每次不重样，带你收获最新测试技术！

微信扫一扫关注我们

投稿邮箱：editor@51testing.com

有限状态自动机（FSM）在自动化测试框架中的实战

◆ 作者：blues_C

一、前言

随着快速迭代和业务复杂度的提升，传统的自动化测试脚本逐渐暴露出维护难、扩展性差、覆盖不全等问题。如何用更科学、更系统的方法来建模和实现复杂业务流程的自动化测试，是每一位测试工程师必须思考的问题。

有限状态自动机（Finite State Machine, FSM）作为一种经典的建模工具，广泛应用于编译原理、协议设计、流程控制等领域。近年来，FSM 思想也逐渐被引入到自动化测试领域，成为提升测试系统性、可维护性和自动化程度的重要利器。

本文将以 Python+Playwright+Pytest+BDD 为技术栈，系统地讲解如何在自动化测试框架中引入 FSM 思想，希望你能从本文中获得有价值的参考和启发。

二、有限状态自动机（FSM）理论基础

2.1 FSM 的基本概念

有限状态自动机（FSM）是一种用于描述系统在有限个状态之间转移的数学模型。它由以下几个核心要素组成：

状态：系统在某一时刻所处的情形。例如，用户未登录、已登录、已注册未验证等。

事件/输入：触发状态转移的外部动作或条件。例如，点击登录按钮、输入验证码、点击退出等。

转移：由事件触发的状态变化过程。例如，用户输入正确账号密码后，从“未登录”

转移到“已登录”。

初始状态：系统开始时的状态。

终止状态：系统结束时的状态（可选）。

FSM 通常用状态转移图或状态转移表来表示。状态转移图用节点表示状态，用有向边表示转移；状态转移表则用二维表格列出所有状态和事件的对应关系。

2.2 FSM 的分类

FSM 主要分为两类：

确定性有限状态自动机（DFA）：在任何时刻，系统在某一状态下对某一输入只有唯一的转移。

非确定性有限状态自动机（NFA）：在某一状态下对某一输入可能有多个转移。

在自动化测试领域，通常采用 DFA 进行流程建模，因为业务流程大多是确定性的。

2.3 FSM 在自动化测试中的优势

FSM 在自动化测试中的应用，带来了如下显著优势：

1. 流程建模清晰：将复杂业务流程拆解为状态和转移，结构化表达系统行为，便于理解和沟通。

2. 测试用例自动生成：可根据 FSM 自动生成覆盖各种状态和转移的测试用例，提升覆盖率。

3. 异常路径检测：便于发现系统在异常输入下的行为，提升健壮性。

4. 可维护性和可扩展性：流程变更时只需调整 FSM 模型，测试代码易于维护和扩展。

5. 与自动化框架无缝集成：FSM 思想可与主流自动化测试框架（如 Selenium、Playwright、Appium 等）无缝结合，提升整体测试能力。

三、FSM 在自动化测试中的应用场景

3.1 典型应用场景

1. 用户注册与登录流程：如注册、邮箱验证、登录、退出等多状态流程。

2. 订单处理流程：如下单、支付、发货、收货、退货等。

3. 权限与审批流程：如申请、审批、驳回、通过等。

4. 多页面、多分支业务流程：如电商购物、内容发布、数据分析等。

3.2 适用性分析

FSM 适用于以下场景：

- 业务流程复杂、状态众多、分支多。
- 流程经常变更、需频繁扩展。
- 需自动生成测试用例、提升覆盖率。
- 需对异常路径、边界条件进行系统性测试。

对于简单的线性流程，FSM 虽可用，但未必必要。对于复杂流程，FSM 则能极大提升测试的系统性和可维护性。

四、项目结构设计

4.1 推荐项目结构

良好的目录结构是高效开发和维护的基础。结合 FSM 思想，pytest-playwright 框架项目结构设计如下：

```
pytest-playwright-demo/
├── fsm/
│   └── user_fsm.py
├── pages/
│   ├── base_page.py
│   ├── login_page.py
```

```
|   ├── register_page.py
|   └── dashboard_page.py
├── tests/
|   └── test_user_flow_fsm.py
├── conftest.py
├── requirements.txt
└── README.md
```

