drag\_and\_drop\_by\_offset(source, xoffset, yoffset)** | 从 source 元素拖拽到相对偏移位置 | actions.drag\_and\_drop\_by\_offset(src, 100, 0) |
| **move\_to\_element(to\_element)** | 将鼠标移动到指定元素中心（悬停） | actions.move\_to\_element(element) |
| **move\_to\_element\_with\_offset(to\_element, xoffset, yoffset)** | 将鼠标移动到元素某一偏移位置 | actions.move\_to\_element\_with\_offset(element, 10, 20) |
| **move\_by\_offset(xoffset, yoffset)** | 将鼠标从当前位置按给定偏移移动 | actions.move\_by\_offset(50, 100) |
| **pause(seconds)** | 在动作链中插入暂停，单位秒 | actions.pause(1.5) |
| **release(on\_element=None)** | 在元素或当前位置释放鼠标左键 | actions.release(element) |
| **key\_down(value, element=None)** | 按住某键（通常用于 Ctrl、Shift、Alt 等） | actions.key\_down(Keys.CONTROL, element) |
| **key\_up(value, element=None)** | 松开某键 | actions.key\_up(Keys.CONTROL, element) |
| \**send\_keys(keys\_to\_send)* | 向当前聚焦元素发送键序（不指定元素） | actions.send\_keys("HELLO") |
| \**send\_keys\_to\_element(element, keys\_to\_send)* | 向指定元素发送键序 | actions.send\_keys\_to\_element(element, "HELLO") |
| **scroll\_to\_element(element)** | 如果元素不在视窗内，将其滚动到视窗底部 | actions.scroll\_to\_element(element) |
| **scroll\_by\_amount(delta\_x, delta\_y)** | 按给定像素量滚动视窗（相对视窗左上角） | actions.scroll\_by\_amount(0, 200) |
| **scroll\_from\_origin(origin, delta\_x, delta\_y)** | 以给定原点（元素中心或视窗左上）为起点滚动 | actions.scroll\_from\_origin(origin, 0, 200) |

##### 注意事项

* **确保调用 perform() 时操作完整**

在调用 perform() 前，确保所有操作已经加入 ActionChains 链式调用。perform() 只能调用一次，调用后无法修改。

* **操作顺序很重要**

ActionChains 中的操作按顺序执行，务必确保顺序正确（如先 move\_to\_element() 再 click()）。

* **元素的可见性和可交互性**

在执行点击、悬停等操作前，确保目标元素是可见且可交互的。可以使用显式等待来确保元素可用。

* **组合键操作需注意释放键**

使用 key\_down() 和 key\_up() 时，务必确保键盘操作的正确性，特别是修饰键（如 Ctrl、Shift）的释放。

* **动态页面操作的等待**

对于动态加载的元素，使用显式等待确保元素在执行操作前可见或可交互，避免操作失败。

* **避免积累过多操作**

在 ActionChains 中不要积累过多的操作，防止内存消耗过高或导致性能问题。

#### 等待机制

Selenium 的等待机制主要是为了解决网页加载、元素出现延迟的问题，防止元素未加载完成就操作导致出错。Selenium 提供了三种等待方式：**强制等待（sleep）**、**隐式等待**、**显式等待**。

##### 强制等待

**强制等待**就是代码执行到 time.sleep() 时，无论页面状态如何，都**强制暂停指定时间**，一动不动，等时间到了再继续。

* 不检查元素状态。
* 浪费时间或等待不足都有可能发生。
* 更像是机械式暂停。

import time

time.sleep(5)  # 暂停执行5秒

特点：

* **简单粗暴**

只管停，啥都不判断

* **容易浪费时间**

元素早就加载好了还继续等

* **容易出错**

如果设置的时间太短，元素还没出现就继续执行，会导致找不到元素错误

* **常用于调试**

在调试阶段插入 sleep，方便观察页面变化

##### 隐式等待

隐式等待是**设置浏览器驱动**，在查找元素时，如果找不到元素，就持续等待一段时间再继续查找，直到：

* 元素找到了（立即继续）
* 超时时间到了（抛出异常）

**注意：**只对找元素（find\_element/find\_elements）有效。

from selenium import webdriver

driver = webdriver.Chrome()

driver.implicitly\_wait(10)  # 最多等10秒，找到了就继续

driver.get("https://example.com")

element = driver.find\_element(By.ID, "target")  # 自动应用隐式等待

element.click()

特点：

* **全局生效**

一旦设置，整个 WebDriver 对象的 find\_element 都带有这个等待时间

* **针对所有元素查找**

每次 find\_element 系列方法自动应用

* **不能指定条件**

只能"存在"就继续，没法像显式等待那样细化条件

* **慎重使用**

与显式等待叠加使用容易出现冲突，导致意外超时

##### 显式等待

显式等待是指：在某一条件成立之前，持续不断地**轮询**（检查），一旦条件成立，就立刻继续执行；如果超时时间到了，条件还不成立，则抛出异常。显示等待一般通过WebDriverWait类和EC条件实现。

**适合用在：**

* 页面中某些元素出现时间不固定
* 元素需要经过一段时间的动态加载或变更

###### WebDriverWait类

WebDriverWait 是 Selenium 中提供的核心类，用来实现显示等待机制。

WebDriverWait(driver, timeout, poll\_frequency=0.5, ignored\_exceptions=None)

**参数说明：**

* **driver：**必须，浏览器驱动对象（ChromeDriver、FirefoxDriver等）
* **timeout：**必须，最大等待时间（秒）
* **poll\_frequency：**检查条件的时间间隔（秒），默认是 0.5 秒。
* **ignored\_exceptions：**遇到哪些异常时继续重试（忽略），而不是直接报错

**类方法：**

* **until(method, message='')：**

持续等待，直到 method 返回 True，返回结果。

* **until\_not(method, message='')：**

持续等待，直到 method 返回 False，返回结果。

**返回值：**

WebDriverWait 的返回值是根据使用的EC 条件决定的。WebDriverWait 本身并不直接决定返回值，而是通过调用不同的期望条件来决定返回的对象类型。

###### expected\_conditions EC条件

定义在selenium.webdriver.support.expected\_conditions 模块的expected\_conditions（简称 EC 条件）是 Selenium 中的一组预定义的条件，主要用于显式等待，用来在执行某些操作之前等待元素满足特定条件，通常与 WebDriverWait 配合使用，确保在执行操作时，元素已满足某种预期状态。

EC条件总表：

|  |  |  |  |
| --- | --- | --- | --- |
| **条件（函数）** | **返回类型** | **说明** | **示例** |
| **presence\_of\_element\_located(locator)** | WebElement | 元素在 DOM 中存在（不一定可见） | EC.presence\_of\_element\_located((By.ID,"u")) |
| **visibility\_of\_element\_located(locator)** | WebElement | False | 元素在 DOM 中可见（displayed 且 width/height>0） | EC.visibility\_of\_element\_located((By.ID,"u")) |
| **visibility\_of\_all\_elements\_located(locator)** | List[WebElement] | 所有匹配 locator 的元素都可见 | EC.visibility\_of\_all\_elements\_located((By.CLASS\_NAME,"item")) |
| **visibility\_of\_any\_elements\_located(locator)** | List[WebElement] | 任意一个匹配 locator 的元素可见 | EC.visibility\_of\_any\_elements\_located((By.TAG\_NAME,"p")) |
| **visibility\_of(element)** | WebElement | 已有元素对象可见 | EC.visibility\_of(elem) |
| **presence\_of\_all\_elements\_located(locator)** | List[WebElement] | 至少一个匹配 locator 的元素出现在 DOM | EC.presence\_of\_all\_elements\_located((By.NAME,"n")) |
| **invisibility\_of\_element\_located(locator)** | bool | 元素在 DOM 中不可见或已移除 | EC.invisibility\_of\_element\_located((By.ID,"loading")) |
| **invisibility\_of\_element(element)** | bool | 已有元素对象不可见或不在 DOM 中 | EC.invisibility\_of\_element(elem) |
| **element\_to\_be\_clickable(locator)** | WebElement | 元素可见且启用，可点击 | EC.element\_to\_be\_clickable((By.ID,"btn")) |
| **element\_to\_be\_selected(element)** | bool | 指定元素已被选中（checkbox/radio） | EC.element\_to\_be\_selected(elem) |
| **element\_located\_to\_be\_selected(locator)** | bool | 匹配 locator 的元素被选中 | EC.element\_located\_to\_be\_selected((By.ID,"chk")) |
| **text\_to\_be\_present\_in\_element(locator, text)** | bool | 指定元素中包含给定文本 | EC.text\_to\_be\_present\_in\_element((By.ID,"msg"),"OK") |
| **text\_to\_be\_present\_in\_element\_value(locator, text)** | bool | 指定元素的 value 属性包含文本 | EC.text\_to\_be\_present\_in\_element\_value((By.ID,"inp"),"abc") |
| **frame\_to\_be\_available\_and\_switch\_to\_it(locator)** | WebDriver | Frame 可切换并自动切换进该 frame | EC.frame\_to\_be\_available\_and\_switch\_to\_it((By.TAG\_NAME,"iframe")) |
| **alert\_is\_present()** | Alert | 弹窗（alert/confirm/prompt）出现 | EC.alert\_is\_present() |
| **title\_is(title)** | bool | 页面标题完全匹配 | EC.title\_is("Home") |
| **title\_contains(substr)** | bool | 页面标题包含子串 | EC.title\_contains("Page") |
| **url\_to\_be(url)** | bool | 当前 URL 完全匹配 | EC.url\_to\_be("https://example.com/") |
| **url\_contains(substr)** | bool | 当前 URL 包含子串 | EC.url\_contains("dashboard") |
| **url\_matches(pattern)** | bool | 当前 URL 符合正则模式 | EC.url\_matches(r"\/users\/\d+$") |
| **staleness\_of(element)** | bool | 元素已从 DOM 移除或失效 | EC.staleness\_of(elem) |
| **number\_of\_windows\_to\_be(count)** | bool | 窗口句柄数量到达指定数 | EC.number\_of\_windows\_to\_be(2) |
| **new\_window\_is\_opened(handles)** | bool | 新窗口/标签打开 | EC.new\_window\_is\_opened(old\_handles) |
| \**any\_of(conditions)* | 任意条件首次满足的返回值 | EC.any\_of(EC.title\_contains("A"), EC.title\_contains("B")) |  |
| \**all\_of(conditions)* | 所有条件均满足才通过 | EC.all\_of(EC.presence\_of(...), EC.element\_to\_be\_clickable(...)) |  |
| \**none\_of(conditions)* | 所有条件都不满足才通过 | EC.none\_of(EC.visibility\_of(...), EC.presence\_of(...)) |  |

any\_of、all\_of、none\_of

* **any\_of：逻辑 OR**
* 复合 EC，等价于逻辑 “OR”​
* 接收多个 EC（Callable），逐一评估。

**返回值**

* 返回**第一个**子条件的非 False 返回值（可为 WebElement、True、Alert 等）。
* 若所有子条件均返回 False，则返回 False​