fsm/：存放 FSM 模型相关代码。

pages/：页面对象模型（POM），每个页面一个类，便于维护。

tests/：测试用例目录。

conftest.py：Pytest 的全局 fixture 配置。

requirements.txt：依赖库清单。

README.md：项目说明文档。

4.2 页面对象模型（POM）设计

页面对象模型（POM）是一种将页面操作与测试逻辑分离的设计模式。每个页面封装为一个类，包含页面元素和操作方法，便于维护和复用。

4.3 FSM 与 POM 的结合

FSM 负责流程建模，POM 负责页面操作。

测试用例只需关注流程和断言，页面细节交由 POM 处理。

FSM 与 POM 解耦，提升代码复用性和可维护性。

五、复杂业务流程 FSM 建模与实现

5.1 业务流程分析

以“用户注册→邮箱验证→登录→退出”为例，设计如下 FSM：

状态

- 未注册
- 已注册未验证
- 已验证未登录
- 已登录

事件

- 注册
- 邮箱验证
- 登录
- 退出

状态转移图

5.2 FSM 代码实现

```python
# fsm/user_fsm.py
class UserFSM:
    def __init__(self):
        self.state = "未注册"
    def register(self):
        if self.state == "未注册":
            self.state = "已注册未验证"
```

```
    def verify_email(self):
        if self.state == "已注册未验证":
            self.state = "已验证未登录"
    def login(self):
        if self.state == "已验证未登录":
            self.state = "已登录"
    def logout(self):
        if self.state == "已登录":
            self.state = "已验证未登录"
```

5.3 页面对象实现

##基础页面类

python

```
# pages/base_page.py
from playwright.sync_api import Page
class BasePage:
    def __init__(self, page: Page):
        self.page = page
```

##注册页面

python

```
# pages/register_page.py
from .base_page import BasePage
class RegisterPage(BasePage):
    def goto(self):
        self.page.goto("https://example.com/register")
    def register(self, username, email, password):
        self.page.fill("#username", username)
        self.page.fill("#email", email)
        self.page.fill("#password", password)
        self.page.click("#register-btn")
```

##登录页面

python

```
# pages/login_page.py
from .base_page import BasePage
class LoginPage(BasePage):
    def goto(self):
        self.page.goto("https://example.com/login")
    def login(self, username, password):
        self.page.fill("#username", username)
        self.page.fill("#password", password)
        self.page.click("#login-btn")
##用户中心页面
python
# pages/dashboard_page.py
from .base_page import BasePage
class DashboardPage(BasePage):
    def logout(self):
        self.page.click("#logout-btn")
```

5.4 Pytest Fixture 配置

```
python
# conftest.py
import pytest
from playwright.sync_api import sync_playwright
@pytest.fixture(scope="function")
def browser_page():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=True)
        page = browser.new_page()
        yield page
        browser.close()
```

5.5 测试用例实现

```
python
# tests/test_user_flow_fsm.py
import pytest
from fsm.user_fsm import UserFSM
from pages.register_page import RegisterPage
from pages.login_page import LoginPage
from pages.dashboard_page import DashboardPage
def test_user_registration_flow(browser_page):
    fsm = UserFSM()
    register_page = RegisterPage(browser_page)
    login_page = LoginPage(browser_page)
    dashboard_page = DashboardPage(browser_page)
    # 1. 未注册 -> 注册
    assert fsm.state == "未注册"
    register_page.goto()
    register_page.register("testuser", "test@example.com", "password123")
    fsm.register()
    assert fsm.state == "已注册未验证"
    # 2. 邮箱验证（假设自动完成）
    # 这里可以模拟邮箱验证的操作
    fsm.verify_email()
    assert fsm.state == "已验证未登录"
    # 3. 登录
    login_page.goto()
    login_page.login("testuser", "password123")
    fsm.login()
    assert fsm.state == "已登录"
    # 4. 退出
    dashboard_page.logout()
    fsm.logout()
    assert fsm.state == "已验证未登录"
```

5.6 断言与流程控制

在每一步操作后，均通过 FSM 的状态断言，确保流程的正确性。这种方式不仅验证了页面操作的有效性，也验证了业务流程的完整性。

六、FSM 在自动化测试中的进阶应用

6.1 多分支流程建模

实际业务往往存在多分支、多异常路径。例如，注册时可能遇到用户名已存在、邮箱格式错误等情况。此时，只需在 FSM 中增加相应状态和转移即可：

```
python
class UserFSM:
    def __init__(self):
        self.state = "未注册"
    def register(self, username_exists=False, email_invalid=False):
        if self.state == "未注册":
            if username_exists:
                self.state = "注册失败_用户名已存在"
            elif email_invalid:
                self.state = "注册失败_邮箱格式错误"
            else:
                self.state = "已注册未验证"
```

6.2 测试用例自动生成

结合 FSM 模型，可以自动生成覆盖所有状态转移的测试用例，提升测试覆盖率。例如，利用图遍历算法（如 DFS、BFS）遍历所有可能路径，自动生成测试脚本。

示例：自动生成测试路径

```
python
def dfs(fsm, path, all_paths):
    if fsm.state == "终止状态":
        all_paths.append(list(path))
```

```
            return
        for event in fsm.get_possible_events():
            fsm_copy = copy.deepcopy(fsm)
            fsm_copy.trigger(event)
            path.append(event)
            dfs(fsm_copy, path, all_paths)
            path.pop()
```

6.3 与 BDD 结合

FSM 与行为驱动开发（BDD）天然契合。可将每个状态转移映射为 Gherkin 语法的 Step，实现业务流程与测试代码的无缝对接。

示例：Gherkin 语法

gherkin

```
Feature: 用户注册与登录流程
  Scenario: 正常注册、验证、登录、退出
    Given 用户未注册
    When 用户注册
    And 用户邮箱验证
    And 用户登录
    Then 用户已登录
    When 用户退出
    Then 用户已验证未登录
```

6.4 状态覆盖与路径覆盖

状态覆盖：确保每个状态都被测试用例覆盖到。

转移覆盖：确保每个状态转移都被测试用例覆盖到。

路径覆盖：确保所有可能的状态路径都被测试用例覆盖到（通常只对关键路径做覆盖，避免组合爆炸）。

6.5 FSM 与数据驱动测试结合

FSM 可与数据驱动测试（DDT）结合，实现同一流程下多组数据的自动化测试。例如，注册流程下测试不同的用户名、邮箱、密码组合。

七、经验分享

7.1 何时引入 FSM

业务流程复杂、状态众多时，建议引入 FSM。

流程经常变更、需频繁扩展时，FSM 能极大提升可维护性。

需自动生成测试用例、提升覆盖率时，FSM 是理想选择。

7.2 FSM 与 POM 结合

FSM 负责流程建模，POM 负责页面操作。

测试用例只需关注流程和断言，页面细节交由 POM 处理。

7.3 状态与断言分离

FSM 只负责状态流转，不直接操作页面。

页面操作与断言通过 POM 实现，保持职责单一。

7.4 代码复用与扩展

页面对象可复用于不同测试用例。

FSM 模型可扩展至更多业务场景，如支付、订单、权限等。

7.5 团队协作与文档化

FSM 模型可作为业务流程文档，便于团队沟通和协作。

建议用状态转移图、表格等方式可视化 FSM，提升可读性。

八、常见问题与解答

8.1 FSM 会不会让测试代码变复杂？

FSM 本身是为了解决复杂流程而设计的。对于简单流程，FSM 可能显得“重”，但对于多状态、多分支的复杂流程，FSM 能极大提升代码的结构性和可维护性。建议根据实际业务复杂度灵活引入 FSM。

8.2 FSM 如何与现有测试框架集成？

FSM 是一种思想和建模方式，可与任何自动化测试框架集成。只需将 FSM 模型作为流程控制器，结合页面对象和测试用例即可。

8.3 FSM 如何应对流程变更？

FSM 模型结构清晰，流程变更时只需调整状态和转移，测试用例可自动适应新流程，极大降低维护成本。

8.4 FSM 如何与 CI/CD 集成？

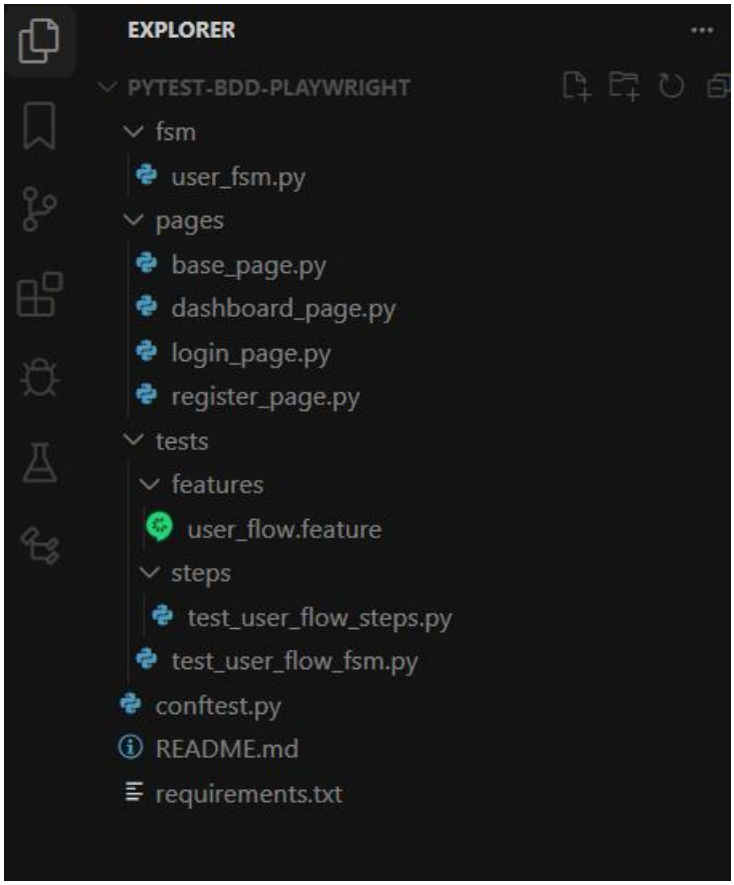
FSM 与自动化测试框架无缝集成，可直接在 CI/CD 流水线中运行。建议结合 pytest-html、Allure 等报告工具，生成可视化测试报告。