**示例**

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

from selenium.webdriver.common.by import By

wait = WebDriverWait(driver, 10)

res = wait.until(

    EC.any\_of(

        EC.visibility\_of\_element\_located((By.ID, "menu")),#可能先出现菜单

        EC.alert\_is\_present() # 或先弹出 alert

    )

)

# res 要么是 WebElement，要么是 Alert，要么 False

* 如果菜单先可见，res 是该 WebElement；
* 否则如果 alert 先出现，res 是 Alert 对象；
* 都没出现则超时并抛 TimeoutException​
* **all\_of：逻辑 AND**
* 复合 EC，等价于逻辑 “AND”​
* 仅在所有子条件都返回**非 False** 时才算通过。

**返回值**

* 若全部成立，返回一个列表，其中是**依次各子条件的原始返回值**（可能包含 WebElement、布尔等混合类型）。
* 若有任一子条件返回 False，则整体返回 False​

**示例**

wait = WebDriverWait(driver, 10)

results = wait.until(

    EC.all\_of(

        EC.presence\_of\_element\_located((By.NAME, "username")), # 元素存在

        EC.visibility\_of\_element\_located((By.NAME, "username")),# 可见

        EC.element\_to\_be\_clickable((By.NAME, "username")) # 可点击

    )

)

# results = [WebElement, WebElement, WebElement]

username\_input = results[0]

username\_input.send\_keys("admin")

上例中只有当用户名输入框存在、可见且可点击时才返回三者列表；否则返回 False 并超时​

Selenium

* **none\_of：逻辑 NOT-OR**
* 复合 EC，等价于逻辑 “NOT (OR)”​
* 仅当所有子条件都不成立时才算通过。

**返回值：Bool**

* 全部子条件均返回 False 时，none\_of 返回 True；
* 只要任一子条件返回非 False，none\_of 即返回 False。

**示例**

wait = WebDriverWait(driver, 10)

loaded = wait.until(

    EC.none\_of(

        EC.visibility\_of\_element\_located((By.ID, "loading-spinner")),

        EC.visibility\_of\_element\_located((By.ID, "overlay"))

    )

)

# loaded 为 True，表示加载指示器和遮罩层都消失

用于等待所有加载/遮罩元素都消失，再继续操作​

* **组合与嵌套**
* 任意深度嵌套均可：

cond = EC.any\_of(

    EC.all\_of(condA, condB),

    EC.none\_of(condC, condD)

)

wait.until(cond)

* 返回类型则遵循上述规则：最外层 any\_of 返回第一个子条件的返回值；若该子条件是 all\_of 会先返回列表；若是 none\_of 则返回布尔。

#### 文件操作

##### 下载原理

Selenium 是驱动浏览器去下载的，下载过程完全是浏览器自己完成的。

也就是说：

* Selenium 不会接管下载流
* Selenium 不会自己开线程处理文件流
* Selenium 不会拦截/处理文件内容

它只是模拟人类的操作，比如点击一个下载按钮，真正的下载动作，是浏览器（Chrome、Firefox等）自己用**原生下载机制**完成的。而Playwright是通过拦截下载流自行实现的下载。

##### 下载操作

Selenium的下载操作主要体现在驱动配置中。由于selenium的文件下载是基于浏览器原生操作，所以只需要配置好初始启动配置，然后模拟下载的一些点击操作即可：

* **下载路径：**

文件指定的下载路径，一般通过Options的实验参数prefs设置

* **静默下载：**

因为selenium只能操作浏览器内，而下载窗口属于操作系统级，需要通过静默下载避免窗口弹出

chrome\_options = Options()

prefs = {

    "**download.default\_directory**": r"C:\path\to\your\download",  # 目录

    "**download.prompt\_for\_download**": False, # 禁止弹出下载确认窗口

    "directory\_upgrade": True,     # 下载目录升级（确保能创建目录）

    "safebrowsing.enabled": True,   # 允许安全浏览，防止拦截下载

    "profile.default\_content\_settings.popups": 0   # 禁止弹出窗口

}

chrome\_options.add\_experimental\_option("prefs", prefs)

driver = webdriver.Chrome(options=chrome\_options)

##### 下载检查

Selenium发起下载动作（比如点击按钮）后，不会帮你等待下载完成。

你需要自己用轮询或者sleep去检查目标目录：

* 文件是否出现？
* .crdownload 文件是否消失？
* 文件大小是否稳定？

##### 文件上传

Selenium最常见最常用的上传方式就是通过带有<input type="file">的元素，向其发送文件的绝对路径：

# 定位到文件上传输入框并选择文件

upload\_element = driver.find\_element(By.ID, "hidden-file-upload")

upload\_element.send\_keys("/path/to/your/file.txt")

#### Cookie与会话管理

Selenium中，**Cookie** 和 **Session** 的管理是实现自动化测试时非常重要的部分，尤其是对于模拟用户的登录状态、跨页面操作以及保持会话信息等场景。其中主要以Cookie进行持久化的会话管理。由于Selenium偏向UI，Session管理就是每次启动浏览器示例分配一个Session ID识别会话，维持连接，不包含特定功能。

下面着重介绍Cookies：

* **Cookie操作**

|  |  |  |  |
| --- | --- | --- | --- |
| **操作** | **方法** | **描述** | **示例代码** |
| **获取所有 Cookie** | driver.get\_cookies() | 获取当前页面的所有 **Cookie**。 | cookies = driver.get\_cookies() |
| **获取指定 Cookie** | driver.get\_cookie(name) | 获取指定名称的 **Cookie**。 | cookie = driver.get\_cookie('sessionid') |
| **添加 Cookie** | driver.add\_cookie(cookie\_dict) | 向当前页面添加一个 **Cookie**。cookie\_dict 是一个包含 name、value、path、domain 等信息的字典。 | driver.add\_cookie({'name': 'sessionid', 'value': 'new\_session\_id'}) |
| **删除指定 Cookie** | driver.delete\_cookie(name) | 删除指定名称的 **Cookie**。 | driver.delete\_cookie('sessionid') |
| **删除所有 Cookie** | driver.delete\_all\_cookies() | 删除当前浏览器中所有的 **Cookie**。 | driver.delete\_all\_cookies() |
| **保存 Cookies** | 无 | 将 Cookies 保存到文件或变量中，方便后续加载。 | import pickle with open("cookies.pkl", "wb") as cookie\_file: pickle.dump(driver.get\_cookies(), cookie\_file) |
| **加载 Cookies** | 无 | 从文件中加载保存的 **Cookies**，然后添加到浏览器。 | with open("cookies.pkl", "rb") as cookie\_file: cookies = pickle.load(cookie\_file) for cookie in cookies: driver.add\_cookie(cookie) |

* **使用场景**
* **模拟登录状态**：可以通过保存并重新使用 cookies 来避免每次运行自动化测试时都进行登录操作。
* **模拟用户行为：**可以模拟用户行为，如购物车、浏览历史等，保持用户会话信息。
* **跨页面操作：**跨页面的自动化操作可能需要保留同一会话中的 cookies，例如进行结账等操作时，保持购物车中的商品信息。
* **示例**

# 打开一个网站并登录

driver.get("http://example.com/login")

# 假设这里进行了一些登录操作...

# 获取并保存 cookies

cookies = driver.get\_cookies()

# 关闭浏览器

driver.quit()

# 在之后的测试中，可以将 cookies 加载到浏览器中

driver = webdriver.Chrome()

driver.get("http://example.com")

for cookie in cookies:

    driver.add\_cookie(cookie)

driver.get("http://example.com/dashboard")  # 访问已登录的页面

#### iframe与弹窗处理

##### iframe

iframe（内联框架）是网页中嵌套的另一个 HTML 文档，通常用于在一个页面中加载另一个页面或显示嵌套内容。 iframe 相当一个独立的文档， **Selenium** 在与其交互时，需要先切换到该 iframe 内部进行操作。

* **定位与切换iframe**
* **通过 id 或 name：**

如果 iframe 元素有 id 或 name 属性，你可以通过这两个属性来定位并切换。

driver.switch\_to.frame("frame\_id\_or\_name")

* **通过 WebElement：**

如果你能找到 iframe 元素的 WebElement，那么也可以直接通过 WebElement 切换。

iframe\_element = driver.find\_element(By.ID, "frame\_id")

driver.switch\_to.frame(iframe\_element)

* **通过 索引：**

frame 元素的索引值从 0 开始。如果页面有多个 iframe，你可以通过索引值来选择。

driver.switch\_to.frame(0)  # 切换到第一个iframe

* **操作iframe**

一旦切换到 iframe，与 iframe 内的元素进行交互时，Selenium 会把操作环境认为是 iframe 内部的 DOM。因此，iframe 内的元素定位和主页面是相**互独立**的。

* 要点：
* **进入iframe后，元素定位都会以当前的 iframe 为 相对上下文**
* **内部操作与未进入iframe一致**
* **退出iframe**

如果你想切换回主页面，使用 switch\_to.default\_content()即可

driver.switch\_to.default\_content()  # 切换回主页面（默认内容）

##### 弹窗

弹窗（Pop-up）是指页面上出现的对话框，通常用来显示提示、警告、确认或获取用户输入。Selenium 提供了多种方式来处理不同类型的弹窗，包括 Alert、Confirm、Prompt 和 File Upload 等。

* **弹窗类型**

Selenium 支持几种常见的弹窗类型，每种类型的操作方式有所不同：

* **Alert：**简单的警告弹窗，通常只有一个“确定”按钮。
* **Confirm：**确认弹窗，通常有“确定”和“取消”按钮。
* **Prompt：**提示弹窗，除了“确定”和“取消”按钮外，还允许用户输入文本。
* **File Upload：**文件上传对话框（处理方式稍有不同）。
* **操作弹窗的基本方法**

Selenium通过 switch\_to.alert() 方法切换到当前页面中的弹窗进行操作。

* **操作 Alert 弹窗**

Alert 弹窗通常只有一个“确定”按钮，点击该按钮关闭弹窗。你可以使用 accept() 方法来点击“确定”按钮，或者使用 dismiss() 方法来点击“取消”按钮（如果弹窗有取消按钮）。

# 切换到弹窗

alert = driver.switch\_to.alert()

# 获取弹窗的文本内容

alert\_text = alert.text

print(f"Alert Text: {alert\_text}")

# 点击“确定”按钮，关闭弹窗

alert.accept()  # 或者 alert.dismiss() 点击“取消”

* **操作 Confirm 弹窗**

Confirm 弹窗通常有两个按钮：确定和取消。使用 accept() 来点击“确定”按钮，使用 dismiss() 来点击“取消”按钮。

# 切换到 Confirm 弹窗

confirm = driver.switch\_to.alert()

# 获取弹窗的文本

confirm\_text = confirm.text

print(f"Confirm Text: {confirm\_text}")

# 点击“确定”按钮

confirm.accept()  # 或者 confirm.dismiss() 点击“取消”

* **操作 Prompt 弹窗**

Prompt 弹窗允许用户输入文本。它也有两个按钮：确定 和 取消。你可以通过 send\_keys() 方法输入文本，使用 accept() 或 dismiss() 来处理确认或取消操作。

# 切换到 Prompt 弹窗

prompt = driver.switch\_to.alert()

# 获取弹窗文本

prompt\_text = prompt.text

print(f"Prompt Text: {prompt\_text}")

# 在弹窗中输入文本

prompt.send\_keys("Hello, this is a test.")