8.5 FSM 如何支持多用户、多角色测试？

可为不同用户、角色设计独立的 FSM 模型，或在同一 FSM 中引入角色状态，支持多用户、多角色的复杂流程测试。

九、完整项目实战案例

9.1 项目结构



9.2 关键代码内容

##FSM 实现

python

```
# fsm/user_fsm.py
class UserFSM:
    def __init__(self):
        self.state = "未注册"
    def register(self):
        if self.state == "未注册":
            self.state = "已注册未验证"
    def verify_email(self):
        if self.state == "已注册未验证":
```

```
                self.state = "已验证未登录"
        def login(self):
            if self.state == "已验证未登录":
                self.state = "已登录"
        def logout(self):
            if self.state == "已登录":
                self.state = "已验证未登录"
```

##页面对象示例

python

```
# pages/register_page.py
from .base_page import BasePage
class RegisterPage(BasePage):
    def goto(self):
        self.page.goto("https://example.com/register")
    def register(self, username, email, password):
        self.page.fill("#username", username)
        self.page.fill("#email", email)
        self.page.fill("#password", password)
        self.page.click("#register-btn")
```

##pytest-bdd feature 文件

gherkin

```
# tests/features/user_flow.feature
Feature: 用户注册与登录流程
  Scenario: 用户完整注册、验证、登录、退出流程
    Given 用户处于未注册状态
    When 用户在注册页面注册新账号
    Then 用户状态应为 "已注册未验证"
    When 用户完成邮箱验证
    Then 用户状态应为 "已验证未登录"
    When 用户在登录页面登录
    Then 用户状态应为 "已登录"
    When 用户在仪表盘页面退出登录
    Then 用户状态应为 "已验证未登录"
```

```
##pytest-bdd 步骤实现
python
# tests/steps/test_user_flow_steps.py
import pytest
from pytest_bdd import scenarios, given, when, then, parsers
from fsm.user_fsm import UserFSM
from pages.register_page import RegisterPage
from pages.login_page import LoginPage
from pages.dashboard_page import DashboardPage
scenarios('../features/user_flow.feature')
@pytest.fixture
def fsm():
    return UserFSM()
@pytest.fixture
def register_page(browser_page):
    return RegisterPage(browser_page)
@pytest.fixture
def login_page(browser_page):
    return LoginPage(browser_page)
@pytest.fixture
def dashboard_page(browser_page):
    return DashboardPage(browser_page)
@given("用户处于未注册状态")
def user_unregistered(fsm):
    assert fsm.state == "未注册"
@when("用户在注册页面注册新账号")
def user_registers(fsm, register_page):
    register_page.goto()
    register_page.register("testuser", "test@example.com", "password123")
    fsm.register()
@then(parsers.parse('用户状态应为 "{expected_state}"'))
def check_state(fsm, expected_state):
    assert fsm.state == expected_state
```

```
@when("用户完成邮箱验证")
def user_verifies_email(fsm):
    fsm.verify_email()
@when("用户在登录页面登录")
def user_logs_in(fsm, login_page):
    login_page.goto()
    login_page.login("testuser", "password123")
    fsm.login()
@when("用户在仪表盘页面退出登录")
def user_logs_out(fsm, dashboard_page):
    dashboard_page.logout()
    fsm.logout()
##传统 Pytest 测试用例（可选）
python
# tests/test_user_flow_fsm.py
import pytest
from fsm.user_fsm import UserFSM
from pages.register_page import RegisterPage
from pages.login_page import LoginPage
from pages.dashboard_page import DashboardPage
def test_user_registration_flow(browser_page):
    fsm = UserFSM()
    register_page = RegisterPage(browser_page)
    login_page = LoginPage(browser_page)
    dashboard_page = DashboardPage(browser_page)
    # 1. 未注册 -> 注册
    assert fsm.state == "未注册"
    register_page.goto()
    register_page.register("testuser", "test@example.com", "password123")
    fsm.register()
    assert fsm.state == "已注册未验证"
    # 2. 邮箱验证（假设自动完成）
    fsm.verify_email()
```

```
    assert fsm.state == "已验证未登录"
    # 3. 登录
    login_page.goto()
    login_page.login("testuser", "password123")
    fsm.login()
    assert fsm.state == "已登录"
    # 4. 退出
    dashboard_page.logout()
    fsm.logout()
    assert fsm.state == "已验证未登录"
```