# 点击“确定”按钮

prompt.accept()  # 或者 prompt.dismiss() 点击“取消”

* **处理文件上传弹窗**

文件上传弹窗（例如浏览器的文件选择对话框）通常无法直接通过 Selenium 进行交互，因为它属于浏览器外部的操作。Selenium 不提供直接的 API 来操作文件选择对话框，但可以通过以下两种方式解决：

* 使用 send\_keys() 方法将文件路径发送到文件输入框。
* 使用第三方库，如 AutoIT（Windows）或 PyAutoGUI，模拟鼠标点击。
* **多个弹窗的处理**

在 Selenium 中，**弹窗（Alert）** 是自动识别和切换的，在页面上出现多个弹窗时，Selenium 默认不会给你一个直观的方式来区分它们。每次切换到一个弹窗时，switch\_to.alert() 会返回当前活动的弹窗，且它没有区分弹窗的标识或索引。

可以通过下面的方法来区分弹窗进而分别处理：

1. **通过弹窗文本区分弹窗**

每个弹窗都有其独特的 文本内容。你可以通过 alert.text 获取弹窗的文本，从而判断不同弹窗的类型或内容。

1. **弹窗的出现顺序**

你可以假设弹窗的出现顺序是固定的，也就是说，最先出现的弹窗是第一个弹窗，接着是第二个，以此类推。通过 while 循环和显式等待，你可以依次处理每个弹窗，并通过文本判断它们的类型。

1. **结合弹窗出现的时间**

有时，弹窗的出现顺序可以作为判断依据。你可以记录每个弹窗出现的时间或顺序，然后根据弹窗的顺序判断。

1. **自定义弹窗标识**

如果你有控制页面的能力，最好在页面上为每个弹窗设置一些特定的标识或 ID，比如在弹窗的文本中加入特定的标记。这使得你可以通过特定的标识来判断弹窗的类型，而不是仅依赖文本内容。

**例如：**

* 弹窗类型 **Confirm** 的文本中含有“确认”
* 弹窗类型 **Alert** 的文本中含有“警告”
* 弹窗类型 **Prompt** 的文本中含有“输入”

对于需要在多个浏览器和操作系统组合上进行跨平台和跨浏览器的自动化测试非常有用。

### Playwright

基础介绍见[playwright](#_Playwright)，下面为深入具体功能介绍

#### Playwright原理

Playwright 直接控制浏览器引擎（如 Chromium 的 Blink），绕过浏览器界面层。通过 DevTools 协议发送指令（如修改网络设置、模拟设备参数），实现底层操作

##### 架构设计原理

* ​​**三层架构模型​​**

Playwright 的架构由三个核心层构成：

* ​​**API 层​​：**提供统一的多语言接口（Python/JS/Java 等），抽象浏览器操作指令为标准化调用。开发者通过简洁的 API 控制浏览器行为，如 page.click() 或 page.fill()。
* ​​**驱动层​​：**内置浏览器专用驱动（Chromium/Firefox/WebKit），将 API 指令转换为浏览器引擎可识别的底层协议（如 Chrome DevTools Protocol）。驱动层直接与浏览器进程通信，无需第三方中间件。
* ​​**浏览器进程​​：**启动独立的浏览器实例，通过进程隔离确保测试稳定性。每个测试用例在独立的浏览器上下文中运行，避免环境污染。
* ​​**跨浏览器兼容性实现​​**

通过统一 API 适配不同浏览器引擎的差异。例如，page.screenshot() 在 Chromium 和 WebKit 中调用不同的底层实现，但对开发者透明。

##### 通信机制与执行流程

* ​​**WebSocket 与 JSON-RPC 协议​​**

Playwright 使用 WebSocket 建立与浏览器的长连接，并通过 JSON-RPC 传输序列化指令。相较于 Selenium 的 HTTP 轮询机制，这种设计减少延迟并支持双向实时通信。

* ​​**进程外执行模型​​**

浏览器进程与测试脚本运行在不同的操作系统进程中，通过 IPC（进程间通信）传递数据。这种设计避免了传统工具因共享内存导致的崩溃问题，同时支持多浏览器并行测试。

简单说，Playwright 的核心原理是通过 ​**​WebSocket长连接 + CDP（Chrome DevTools Protocol）接口+JSON-RPC传输​**​ 直接连接浏览器原生接口，从而避免繁琐的中间层而提升效率。

#### 层次资源管理

Playwright 采用 **浏览器 (Browser)**、**浏览器上下文 (Context)** 和 **页面 (Page)** 三个层次来进行资源管理，这种层次化的设计有助于更细粒度地控制和隔离测试环境。

* **Browser（浏览器层）**
* 表示一个浏览器实例，负责启动、关闭浏览器以及管理多个浏览器上下文（Context）。
* 用于创建浏览器上下文、启动和关闭浏览器。
* **Context（浏览器上下文层）**
* 表示一个独立的浏览器环境，隔离 cookies、缓存等数据。
* 用于管理独立会话，创建页面（Page）并隔离不同测试环境。
* **Page（页面层）**
* 表示浏览器中的一个标签页，允许与页面内容进行交互。
* 用于执行页面操作，如加载 URL、点击元素、截屏等。

##### Browser层

###### Browser层的职责

在 Playwright 中，Browser 对象负责管理整个浏览器实例的生命周期。它的主要职责包括：

1. **启动与关闭浏览器实例**：

通过 launch() 启动浏览器，使用 close() 关闭浏览器实例。

1. **管理浏览器上下文（BrowserContext）**：

BrowserContext 用于隔离不同的会话，每个上下文可以有独立的 cookies、缓存等，多个上下文可以共存于一个浏览器实例中。

1. **创建页面（Page）：**

通过 new\_page() 方法在浏览器中创建新页面，每个页面代表一个标签页，可以执行浏览、点击、输入等操作。

1. **浏览器配置：**

在启动时，可以通过参数配置浏览器行为，如是否启用无头模式、设置用户代理、代理服务器等。

1. **事件管理：**

Browser 监听浏览器相关的事件（如崩溃、断开连接等），有助于增强脚本的稳定性。

###### Launch（）参数列表

**launch()** 函数是 Playwright 中启动浏览器实例的核心方法。通过它，你可以指定各种浏览器启动参数，定制浏览器的行为。launch() 方法用于创建一个新的浏览器实例并且进行各种配置。

参数列表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **​参数​​** | **​​类型/默认值​​** | **​​描述​​** | **​​浏览器兼容性​​** | **​​示例​​** |
| headless | bool (默认 True) | 是否以无头模式（无界面）启动浏览器。 | 全浏览器支持 | headless=False 显示浏览器界面 |
| args | List[str] | 传递给浏览器的命令行参数，用于配置浏览器行为。 | 全浏览器支持 | args=['--start-maximized', '--disable-blink-features=AutomationControlled'] |
| executable\_path | str | 指定浏览器可执行文件路径（如自定义安装的 Chrome）。 | 全浏览器支持 | executable\_path=r"C:\Chrome\chrome.exe" |
| proxy | dict | 设置代理服务器，支持 HTTP 和 SOCKS（全局代理）。 | 全浏览器支持 | proxy={"server": "http://myproxy.com:3128"} |
| slow\_mo | int (毫秒) | 操作间延迟，用于调试（如观察自动化执行过程）。 | 全浏览器支持 | slow\_mo=1000 延迟 1 秒 |
| channel | str | 指定浏览器类型（仅限 Chromium 系），如 "chrome"、"msedge"。 | Chromium 系 | channel="chrome" 使用本地安装的 Chrome |
| ignore\_default\_args | List[str] | 忽略浏览器默认启动参数（如禁用自动化标识）。 | Chromium 系 | ignore\_default\_args=['--enable-automation'] |
| timeout | int (毫秒) | 浏览器启动超时时间。 | 全浏览器支持 | timeout=30000 超时 30 秒 |
| ​**​Chromium 独有参数​**​ |  |  |  |  |
| chromium\_sandbox | bool | 禁用 Chromium 沙盒（需管理员权限）。 | Chromium 系 | chromium\_sandbox=False |

###### 其它浏览器启动方法

* **launch\_persistent\_context（）：**

launch\_persistent\_context() 是 Playwright 提供的一个特殊方法，用于启动一个 持久化浏览器上下文，它与常规的 launch() 和 new\_context() 方法不同，专门用于创建带有持久化存储（如用户数据、会话、Cookies 等）的上下文。

* **核心参数**
* user\_data\_dir, 一个持久化的目录路径，类似于浏览器初始化用的User Data目录
* options，同launch（）
* **应用场景**
* **模拟长期用户会话**

模拟用户多个会话中维持登录状态。

* **测试需要持续数据的场景**

持久化上下文来保留数据。

* **多测试用例共享浏览器状态**

在多个测试用例之间共享浏览器状态，避免每个测试用例都重新初始化浏览器或进行登录。

* **需要启动多个浏览器操作**

当需要同时启用多个同类（edge、chrome这种）浏览器，需要分别使用不同的user data，否则会导致资源冲突，因为都使用默认的User Data

* **连接浏览器**
* **connect()**
* 通过 connect() 方法，可以连接到一个已启动的浏览器实例。这个方法允许在远程机器或容器中启动浏览器实例后，通过 WebSocket 连接进行控制。
* 适用于分布式环境或需要远程控制浏览器的场景。

browser = playwright.chromium.connect("ws://localhost:9222")

* **connect\_over\_cdp()**
* connect\_over\_cdp() 通过 Chrome DevTools Protocol（CDP）连接到一个 Chrome 或 Chromium 实例。通过 CDP，你可以执行一些浏览器级别的操作，如调试、网络请求拦截等。
* 适用于需要使用 CDP 提供的高级调试功能时。

browser = playwright.chromium.connect\_over\_cdp("ws://localhost:9222")