结语

有限状态自动机（FSM）为自动化测试带来了流程建模的全新视角。结合 Python、Playwright 和 Pytest，测试工程师可以高效、系统地实现复杂业务流程的自动化测试。通过 FSM 的引入，测试代码结构更加清晰，流程更加可控，维护和扩展也变得更加简单。

在日常测试项目中，我们可以根据业务复杂度灵活引入 FSM 思想，结合 POM、BDD 等最佳实践，打造高效、健壮、可持续演进的自动化测试体系。随着 AI、低代码等新技术的发展，FSM 与自动化测试的结合将更加紧密，助力企业实现更高质量、更高效率的软件交付。

拓展学习

[1] 领取【企业级全链路测试实战】项目包

咨询：微信 atstudy-js 备注：11

打造基于 WebSocket + CDP 的 Selenium 替代方案

◆作者：Tynam

题记：
在自动化测试和爬虫开发领域，Selenium 是一个广泛使用的工具，但随着技术的发展，Chrome DevTools Protocol（CDP）提供了更高效、更强大的功能。本文将介绍如何结合 WebSocket 和 CDP，打造一个类似 Selenium 的自动化工具，实现对 Chrome 浏览器的精细控制。这种方法不仅性能更高，功能更强大，而且更加轻量级。

一、概述：

在进入实操之前，先赘叙下 WebSocket、CDP 和 Selenium 与浏览器通信流程三个概念，以及为什么选择 WebSocket + CDP。

1.WebSocket

WebSocket 是一种网络通信协议，提供了在单个 TCP 连接上进行全双工通信（客户端和服务器可以同时发送和接收消息，而无需等待对方的响应。这使得实时数据交换变得更加高效和流畅。）的能力。它允许客户端和服务器之间进行实时、双向的数据交换，而无需像传统的 HTTP 请求那样频繁地建立和关闭连接。例如你去商店买东西，你问店员商品价格，店员告诉你，然后你就走了，这就是一次 HTTP 请求；而 WebSocket 可以在客户端（比如你的电脑、手机）和服务器之间建立一个持久的连接。这个连接一旦建立，双方就可以随时互相发送数据，就像打电话，电话一直通着，我们可以随时聊天，想说啥就说啥，不用像发短信那样还得等对方回复才能再发下一条。

WebSocket 支持传输文本数据和二进制数据，这使得它可以用于各种应用场景，如实时文本聊天、在线游戏、文件传输等。他也得到了大多数现代浏览器的支持，包括 Chrome、Firefox、Safari 和 Edge。此外，也有许多服务器端实现，如 Node.js 的 ws 库、Python 的 websocket-client 和 websockets 库等。

2.CDP

Chrome DevTools Protocol (CDP) 是一套开放协议，允许外部程序通过 Chrome 浏览器提供的接口与其进行交互。CDP 提供了丰富的功能，使开发者可以远程控制 Chrome 浏览器，包括操作 DOM、监控网络请求、调试代码、截取屏幕快照等。

CDP 的核心特点有：

• 基于 JSON-RPC：CDP 协议使用 JSON 格式传输数据，简单易读，易于解析和生成。

• 双向通信：不仅调试器可以发送命令，浏览器也会主动推送事件（比如断点触发、网络请求完成）。

• 模块化设计：CDP 协议分为多个模块（如 DOM、Network、Runtime 等），每个模块负责不同的功能。

CDP 的工作流程如下，这也是本文介绍的方案的工作流程：

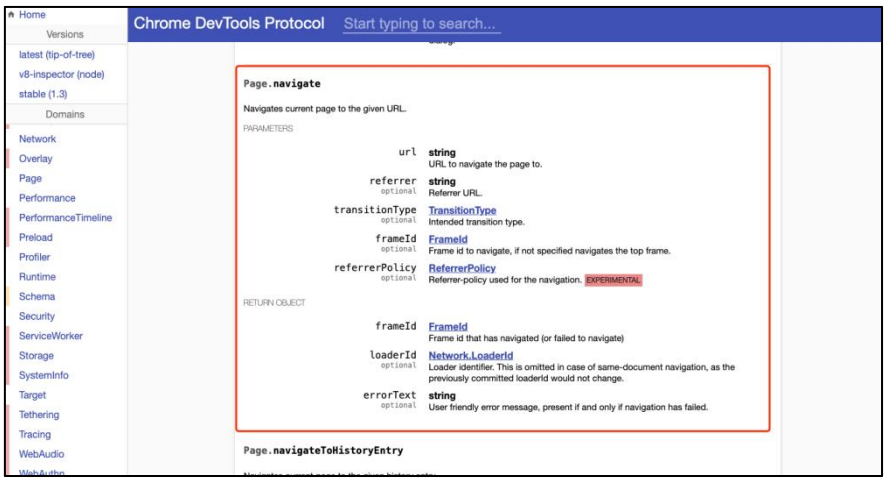
1.建立 WebSocket 连接：通过 WebSocket 与浏览器内核建立连接。