###### browser对象常用方法

|  |  |  |  |
| --- | --- | --- | --- |
| **​属性/方法​​** | **​​类型/返回值​​** | **​​描述​​** | **​​示例​​** |
| ​**​属性​**​ |  |  |  |
| browser\_type | BrowserType | 返回浏览器类型（如 chromium, firefox, webkit）。 | browser.browser\_type.name # 输出 'chromium' |
| version() | str | 获取浏览器版本号。 | print(browser.version()) # 输出 'Chrome/120.0.0.0' |
| is\_connected() | bool | 检查浏览器实例是否仍处于连接状态（未被关闭）。 | if browser.is\_connected(): ... |
| ​**​方法​**​ |  |  |  |
| new\_page() | Page | 创建一个新页面（Page 对象），并自动关联到默认的浏览器上下文。 | page = browser.new\_page() |
| new\_context() | BrowserContext | 创建一个新的浏览器上下文（隔离的会话环境，如无痕模式）。 | context = browser.new\_context(user\_agent='自定义UA') |
| contexts | List[BrowserContext] | 返回所有已打开的浏览器上下文列表。 | for ctx in browser.contexts: print(ctx) |
| close() | None | 关闭浏览器实例及其所有页面，释放资源。 | await browser.close()（异步）或 browser.close()（同步） |
| on('disconnected') | 事件监听 | 监听浏览器断开连接事件（如手动关闭或崩溃）。 | browser.on('disconnected', lambda: print('浏览器已断开')) |
| ​**​Chromium 独有方法​**​ |  |  |  |
| new\_browser\_cdp\_session() | CDPSession | 创建与 Chromium 的 CDP（DevTools 协议）会话，用于底层调试。 | cdp\_session = browser.new\_browser\_cdp\_session() |

##### Context层

BrowserContext 对象是一个用于隔离多个会话的环境，它代表了一个浏览器的独立实例，提供了隔离的 **cookies**、缓存、存储等数据。每个 BrowserContext 中可以有多个页面（Page 对象），它是实现多用户测试、模拟不同用户状态等场景的关键。

###### BrowserContext 对象的职责

* **会话隔离**：

每个上下文是独立的，浏览器上下文之间不会共享 cookies、localStorage、sessionStorage 等。适用于模拟多个用户在同一浏览器实例中的操作。

* **管理会话状态**：

可以为不同的浏览器上下文配置不同的代理、用户代理、视口大小等设置。

* **页面管理**：

一个 BrowserContext 可以创建多个页面，每个页面代表一个独立的浏览器标签。

###### BrowserContext 对象获取

BrowserContext 是通过 browser.new\_context() 或 browser.contexts() 获取的，contexts（）主要为获取当前已经打开的上下文，下面重点介绍new\_context()：

|  |  |  |  |
| --- | --- | --- | --- |
| **参数​​** | **​​类型/默认值​​** | **​​描述​​** | **​​示例/来源​​** |
| viewport | dict | 设置视口尺寸（宽高），默认 {"width": 1280, "height": 720}。 | viewport={"width": 1920, "height": 1080} |
| no\_viewport | bool (默认 False) | 禁用默认视口，允许浏览器窗口自由调整大小（需配合 args 最大化参数）。 | no\_viewport=True 结合 args=['--start-maximized'] 实现全屏 |
| user\_agent | str | 自定义用户代理字符串。 | user\_agent="Mozilla/5.0 (自定义 UA)" |
| proxy | dict | 上下文级别的代理设置（覆盖 launch() 的全局代理）。 | proxy={"server": "socks5://127.0.0.1:8888"} |
| storage\_state | dict/str | 加载本地存储状态（Cookies、LocalStorage），支持文件路径或字典。 | storage\_state="cookies.json" 加载预存会话 |
| base\_url | str | 设置基准 URL，允许在 page.goto() 中使用相对路径。 | base\_url="https://example.com"，page.goto("/path") 自动补全 |
| ignore\_https\_errors | bool (默认 False) | 忽略 HTTPS 证书错误（如测试环境）。 | ignore\_https\_errors=True |
| permissions | List[str] | 授予权限（如 ["geolocation", "microphone"]）。 | permissions=["geolocation"] 允许定位 |
| ​**​高级参数​**​ |  |  |  |
| color\_scheme | str | 模拟系统配色模式（如 "dark"、"light"）。 | color\_scheme="dark" |
| device\_scale\_factor | float | 设置设备像素比（如 2.0 适配高分辨率屏幕）。 | device\_scale\_factor=2.0 |
| ​**​视频录制参数​**​ |  |  |  |
| record\_video\_dir | str | 录制视频的保存路径。 | record\_video\_dir="videos/" |
| record\_video\_size | dict | 指定视频分辨率（如 {"width": 640, "height": 480}）。 | record\_video\_size={"width": 1280, "height": 720} |

###### BrowserContext 对象的属性和方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类别​​** | **​​名称​​** | **​​类型/返回值​​** | **​​描述​​** | **​​示例​​** |
| ​**​属性​**​ |  |  |  |  |
|  | browser | Browser | 返回创建该上下文的浏览器实例。 | context.browser.close() |
|  | pages | List[Page] | 获取该上下文中所有已打开的页面列表。 | for page in context.pages: print(page.url) |
| ​**​方法​**​ |  |  |  |  |
| ​**​页面管理​**​ | new\_page() | Page | 创建并返回一个新页面。 | page = context.new\_page() |
|  | expect\_page() | EventContextManager | 监听新页面的创建（如点击链接触发新标签页）。 | with context.expect\_page() as new\_page: <触发新页面的操作> |
| ​**​存储管理​**​ | storage\_state() | dict | 返回当前上下文的存储状态（Cookies、LocalStorage）。可保存为文件。 | state = context.storage\_state(path="state.json") |
|  | add\_cookies() | None | 向上下文添加一组 Cookie。 | context.add\_cookies([{"name": "user", "value": "test"}]) |
|  | clear\_cookies() | None | 清除所有 Cookie。 | context.clear\_cookies() |
| ​**​网络控制​**​ | route() | None | 拦截和修改网络请求（如屏蔽图片、修改响应）。 | context.route("\*\*/\*.png", lambda route: route.abort()) |
| ​**​权限管理​**​ | grant\_permissions() | None | 授予权限（如定位、通知）。 | context.grant\_permissions(["geolocation"]) |
|  | clear\_permissions() | None | 清除所有授予的权限。 | context.clear\_permissions() |
| ​**​生命周期​**​ | close() | None | 关闭上下文及其所有页面，释放资源。 | context.close() |
| ​**​事件监听​**​ | on('page') | None | 监听新页面创建事件。 | context.on("page", lambda page: print("新页面:", page.url)) |
| ​**​模拟功能​**​ | set\_geolocation() | None | 设置模拟的地理位置。 | context.set\_geolocation({"latitude": 37.7749, "longitude": -122.4194}) |
|  | set\_offline() | None | 模拟离线网络环境。 | context.set\_offline(True) |

##### Page层

Page 层代表浏览器上下文（Context）中的一个​**​独立标签页​**​或​**​弹出窗口​**​，是用户与网页交互的最小单元。一般通过context的new\_page()创建page对象，new\_page() 方法本身并没有太多直接的配置项，主要是通过 context 创建 page，而 page 在创建时会继承自 context 的一些全局配置。

###### 职责

page 层主要负责处理页面级的任务，包括但不限于：

1. **页面交互**：模拟用户的行为，如点击、输入文本、滚动、拖动等。
2. **获取页面信息**：获取页面的 URL、标题、截图、PDF 等内容。
3. **网络监控**：拦截、修改、等待网络请求和响应。
4. **执行 JavaScript**：在页面中执行 JavaScript 代码。
5. **管理元素**：查找、操作 DOM 元素，等待特定元素的出现或状态变化。
6. **文件下载与上传**：处理页面中的文件下载与上传事件。

###### Page对象属性方法

**核心属性**

|  |  |  |
| --- | --- | --- |
| **​​属性​​** | **​​作用​​** | **​​示例/说明​​** |
| content() | 获取当前页面的完整 HTML 内容 | html = page.content() |
| title() | 获取页面标题 | title = page.title() |
| url | 当前页面的 URL | 动态更新，适用于导航监控 |
| main\_frame | 获取页面的主 Frame 对象 | 处理多 Frame 页面时操作主文档内容 |
| context | 返回关联的 BrowserContext 对象 | 用于访问上下文级配置（如 Cookies、代理） |
| is\_closed() | 判断页面是否已关闭 | if page.is\_closed(): ... |

**核心方法：**后面还会详细说明

|  |  |  |  |
| --- | --- | --- | --- |
| **​​功能分类​​** | **方法​​** | **​​功能描述​​** | **​​示例/参数说明​​** |
| ​**​导航控制​**​ | goto(url, \*\*kwargs) | 跳转到指定 URL，支持等待策略（如 networkidle）和超时设置。 | await page.goto("https://example.com", wait\_until="networkidle") |
| ​**​导航控制​**​ | go\_back() | 后退到浏览器历史记录的上一页。 | await page.go\_back() |
| ​**​导航控制​**​ | go\_forward() | 前进到浏览器历史记录的下一页。 | await page.go\_forward() |
| ​**​导航控制​**​ | reload(\*\*kwargs) | 刷新当前页面。 | await page.reload(timeout=30000) |
| ​**​元素定位​**​ | locator(selector) | 返回元素的定位器（Locator 对象），支持链式操作。 | button = page.locator("button:has-text('Submit')") |
| ​**​元素定位​**​ | get\_by\_text(text) | 按文本内容定位元素（支持精确或模糊匹配）。 | link = page.get\_by\_text("Login", exact=True) |
| ​**​元素定位​**​ | get\_by\_role(role) | 按 ARIA 角色定位元素（如 button/link）。 | button = page.get\_by\_role("button", name="Submit") |
| ​**​元素操作​**​ | click(selector) | 点击指定元素，自动等待元素可点击。 | await page.click("button#submit") |
| ​**​元素操作​**​ | fill(selector, value) | 向输入框填充文本。 | await page.fill("input#username", "user123") |
| ​**​元素操作​**​ | select\_option(selector, \*\*kwargs) | 选择下拉框选项（支持按值/标签/索引）。 | await page.select\_option("select#country", value="CN") |
| ​**​元素操作​**​ | drag\_and\_drop(source, target) | 拖放元素。 | await page.drag\_and\_drop("#item", "#target") |
| ​**​等待机制​**​ | wait\_for\_selector(selector) | 等待元素出现并可见。 | await page.wait\_for\_selector(".loading", state="hidden") |
| ​**​等待机制​**​ | wait\_for\_load\_state(state) | 等待页面加载至指定状态（如 domcontentloaded）。 | await page.wait\_for\_load\_state("networkidle") |
| ​**​事件处理​**​ | wait\_for\_event(event) | 监听特定事件（如 dialog/download）。 | page.on("dialog", lambda dialog: dialog.accept()) |
| ​**​事件处理​**​ | expect\_popup() | 等待新标签页或弹窗打开，需配合 async with 使用。 | async with page.expect\_popup() as popup\_info: ... |
| ​**​资源管理​**​ | screenshot(\*\*kwargs) | 截取页面或元素截图。 | await page.screenshot(path="screenshot.png", full\_page=True) |
| ​**​资源管理​**​ | pdf(\*\*kwargs) | 生成页面 PDF。 | await page.pdf(path="report.pdf", format="A4") |
| ​**​网络控制​**​ | route(url, handler) | 拦截和修改网络请求。 | await page.route("\*\*/\*.png", lambda route: route.abort()) |
| ​**​脚本执行​**​ | evaluate(script) | 在页面上下文中执行 JavaScript，返回结果。 | title = await page.evaluate("() => document.title") |
| ​**​脚本执行​**​ | expose\_function(name, func) | 向页面注入自定义函数，实现双向通信。 | await page.expose\_function("md5", lambda data: hashlib.md5(data).hexdigest()) |
| ​**​环境模拟​**​ | set\_viewport\_size(width, height) | 设置视口尺寸。 | await page.set\_viewport\_size(1280, 720) |

#### 元素定位于操作

Playwright的定位器核心有三种：Locator、selector、page；定位方式最常用为：Xpath、CSS选择器；其中，定位器一般共享定位方式，也就是定位表达式一般是通用的。

##### 定位方式

元素的定位是通过不同的选择器表达方式来实现的。这些表达方法提供了多种方式来选择页面上的元素，从简单的 CSS 选择器到复杂的 XPath 表达式，每种方法都有不同的适用场景。

###### CSS选择器

基础选择器（Basic Selectors）

|  |  |  |  |
| --- | --- | --- | --- |
| **​选择器类型​​** | **​​语法​​** | **​​作用​​** | **​​示例​​** |
| ​**​元素选择器​**​ | tag | 按标签名选择元素 | div → 所有 <div> 元素 |
| ​**​类选择器​**​ | .class | 按 class 属性选择元素 | .submit-btn → class="submit-btn" 的元素 |
| ​**​ID 选择器​**​ | #id | 按 id 属性选择元素 | #header → id="header" 的元素 |
| ​**​通用选择器​**​ | \* | 选择所有元素 | \* → 页面中所有元素 |
| ​**​属性选择器​**​ | [attribute] | 按属性存在性选择元素 | [disabled] → 所有含 disabled 属性的元素 |

组合选择器（Combinators）

|  |  |  |  |
| --- | --- | --- | --- |
| **符号​​** | **​​语法​​** | **​​作用​​** | **​​示例​​** |
| ​**​后代选择器​**​ | A B | 选择 A 的所有后代 B（不限层级） | div span → div 内的所有 <span> |
| ​**​子元素选择器​**​ | A > B | 选择 A 的直接子元素 B | ul > li → <ul> 的直接子 <li> |
| ​**​相邻兄弟选择器​**​ | A + B | 选择紧接在 A 后的第一个兄弟 B | h1 + p → 紧接 <h1> 后的第一个 <p> |
| ​**​通用兄弟选择器​**​ | A ~ B | 选择 A 之后的所有兄弟 B | h2 ~ p → 所有在 <h2> 后的 <p> |

属性选择器（Attribute Selectors）

|  |  |  |
| --- | --- | --- |
| **​语法​​** | **​​作用​​** | **​​示例​​** |
| [attribute="value"] | 属性值完全匹配 | [type="text"] → type="text" 的元素 |
| [attribute^="value"] | 属性值以 value 开头 | [href^="https"] → href 以 https 开头的元素 |
| [attribute$="value"] | 属性值以 value 结尾 | [src$=".png"] → src 以 .png 结尾的元素 |
| [attribute\*="value"] | 属性值包含 value 子字符串 | [class\*="error"] → class 含 error 的元素 |
| [attribute~="value"] | 属性值包含以空格分隔的完整单词 value | [class~="active"] → class 包含独立单词 active 的元素 |
| [attribute|="value"] | 属性值以 value 开头且紧随连字符或为空 | [lang|="en"] → lang 为 en 或 en-US 的元素 |

伪类（Pseudo-classes）​

|  |  |  |
| --- | --- | --- |
| **伪类​​** | **​​作用​​** | **​​示例​​** |
| :hover | 鼠标悬停状态的元素 | a:hover → 鼠标悬停的链接 |
| :focus | 获得焦点的元素 | input:focus → 当前聚焦的输入框 |
| :first-child | 父元素的第一个子元素 | li:first-child → 列表的第一个项 |
| :last-child | 父元素的最后一个子元素 | div:last-child → 父容器最后一个 <div> |
| :nth-child(n) | 父元素的第 n 个子元素（支持公式） | tr:nth-child(2n) → 所有偶数行 |
| :not(selector) | 排除匹配 selector 的元素 | input:not([disabled]) → 非禁用的输入框 |

伪元素（Pseudo-elements）​

|  |  |  |
| --- | --- | --- |
| **伪元素​​** | **​​作用​​** | **​​示例​​** |
| ::before | 在元素内容前插入生成内容 | p::before { content: "→ "; } |
| ::after | 在元素内容后插入生成内容 | a::after { content: " ↗"; } |
| ::first-line | 选择元素的第一行文本 | p::first-line { color: red; } |
| ::first-letter | 选择元素的第一个字母 | h1::first-letter { font-size: 2em; } |

###### Xpath

XPath（XML Path Language）是一种用于在XML文档中查找信息的语言，它可以通过路径表达式来选择节点。而且用浏览器F12获取xpath是比较方便的。

基本语法结构为

* **节点选择​​**

|  |  |  |
| --- | --- | --- |
| **​​表达式​​** | **​​说明​​** | **​​示例​​** |
| nodename | 选择当前节点的子节点 | div → 所有 <div> 子元素 |
| / | 从根节点开始选择（绝对路径） | /html/body/div → 根节点下的 <div> |
| // | 从任意位置选择节点（相对路径） | //div → 文档中所有 <div> 元素 |
| . | 当前节点 | ./span → 当前节点的子 <span> |
| .. | 父节点 | //input/.. → <input> 的父元素 |
| @ | 选择属性 | //img/@src → 所有 <img> 的 src 属性 |

* **谓语（Predicates）​​**

用于过滤符合条件的节点，用 [] 包裹逻辑表达式：

//div[@id="header"]         // 选择 id 为 "header" 的 div

//ul/li[1]                  // 选择第一个 <li> 子元素

//input[contains(@class, "error")]  // class 属性包含 "error" 的输入框

* ​​**通配符​​**

​​ 表示任意的节点获知属性

|  |  |  |
| --- | --- | --- |
| **​通配符​​** | **​​作用​​** | **​​示例​​** |
| \* | 匹配任意元素节点 | //div/\* → div 的所有子元素 |
| @\* | 匹配任意属性 | //a[@\*] → 所有带属性的 <a> |

逻辑运算符与函数​​

* ​​**逻辑运算符​​**

|  |  |  |
| --- | --- | --- |
| ​**​运算符​​** | **​​说明​​** | **​​示例​​** |
| and | 逻辑与 | //div[@id="main" and @class="content"] |
| or | 逻辑或 | //input[@type="text" or @type="password"] |
| not() | 逻辑非 | //div[not(@class)] → 无 class 属性的 div |

* ​​**常用函数​​**

|  |  |  |
| --- | --- | --- |
| ​​**函数​​** | **​​作用​​** | **​​示例​​** |
| text() | 匹配元素文本内容 | //button[text()="Submit"] |
| contains(str1, str2) | 字符串包含 | //a[contains(text(), "Next")] |
| starts-with(str1, str2) | 字符串以某子串开头 | //input[starts-with(@id, "user")] |
| string-length(text) | 文本长度 | //p[string-length(text()) > 100] |
| position() | 节点位置 | //tr[position()=2] → 第二个 <tr> |

轴（Axes）​​

XPath 通过轴定义节点与其他节点的关系，语法为 **轴名::节点**。

|  |  |  |
| --- | --- | --- |
| **轴​​** | **​​说明​​** | **​​示例​​** |
| child | 当前节点的子节点 | //div/child::span → div 的直接子 <span> |
| parent | 当前节点的父节点 | //input/parent::form → <input> 的父表单 |
| ancestor | 当前节点的所有祖先节点 | //span/ancestor::div → <span> 的所有上级 div |
| descendant | 当前节点的所有后代节点 | //div/descendant::p → div 的所有后代 <p> |
| following-sibling | 当前节点之后的所有同级节点 | //li/following-sibling::li → 后续所有 <li> |
| preceding-sibling | 当前节点之前的所有同级节点 | //td/preceding-sibling::td → 前面的所有 <td> |
| attribute | 当前节点的属性 | //img/attribute::src → <img> 的 src 属性 |

##### Locator

Locator 是 **Playwright** 提供的一种非常强大的元素定位方式，Locator 为了 **更高效、更可靠地操作页面元素** 而设计的，它提供了 **自动等待**、**元素重试**、**状态检测** 等功能，使得自动化脚本在应对动态页面时更为稳定。

Locator是**最强大，最推荐使用**的定位器。

###### Locator对象

在 Playwright 中，Locator 是 ​**​最核心的元素操作对象​**​，它封装了元素定位、自动等待、链式操作等关键功能。

**Locator 的核心特性**

1. ​​**延迟执行 (Lazy Evaluation)​​：**

创建 Locator 时不会立即查找元素，直到执行操作（如 click()）时才真正定位

1. **​​自动等待 (Auto-Waiting)​​：**

所有操作自带 30 秒等待（元素可见/可操作）

1. **​​​​链式操作 (Chaining)​​：**

支持连续调用多个方法​​，形成操作流水线

1. **​​元素复用 (Reusability)​​：**

同一个 Locator 可多次使用，避免重复编写选择器

###### 创建locator

通过 page.locator() 方法，传入xpath、CSS选择器或其它选择字符，来创建一个 Locator 对象。此外，还可以从locator获取locator、从frame获取locator。