2.发送协议命令：客户端发送 JSON 格式的命令。

3.执行协议命令：浏览器内核执行命令并返回结果。

4.接收结果：客户端接收并显示结果。

在发送协议命令阶段，我们需要知道 CDP 都提供了哪些命令，这也是学习本文时需要重点关注的一点。访问 CDP 协议网站 https://chromedevtools.github.io/devtools-protocol/ 可获取 CPD 协议支持的命令或提供的方法。例如 Page.navigate 方法，如下图：



Page.navigate 方法用于将当前页面导航到指定的 URL。它支持多种参数，允许开发者指定导航的详细行为，如引用页、过渡类型等。该方法返回一个对象，包含导航结果的相关信息。

提供的参数有：

• url (string)：指定浏览器将要加载的页面地址。

• referrer (string)：指定导航请求的来源页面。这在某些情况下会影响目标页面的行为，例如，某些网站会根据引用页来设置不同的内容或行为。

• transitionType (TransitionType)：指定导航的类型，这在某些情况下会影响浏览器的行为，例如，某些过渡类型可能会触发不同的缓存策略。

• frameId (FrameId)：指定要导航的具体框架。在多框架页面中，这允许开发者控制特定框架的导航行为。

• referrerPolicy (ReferrerPolicy)：指定引用页信息的发送策略，这在处理安全性和隐私问题时非常重要。

可设置的返回值有：

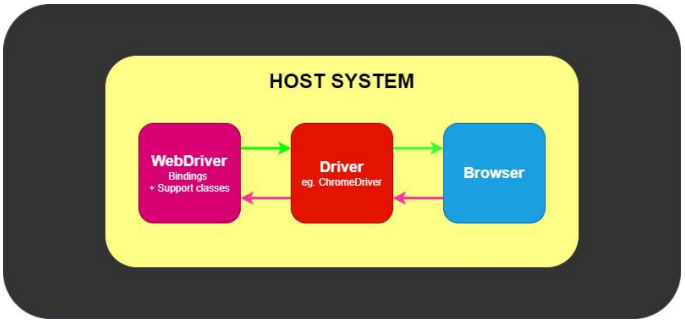
• frameId (FrameId)：返回导航操作所涉及的框架 ID，便于后续操作和调试。

• loaderId (Network.LoaderId)：加载器标识符。在同文档导航的情况下，此字段会被省略，因为之前提交的加载器 ID 不会改变。

• errorText (string)：在导航失败时，提供详细的错误信息，便于调试和问题解决。

5.Selenium 与浏览器通信流程

Selenium 与浏览器通信的基本流程图如下：



图片来源：https://www.selenium.dev/zh-cn/documentation/overview/components/

图中由三部分内容组成，WebDriver、Driver 和 Browser。其中 WebDriver 是一个抽象接口，定义了通用的方法和属性，适用于所有支持 Selenium 的浏览器；driver 是 WebDriver 接口的具体实现，负责与特定的浏览器进行通信。每个浏览器都有自己的驱动程序，如 ChromeDriver、GeckoDriver 等；Browser 就是支持的浏览器，例如 Chrome、Firefox 等。

基本通信流程为：

1.客户端代码通过 Selenium WebDriver API 发送命令。

2.WebDriver 将命令序列化为 JSON 格式，并通过 HTTP 请求发送到浏览器驱动。

3.浏览器驱动将命令转发给浏览器。

4.浏览器执行命令，并将结果返回给浏览器驱动。

5.浏览器驱动将结果序列化为 JSON 格式，并通过 HTTP 响应发送回客户端代码。

6.客户端代码接收到响应，并根据结果进行后续操作。

而本文操作流程与之类似，如下：

1.启动 Chrome 并开启远程调试功能。

2.建立 WebSocket 连接：通过 WebSocket 与浏览器内核建立连接。

3.发送协议命令：客户端通过代码发送 JSON 格式的 CDP 命令。

4.执行协议命令：浏览器内核执行命令并返回结果。

5.接收结果：客户端接收并显示结果。

与 Selenium 通信相比，我们将直接通过 CDP 与 Chrome 的内部机制交互，绕过了中间的 WebDriver 层，从而减少了不必要的开销。

6.为什么选择 WebSocket + CDP

性能优势：使用 WebSocket 和 CDP 的主要优势之一是性能。与 Selenium 相比，CDP 直接与 Chrome 的内部机制交互，绕过了中间的 WebDriver 层，从而减少了不必要的开销。这使得操作更加高效，响应时间更短，特别适合需要高效率的自动化测试和爬虫任务。