###### Locator操作

下面是locator对象的常用属性方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **分类​​** | **​​方法​​** | **​​作用​​** | **​​参数示例​​** | **​​代码示例​​** |
| ​**​基础交互​**​ | click(options) | 点击元素 | timeout=5000, force=True, delay=100 | page.locator("#submit").click(timeout=5000) |
| dblclick(options) | 双击元素 | button="right" | page.locator(".file").dblclick() |
| fill(value, options) | 填充输入框 | value="admin" | page.locator("#username").fill("admin") |
| type(text, options) | 模拟键盘输入 | text="Hello", delay=50 | page.locator("#input").type("Playwright") |
| press(key, options) | 触发键盘事件 | key="Enter" | page.locator("#search").press("Enter") |
| hover(options) | 鼠标悬停 | timeout=3000 | page.locator(".menu").hover() |
| ​**​状态判断​**​ | is\_visible() | 元素是否可见 | - | if locator.is\_visible(): print("可见") |
| is\_hidden() | 元素是否隐藏 | - | assert locator.is\_hidden() |
| is\_enabled() | 元素是否可操作 | - | if locator.is\_enabled(): locator.click() |
| is\_checked() | 复选框是否选中 | - | if locator.is\_checked(): print("已选中") |
| ​**​内容获取​**​ | inner\_text() | 获取元素文本 | - | text = locator.inner\_text() |
| inner\_html() | 获取元素HTML | - | html = locator.inner\_html() |
| all\_inner\_texts() | 获取所有匹配元素的文本列表 | - | texts = locator.all\_inner\_texts() |
| get\_attribute(name) | 获取元素属性值 | name="href" | href = locator.get\_attribute("href") |
| ​**​元素过滤​**​ | first | 第一个匹配元素 | - | locator.first.click() |
| last | 最后一个匹配元素 | - | locator.last.hover() |
| nth(index) | 第 N 个元素（从 0 开始） | index=2 | locator.nth(2).click() |
| Count() | 获取元素个数 |  | records\_count = self.page.locator('//\*[@id="kjnum"]/tr').count() |
| filter(condition) | 按条件过滤元素 | has\_text="OK", has=other\_locator | locator.filter(has\_text="Active").click() |
| ​**​高级操作​**​ | scroll\_into\_view\_if\_needed() | 滚动到元素可见区域 | - | locator.scroll\_into\_view\_if\_needed() |
| screenshot(options) | 截取元素区域 | path="element.png", type="png" | locator.screenshot(path="chart.png") |
| drag\_to(target) | 拖拽到目标元素 | target=other\_locator | locator.drag\_to(target\_locator) |
| set\_input\_files(files) | 上传文件 | files="file.txt" 或 files=["file1.txt", "file2.png"] | locator.set\_input\_files("avatar.png") |
| ​**​等待方法​**​ | wait\_for(options) | 等待元素状态 | state="visible", timeout=10000 | locator.wait\_for(state="hidden") |
| wait\_for\_selector(selector) | 等待子元素出现 | selector=".child", state="attached" | locator.wait\_for\_selector(".loading") |

* ​**​通用参数​**​：
  + timeout：操作超时时间（默认 30 秒）
  + force：强制操作（绕过可见性检查）
  + state：元素状态（visible/hidden/attached/detached）
  + has\_text：过滤包含指定文本的元素
  + has：基于其他 Locator 过滤元素

##### Query Selector

在playwright中，query\_selector(),query\_selector\_all()也是常用的定位器，相比于locator

###### ElementHandle对象

通过query\_selector(),query\_selector\_all()定位元素后会获取到ElementHandle对象，区别是前者只会匹配第一个，后者得到匹配到的元素列表。其中，**ElementHandle**​表示页面中某个**特定的 DOM 元素**。

**特点**：​一旦获取，ElementHandle 就与该元素绑定，即使页面发生变化，它仍指向原始元素。

###### ElementHandle操作

ElementHandle的元素操作于locator基本一致，下面尤其展示与locator的区别

|  |  |  |  |
| --- | --- | --- | --- |
| **属性/方法​​** | **​​作用​​** | **​​与 Locator 的异同​​** | **​​示例/说明​​** |
| ​**​click()​**​ | 点击元素。 | ​**​相同操作​**​，但 ElementHandle 直接操作现有元素，​**​不自动等待可点击性​**​（需手动处理）。 | element\_handle.click() |
| ​**​dblclick()​**​ | 双击元素。 | ​**​相同操作​**​，差异同 click()。 | element\_handle.dblclick() |
| ​**​fill(value)​**​ | 填充输入框。 | ​**​相同操作​**​，但 ElementHandle ​**​不验证元素是否为输入框​**​（可能直接报错）。 | element\_handle.fill("text") |
| ​**​hover()​**​ | 悬停元素。 | ​**​相同操作​**​，差异同 click()。 | element\_handle.hover() |
| ​**​is\_visible()​**​ | 判断元素是否可见。 | ​**​相同操作​**​，但 ElementHandle ​**​直接检查当前状态​**​（可能已失效）。 | if element\_handle.is\_visible(): ... |
| ​**​is\_enabled()​**​ | 判断元素是否启用。 | ​**​相同操作​**​，差异同 is\_visible()。 | enabled = element\_handle.is\_enabled() |
| ​**​is\_checked()​**​ | 判断复选框是否选中。 | ​**​相同操作​**​，差异同 is\_visible()。 | checked = element\_handle.is\_checked() |
| ​**​get\_attribute()​**​ | 获取元素属性值。 | ​**​相同操作​**​，但 ElementHandle ​**​直接访问当前元素​**​（可能值已过期）。 | href = element\_handle.get\_attribute("href") |
| ​**​text\_content()​**​ | 获取元素文本内容。 | ​**​相同操作​**​，差异同 get\_attribute()。 | text = element\_handle.text\_content() |
| ​**​inner\_html()​**​ | 获取元素内部 HTML。 | ​**​相同操作​**​，差异同 get\_attribute()。 | html = element\_handle.inner\_html() |
| ​**​inner\_text()​**​ | 获取元素可见文本。 | ​**​相同操作​**​，差异同 get\_attribute()。 | text = element\_handle.inner\_text() |
| ​**​evaluate()​**​ | 在元素上下文中执行 JavaScript。 | ​**​相同操作​**​，但 ElementHandle ​**​直接绑定到元素​**​，无需重新定位，性能更高。 | width = element\_handle.evaluate("el => el.clientWidth") |
| ​**​screenshot()​**​ | 截取元素区域。 | ​**​相同操作​**​，但 ElementHandle ​**​直接截取已知元素​**​，无需等待。 | element\_handle.screenshot(path="element.png") |
| ​**​scroll\_into\_view()​**​ | 滚动元素到视图中。 | ​**​相同操作​**​，但 ElementHandle ​**​强制滚动​**​（即使元素已可见）。 | element\_handle.scroll\_into\_view() |
| ​**​bounding\_box()​**​ | 获取元素坐标和尺寸。 | ​**​相同操作​**​，但 ElementHandle ​**​直接返回精确值​**​，Locator 需通过 evaluate() 计算。 | box = element\_handle.bounding\_box() |
| ​**​dispose()​**​ | ​**​手动释放元素句柄​**​（减少内存占用）。 | ​**​ElementHandle 独有​**​，Locator 无此方法。 | element\_handle.dispose() |
| ​**​as\_element()​**​ | 返回自身（用于兼容性设计）。 | ​**​ElementHandle 独有​**​，Locator 无此方法。 | element = element\_handle.as\_element() |
| ​**​owner\_frame()​**​ | 获取元素所属的 Frame 对象。 | ​**​ElementHandle 独有​**​，Locator 需通过 frame\_locator() 间接操作。 | frame = element\_handle.owner\_frame() |
| ​**​content\_frame()​**​ | 若元素是 <iframe>，返回其内部 Frame。 | ​**​ElementHandle 独有​**​，Locator 需先转换为 ElementHandle 再调用此方法。 | iframe\_element.content\_frame().click("button") |
| ​**​is\_connected()​**​ | 判断元素是否仍连接在 DOM 树中（未被移除）。 | ​**​ElementHandle 独有​**​，Locator 需通过 evaluate() 判断。 | if element\_handle.is\_connected(): ... |
| ​**​query\_selector()​**​ | 在元素内部查找子元素（返回 ElementHandle）。 | ​**​ElementHandle 独有​**​，Locator 使用 locator() 返回新的 Locator 对象。 | child = element\_handle.query\_selector(".child") |

##### Page

Page对象也可以直接进行操作，通过 page 对象直接操作元素的常用方法整理（如 page.click()），适用于快速执行简单操作：

|  |  |  |  |
| --- | --- | --- | --- |
| **方法​​** | **​​作用​​** | **​​参数示例​​** | **​​代码示例​​** |
| page.click(selector) | 点击匹配选择器的第一个元素。 | selector: str, button="left", delay=100 | page.click("button#submit") |
| page.dblclick(selector) | 双击匹配选择器的第一个元素。 | 同 click() | page.dblclick(".item") |
| page.fill(selector, value) | 向输入框填充文本（清空后输入）。 | selector: str, value: str | page.fill("input#username", "user123") |
| page.type(selector, text) | 模拟用户逐个字符输入（保留原有内容）。 | selector: str, text: str, delay=50（字符间隔） | page.type("textarea", "Hello") |
| page.check(selector) | 勾选复选框或单选框。 | selector: str | page.check("#agree") |
| page.uncheck(selector) | 取消勾选复选框。 | selector: str | page.uncheck("#option") |
| page.hover(selector) | 悬停在元素上。 | selector: str, force=True（强制悬停不可见元素） | page.hover(".tooltip-trigger") |
| page.select\_option(selector, value) | 选择下拉框的选项。 | selector: str, value="cn" 或 label="China" 或 index=1 | page.select\_option("select#country", value="CN") |
| page.focus(selector) | 聚焦到元素。 | selector: str | page.focus("input#search") |
| page.press(selector, key) | 在元素上触发按键事件（如 Enter）。 | selector: str, key: str（如 "Enter", "Tab"） | page.press("input#query", "Enter") |
| page.set\_checked(selector, checked) | 勾选或取消勾选复选框。 | selector: str, checked=True/False | page.set\_checked("#toggle", True) |
| page.get\_attribute(selector, name) | 获取元素属性值。 | selector: str, name: str（如 "href"） | href = page.get\_attribute("a.link", "href") |
| page.text\_content(selector) | 获取元素文本内容（包括隐藏文本）。 | selector: str | text = page.text\_content(".title") |
| page.inner\_text(selector) | 获取元素可见文本（受 CSS 影响）。 | selector: str | text = page.inner\_text(".status") |
| page.is\_visible(selector) | 判断元素是否可见。 | selector: str | visible = page.is\_visible(".loading") |
| page.is\_enabled(selector) | 判断元素是否启用。 | selector: str | enabled = page.is\_enabled("button") |
| page.is\_checked(selector) | 判断复选框是否选中。 | selector: str | checked = page.is\_checked("#checkbox") |

##### Locator、Selector、Page页面操作对比

下面是三种定位操作方法的对比总览

|  |  |  |  |
| --- | --- | --- | --- |
| **维度​​** | **​​Locator​​** | **​​Selector（ElementHandle）​​** | **​​直接 Page 操作（如 page.click()）​​** |
| ​**​定位机制​**​ | 动态定位（延迟查找元素） | 立即查找并绑定元素 | 立即查找第一个匹配元素 |
| ​**​自动等待​**​ | ✅ 自动等待元素可交互/可见 | ❌ 需手动调用等待方法（如 wait\_for\_element\_state()） | ✅ 自动等待元素可交互（但仅一次） |
| ​**​生命周期​**​ | 每次操作重新查找元素 | 元素句柄可能失效（需重新定位） | 每次调用重新查找元素 |
| ​**​返回值​**​ | Locator 对象（描述如何查找元素） | ElementHandle 对象（直接指向 DOM 元素） | 无返回值（直接操作元素） |
| ​**​链式调用​**​ | ✅ 支持（如 page.locator().locator()） | ❌ 需分步操作 | ❌ 无链式操作 |
| ​**​错误处理​**​ | 自动重试，适合动态内容 | 操作失效元素会抛出错误 | 若元素不存在直接报错 |
| ​**​适用场景​**​ | 动态页面、复杂交互、健壮性要求高 | 精准控制元素、高频属性访问、执行 JS | 快速脚本、简单操作 |

**详解**

1. **定位机制**

* **Locator：**
* 动态定位：​每次操作时重新查询 DOM，确保操作最新元素。
* **Selector（query\_selector / query\_selector\_all）：**
* 静态绑定：​立即查找元素并返回 ElementHandle，若页面更新，句柄可能失效。
* **直接 Page 操作：**
* 立即查找：​每次调用时查找第一个匹配元素，无状态保留。

1. **自动等待与重试**

* **Locator：**
* 自动等待：​自动等待元素可见/可交互，并自动重试操作（默认超时 30 秒）。
* **Selector（ElementHandle）：**
* 无自动等待：​需手动处理等待和状态检查。
* **直接 Page 操作：**
* 单次自动等待：​操作前等待元素可交互，但若元素在等待后消失仍会失败。

1. **错误处理与健壮性**

* **Locator：**
* 自动重试：​网络波动或短暂元素缺失时自动重试，适合不稳定环境。
* **Selector（ElementHandle）：**
* 严格报错：​元素失效后操作直接抛出错误，需自行捕获异常。
* **直接 Page 操作：**
* 无重试机制：​若元素在查找后消失，操作失败。

1. **性能与代码风格**

* **Locator：**
* 性能略低：​因重试和动态查找，但代码更健壮。
* 链式语法：​适合复杂定位。
* **Selector（ElementHandle）：**
* 高性能：​直接操作元素句柄，适合高频属性访问或 JS 执行。
* **直接 Page 操作：**
* 代码简洁：​适合简单脚本或快速验证。

**✅ 推荐实践**

* **优先使用 Locator**：​它提供了更强大的自动等待和重试能力，适用于大多数场景。
* **使用 Selector（query\_selector / query\_selector\_all）**：​当需要获取元素句柄进行复杂操作时使用。
* **直接 Page 操作**：​仅在页面结构简单且稳定时使用。

#### 等待机制

Playwright不像Selenium对于操作元素前需要主动等待元素加载后再操作，其基本的页面操作都封装了自动等待机制，当然，对于一些特殊的情况也可以进行主动的等待或者自定义等待。

##### 自动等待

Playwright 在执行元素操作时默认启用智能等待，确保元素 ​​可交互​​ 后才执行操作，无需手动添加等待。

* ​​**触发自动等待的操作​**

|  |  |  |
| --- | --- | --- |
| **​​方法​​** | **​​描述​​** | **​​等待条件​​** |
| click() | 点击元素 | 元素存在、可见、未被遮挡、可接收点击（如 pointer-events 非 none）。 |
| fill() | 输入文本 | 元素存在、可见、可编辑（非 disabled 或 readonly）。 |
| check() | 勾选复选框 | 元素存在、可见、可交互。 |
| hover() | 悬停元素 | 元素存在、可见。 |
| type() | 模拟键盘输入 | 同 fill()。 |

* **自动等待的默认超时**
* **全局默认超时​​：**30 秒。
* **​​自定义超时​​：**通过 timeout 参数覆盖：
* **自动等待的局限性​**
* **​​无法处理复杂条件​​：**如等待元素数量变化或特定文本内容。
* **​​不适用于非元素操作​​：**如等待网络请求完成或页面跳转。

##### 主动等待

当需要等待特定条件（如元素出现、网络请求完成）时，使用显式等待 API。

* **元素状态等待​​**

|  |  |  |
| --- | --- | --- |
| ​**方法​​** | **​​描述​​** | **​​示例​​** |
| locator.wait\_for() | 等待元素达到特定状态。 | locator.wait\_for(state="visible") |
| page.wait\_for\_selector() | 等待选择器匹配的元素出现/消失。 | page.wait\_for\_selector(".loading", state="hidden", timeout=5000) |
| page.wait\_for\_function() | 等待自定义 JavaScript 条件。 | page.wait\_for\_function("() => document.title === 'Success'") |

​​**状态参数 (state)​​**：

* **"visible"**：元素可见。
* **"hidden"**：元素不可见。
* **"attached"**：元素存在于 DOM。
* **"detached"**：元素不存在于 DOM。
* ​**导航与页面加载​​**

|  |  |  |
| --- | --- | --- |
| ​​**方法​​** | **​​描述​​** | **​​示例​​** |
| page.wait\_for\_navigation() | 等待页面导航完成。 | with page.expect\_navigation(): await page.click("a#next") |
| page.wait\_for\_url() | 等待 URL 匹配指定模式。 | page.wait\_for\_url("\*\*/dashboard", timeout=10000) |
| page.wait\_for\_load\_state() | 等待页面加载到指定状态。 | page.wait\_for\_load\_state("networkidle") |

​​**加载状态 (state)​​：**

* **"load"：**等待 load 事件触发。
* **"domcontentloaded"**：等待 DOM 解析完成。
* **"networkidle"**：等待网络空闲（500ms 内无请求）。
* ​​**网络请求与响应​​**

|  |  |  |
| --- | --- | --- |
| ​​**方法​​** | **​​描述​​** | **​​示例​​** |
| page.wait\_for\_request() | 等待特定请求触发。 | with page.expect\_request("\*\*/api/data") as req: ... |
| page.wait\_for\_response() | 等待特定响应完成。 | with page.expect\_response("\*\*/api/data") as resp: ... |

对于复杂场景，可结合循环和条件判断实现自定义等待：例如等待所有符合元素加载、等待某个元素消失等。

#### 文件操作

Playwright为默认的静默文件下载，最常使用的文件下载方式为page.on()监听下载事件和用expected\_download()来预计下载事件。文件上传可见[selenium文件上传](#_文件上传)，用法基本一致

##### playwright下载原理

playwright会拦截下载流，当监听或预计到下载事件后，会把文件下载成一个临时文件，同时可以得到一个download对象用于得到临时路径、推荐文件名等信息，然后通过save\_as把临时文件保存到一个指定的目录中。

##### on(‘download‘)监听下载事件

page.on('download'，回调函数) 是 Playwright 中用于监听文件下载事件的一个非常有用的功能。它捕获所有由页面上触发的下载事件，通过事件监听器，你可以捕获到下载的文件，并进行处理。

示例：

import os

from playwright.sync\_api import sync\_playwright

# 用于处理下载事件的函数

def handle\_download(download):

    # 获取临时文件路径

    temp\_file\_path = download.path()

    print(f"临时文件路径：{temp\_file\_path}")

    # 获取浏览器建议的文件名

    suggested\_filename = download.suggested\_filename()

    print(f"推荐的文件名：{suggested\_filename}")

    # 保存文件到指定目录

    save\_path = os.path.join("/path/to/save", suggested\_filename)

    download.save\_as(save\_path)

    print(f"文件已保存到：{save\_path}")

# 启动 Playwright 并执行脚本

with sync\_playwright() as p:

    browser = p.chromium.launch()

    page = browser.new\_page()

    # 设置页面监听下载事件

    page.on('download', handle\_download)  # 注册事件监听

    # 访问目标页面

    page.goto('https://example.com/download-page')

    # 假设页面上有一个下载按钮，点击下载

    page.locator('button#download').click()

    # 等待下载完成

    page.wait\_for\_timeout(5000)  # 等待5秒确保下载完成

    browser.close()

此处，在创建page后我们就注册了一个下载监听器，当触发事件‘download‘时，监听器会自动捕获到download对象注入后面的回调函数，download对象一般包含的方法属性有：

* **path()：**临时文件路径
* **suggested\_filename()：**览器建议的文件名
* **size()：**返回下载文件的大小，以字节（bytes）为单位。
* **save\_as()：**保存文件的函数，接收文件路径

在回调函数（也可以用lamda匿名函数简单写）中，通过调用download.save\_as()来把文件保存到指定的路径、指定的文件名。然后可以在需要的时候主动通过off ("download", callback)来取消监听。

##### expected\_download()预期下载事件

with self.page.expect\_download() as download\_info：用于包裹触发下载信息的语句，然后会获取到一个download\_info对象，需要注意的是，download\_info对象是一个上下文管理对象，需要通过download\_info.value才能得到与前面on监听得到的download对象。获取到download对象后操作与前面基本一致。

with self.page.expect\_download() as download\_info:

     self.page.locator(self.download\_btn).click()   #下载文件

download = download\_info.value

tmp\_path = download.path()

download.save\_as(os.path.join(download\_dir, download.suggested\_filename))

##### on()与expected\_download()的对比

其实更推荐使用expected\_download来监听下载，可以按需下载，同时避免未知的下载事件导致风险。

|  |  |  |
| --- | --- | --- |
| **对比维度​​** | **​​page.on('download')​​** | **​​expect\_download()​​** |
| ​**​触发方式​**​ | 全局监听所有下载事件（包括未知或多次触发） | 精准捕获特定操作触发的单个下载事件（如点击按钮） |
| ​**​代码结构​**​ | 需注册回调函数，可能涉及异步逻辑 | 通过 async with 上下文管理器限定作用域，结构简洁 |
| ​**​适用场景​**​ | 监听未知或多次下载（如页面自动下载、批量导出） | 明确的用户操作触发的下载（如点击“导出 CSV”按钮） |
| ​**​事件隔离性​**​ | 可能捕获非目标下载（需手动过滤） | 仅捕获绑定操作后的下载，避免干扰 |
| ​**​超时与错误处理​**​ | 需手动实现超时和错误检测 | 内置超时机制，自动抛出 TimeoutError |
| ​**​资源管理​**​ | 需手动移除监听器（否则可能内存泄漏） | 自动释放资源（上下文结束后） |
| ​**​并发下载处理​**​ | 需自行管理下载队列（如通过列表存储并逐个处理） | 每次仅处理一个下载，需多次调用以处理多个下载 |
| ​**​代码复杂度​**​ | 较高（需处理事件监听、状态跟踪、资源清理） | 较低（开箱即用，无需额外逻辑） |
| ​**​性能影响​**​ | 全局监听可能占用额外资源 | 按需监听，资源占用更高效 |