功能强大：CDP 提供了对浏览器的全面控制，包括页面导航、DOM 操作、网络请求拦截、性能分析等。这些功能不仅涵盖了 Selenium 的所有功能，还提供了许多 Selenium 难以实现的高级功能。例如，通过 CDP，你可以拦截和修改网络请求，这对于测试和开发复杂的 Web 应用非常有用。

轻量级：与 Selenium 相比，基于 WebSocket 和 CDP 的解决方案更加轻量级。它不需要安装庞大的 WebDriver，减少了依赖项和资源消耗。这使得你的自动化脚本更加简洁，部署更加方便。

易于扩展：通过 WebSocket 和 CDP，你可以轻松实现各种高级功能，如监听页面加载完成事件、进行 DOM 操作等。这些功能可以根据你的需求进行灵活扩展，使得你的自动化工具更加强大和适应不同的应用场景。

二、启动 Chrome 并开启远程调试

1、准备工作

在开始之前，你需要确保已经安装了以下工具和库：

• Chrome 浏览器：确保你的 Chrome 浏览器版本支持 CDP。

• Python：用于编写自动化脚本。版本最好在 Python3.6 以上。

• websocket-client：用于与 Chrome 的 WebSocket 调试端口通信。websocket-client 是 Python 的一个第三方库，需要提前安装。安装命令：`pip install websocket-client`

• requests：一个非常流行的 Python HTTP 库，用于发送 HTTP 请求，支持多种 HTTP 方法（如 GET、POST、PUT、DELETE 等），并提供了丰富的功能等。

2、启动 Chrome 并开启远程调试

不同操作系统，启动 Chrome 并开启远程调试的命令稍微有所区别，下面对主流的 Windows、Linux 和 MacOS 系统进行介绍。

在 Windows 系统中，可以通过以下命令启动 Chrome 并开启远程调试：

```
"C:\Program Files\Google\Chrome\Application\chrome.exe" --remote-debugging-port=9222
```

在 Linux 系统中，可以通过以下命令启动 Chrome 并开启远程调试：

google-chrome --remote-debugging-port=9222

在 MacOS 系统中，可以通过以下命令启动 Chrome 并开启远程调试：

/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --remote-debugging-port=9222

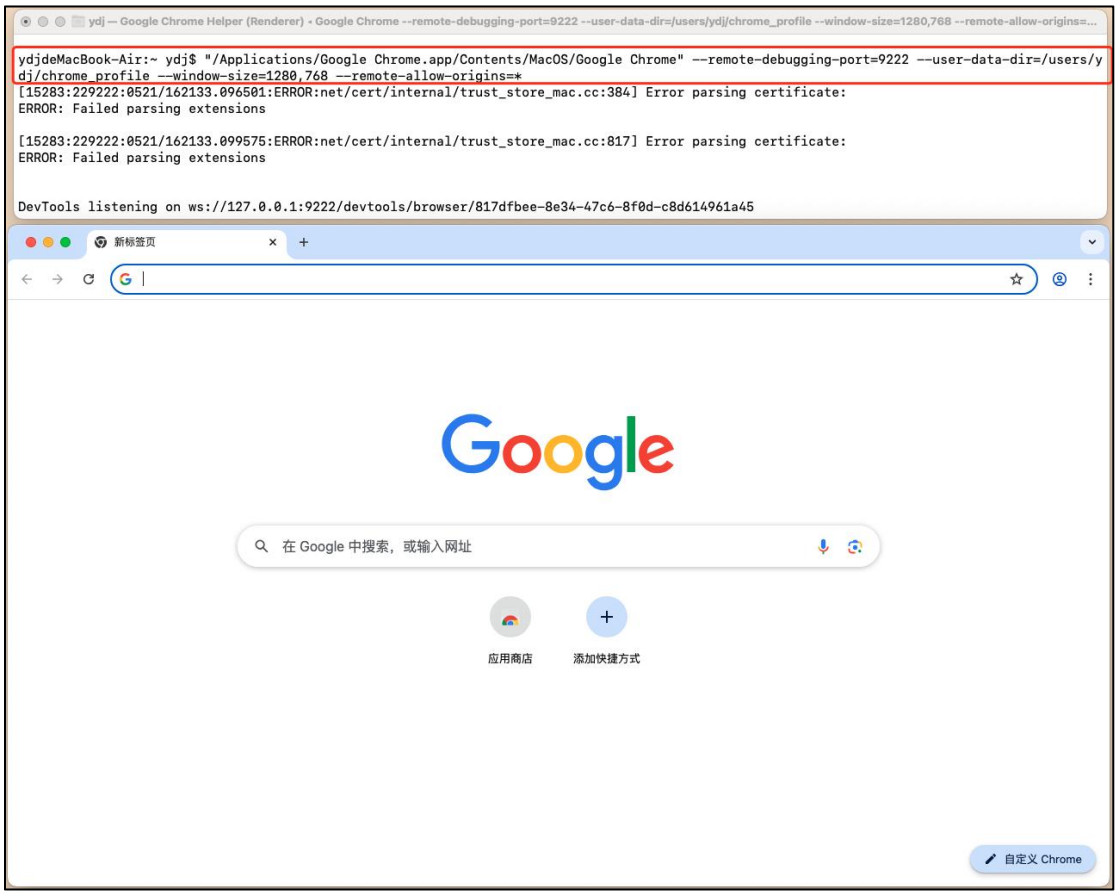
在启动时，我们还可以添加一些参数来指定调试端口、用户数据目录、代理服务器等。以下是一些常用的参数及其说明：