##### 下载检查

Playwright天生就考虑下载场景，不需要主动检查文件下载。其page.expect\_download() 和 download.save\_as() 方法：

* 自动等待下载完成
* 下载成功后才继续执行后续代码

Playwright内部会做这些事情：

* 下载请求一发起，它就监听
* 文件数据流一旦开始，它捕获
* 文件数据流保存完毕，才继续
* 出现异常（比如下载失败），它直接抛错

所以你不用自己检测文件目录、也不用担心.crdownload这类临时文件。

#### 异步操作

Playwright 提供了 **同步** 和 **异步** 两种模式来进行浏览器自动化测试。**同步操作**和**异步操作**的主要区别在于是否会阻塞代码的执行。同步操作按顺序执行，每个操作完成后才能进行下一个操作；而异步操作则允许其他任务在等待过程中继续执行，从而提高了效率，尤其适用于需要等待外部资源（如页面加载、网络请求等）的场景。

##### 同步与异步

之前介绍的内容大部分是通过同步playwright介绍，其实playwright的同步与异步的基本使用是基本一致，就是异步需要用asyn、await来管理函数或者资源操作。

* **同步操作**
* 实现：

with sync\_playwright() as p:

* 特点：
* **阻塞：**同步操作会阻塞程序的执行，直到当前任务完成后才会继续。
* **顺序执行：**操作会按照书写顺序逐个执行。
* **没有 async / await**：同步函数通常不需要使用 async 和 await，代码的执行顺序会严格按照定义的顺序来执行。
* **异步操作**
* 实现：

async with async\_playwright() as p:

* 特点：
* **非阻塞：**异步操作不会阻塞主线程，允许在等待某个任务完成时继续执行其他任务。
* **并发执行：**多个异步任务可以同时进行（并发），但每个异步任务依然会在其完成后继续执行。
* **需要 async / await**：异步函数必须使用 async 关键字进行定义，并且需要使用 await 来等待异步操作完成。
* **整体区别**
* ​**​等待机制​**​
  + ​**​同步​**​：操作顺序执行，每一步必须等待完成才能继续。
  + ​**​异步​**​：用 await 明确等待操作完成，期间可执行其他任务。
* ​**​性能差异​**​
  + ​**​同步​**​：I/O 操作（如网络请求）串行执行，多任务时效率低。
  + ​**​异步​**​：I/O 操作可并发执行，适合高吞吐量场景。
* ​**​代码结构​**​
  + ​**​同步​**​：直接编写，无需 async/await。
  + ​**​异步​**​：需 async/await 和事件循环管理。

##### 异步原理

Playwright 的异步操作是其最强大的特性之一，特别适合于提高自动化测试的效率。通过 async 和 await 语法，Playwright 可以在等待浏览器操作完成时执行其他任务，从而有效地提升执行速度，尤其是在涉及大量网络请求或页面加载的场景中。

**异步操作的原理：**

Playwright 的异步操作本质是 **​​Promise 状态机驱动的事件循环调度​​：**

1. **​​每个异步操作（如 page.click()）返回一个 Promise​​：**

* 代表操作的「未来完成状态」（pending/fulfilled/rejected）。

1. **​​await 的作用​​：**

* 将 Promise 提交给事件循环，​​挂起当前协程​​，线程转去执行其他就绪的 Promise。

1. **​​浏览器通信层​​：**

* Playwright 向浏览器发送指令后，​​立即返回 Promise​​，不等待响应。
* 浏览器完成操作后，通过 WebSocket 回传结果，​​标记 Promise 为 fulfilled​​。

1. **​​事件循环检测到 Promise 完成​​：**

* 将对应协程重新加入队列，​​恢复执行后续代码​​

总地说就是：执行异步操作时，操作完成会返回一个promise，当使用await关键字时，会阻塞，等待操作promise才会进行下一步。而用asyn def创建异步函数时，会自动的让其完成操作后饭后一个promise。

##### 异步操作的使用

###### 异步操作的创建

不像同步操作即可以用with也可以用start()/close()主动管理，异步操作异步只用with管理：

**async with async\_playwright() as p:**

from playwright.async\_api import async\_playwright

import asyncio

async def test\_playwright():

    async with async\_playwright() as p:

        browser = await p.chromium.launch(headless=False)  # 启动浏览器

        page = await browser.new\_page()  # 创建新页面

        await page.goto("https://example.com")  # 异步操作

        # 无需显式调用 browser.close()，因为 async with 会自动处理

        # 浏览器实例会在 async with 结束时自动关闭

# 运行异步函数

asyncio.run(test\_playwright())

###### await与asyn

async 和 await 关键字是用于处理 **异步编程** 的，它们是为了使代码能够执行异步操作，避免阻塞主线程，提升性能和响应能力。Python 对 async 和 await 的最主要的限制之一就是它们只能在 **异步函数** 中使用。

* **asyn：**

async 关键字用于定义 异步函数。异步函数在被调用时不会立即执行，而是返回一个 协程对象（coroutine object），该协程对象可以被调度执行。

**注意事项：**

* 异步函数的执行是非阻塞的，可以在执行过程中挂起，等待异步操作完成。
* 异步函数返回的是一个 协程对象，你必须通过 await 或者 事件循环 来执行这个协程。
* **await：**

await 用来等待一个异步操作完成，它会挂起当前协程并让出控制权给事件循环，直到所等待的操作完成后，才会继续执行后续代码

* **async 和 await 的执行流程**
* **async：**当定义一个异步函数时，函数体内的代码并不会立即执行，而是返回一个协程对象。
* **await：**当我们在协程中使用 await 时，程序会挂起当前协程，去执行 await 后面的异步操作，等待其完成后，再继续执行协程。

总的来说，与异步playwright相关的库操作一般都是用asyn创建的，自定义异步操作也需要使用asyn进行创建，在实现代码时，如果需要主动等待操作而不是让其异步操作时，使用await来主动等待，以防达不到后面操作的前置条件。

#### Playwright的API操作

#### Playwright Test框架

Playwright Test 是 Playwright 官方自带的测试运行器，专门用来做：

* Web UI自动化测试
* 端到端（E2E）流程测试

它把传统需要分开配环境（比如 Mocha + Selenium + Chai + WebDriver）的流程，**一体化集成**，直接开箱即用，速度极快，稳定性高。

##### 框架功能

|  |  |  |
| --- | --- | --- |
| **分类** | **功能** | **说明** |
| 测试管理 | test()、describe()、beforeEach()、afterEach() | 组织用例、编排前后处理逻辑 |
| 自动等待 | 全自动处理元素可见、可点击、网络稳定 | 解决了 Selenium 常见的同步问题 |
| 断言库 | expect(locator)、expect(page) | 为 UI 元素设计的高阶断言，专门适配异步状态 |
| 浏览器控制 | 同时支持 Chromium、Firefox、WebKit | 统一API，轻松多浏览器测试 |
| 设备模拟 | 手机、平板、不同网络环境模拟 | 直接模拟 iPhone、Android等 |
| 多线程并发 | 测试用例并行执行，加速测试 | 默认就支持，无需额外配置 |
| Tracing机制 | 测试失败时生成操作轨迹文件 | 可以回放看到每一步操作 |
| 截图/录像 | 自动截图，录制视频，辅助调试 | 用于生成详细测试报告 |
| 请求拦截 | 网络请求 Mock、修改、监听 | 可控制后端数据，提升测试稳定性 |
| Session复用 | 登录态管理（storageState） | 避免每次登录，提升效率 |
| CLI工具 | 灵活运行、调试、生成报告 | npx playwright test、npx playwright codegen 等 |

##### 核心概念

|  |  |
| --- | --- |
| **概念** | **描述** |
| test() | 定义一个测试用例 |
| expect() | 断言页面状态或元素属性 |
| fixture | 抽象公共资源（如page、browserContext） |
| project | 配置不同测试场景（如浏览器类型、设备） |
| reporter | 测试报告输出（list、HTML、json） |
| storageState | 保存登录态/session，复用登录 |

##### 基本使用

import { test, expect } from '@playwright/test';

test('basic test', async ({ page }) => {

  await page.goto('https://example.com');

  await expect(page).toHaveTitle(/Example Domain/);

  const heading = page.locator('h1');

  await expect(heading).toBeVisible();

});

* test()：定义测试
* page.goto()：跳转网页
* expect()：断言页面状态

##### 特色能力

* **自动同步等待**
* Playwright 自动等待元素状态，比如：
* 等元素出现
* 等元素可交互
* 等网络请求完成
* 不需要自己写 sleep()、time.sleep() 之类。
* **多浏览器和多设备**
* 一个项目可以同时跑：
* Chromium（Chrome/Edge内核）
* Firefox
* WebKit（Safari内核）
* 可以模拟不同手机型号、网络环境。
* **多线程并行**
* Playwright 默认可以同时跑很多个用例。
* 比如 10个用例，能开10个线程，同时跑，大大加快执行速度。
* **失败自动收集Trace**
* 失败自动录制操作过程。
* 生成 .zip 文件，里面有点击、跳转、请求响应的全过程。
* 可以在浏览器打开回放，非常清晰定位问题。

##### 框架比对

|  |  |  |
| --- | --- | --- |
| **项目** | **Playwright Test** | **Selenium + 测试框架** |
| 配置难度 | 极低，开箱即用 | 高，需要搭环境 |
| 并发执行 | 原生支持 | 需要自行加线程池或并发库 |
| 等待机制 | 自动等待元素稳定 | 需自己加等待处理 |
| 失败调试 | Trace回放、录像截图丰富 | 主要靠日志 |
| 执行速度 | 快（多线程） | 慢（单线程居多） |
| 浏览器支持 | Chromium、Firefox、WebKit | 依赖 WebDriver 版本 |
| 适合测试类型 | UI + 流程测试 | UI为主 |

## APP自动化

### Appium环境搭建与ADB命令

### Appium启动参数详解及元素定位

### Appium常用方法以及键盘鼠标手势事件

### Appium等待方式与二次封装

### Appium之H5及Toast处理

## 自动化框架

可详见项目实战