参数	说明
--remote-debugging-port=<port>	指定远程调试端口，默认为 9222
--user-data-dir=<path>	指定用户数据目录，避免与默认的 Chrome 配置冲突
--proxy-server=<proxy>	设置代理服务器，格式为 protocol://username:password@host:port
--headless=new	无头模式运行，适合自动化测试
--window-size=WIDTH,HEIGHT	设置初始窗口尺寸
--lang=<language>	设置浏览器语言
--disable-extensions	禁用扩展程序
--incognito	启动隐身模式
--remote-allow-origins=<origins>	限制访问范围，* 表示所有人都可以访问

例如我们启动一个调试端口为 9222、用户数据目录为 /users/ydj/chrome_profile、初始窗口大小为 1280x768 、限制访问范围为所有人都可以访问的 Chrome。命令就可以写成：

"/Applications/Google Chrome.app/Contents/MacOS/Google Chrome" --remote-debugging-port=9222 --user-data-dir=/users/ydj/chrome_profile --window-size=1280,768 --remote-allow-origins=*

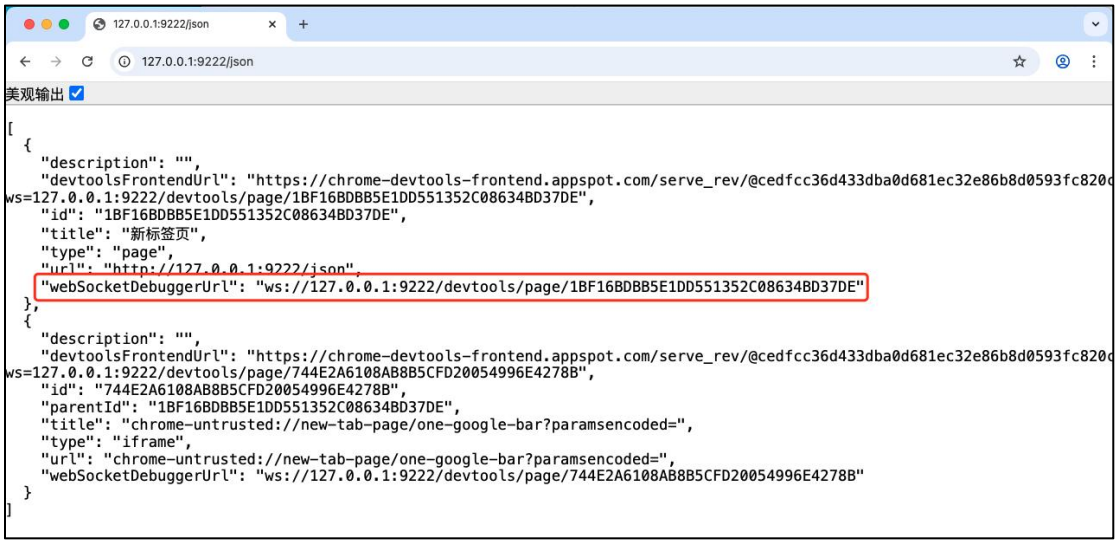
在命令行窗口执行上面命令后，会按照参数设置的启动浏览器，如下图所示：



在命令行工具中可以看到 DevTools listening on ws://127.0.0.1:9222/devtools/browser/817dfbee-8e34-47c6-8f0d-c8d614961a45 信息。表示 Chrome 已经成功启动，并且 DevTools（开发者工具）正在监听 WebSocket 地址 ws://127.0.0.1:9222/devtools/browser/817dfbee-8e34-47c6-8f0d-c8d614961a45。这意味着我们可以通过这个地址连接到 Chrome 的远程调试接口。

3、获取 WebSocket 调试 URL

启动 Chrome 并开启远程调试后，输入地址 http://127.0.0.1:9222/json 将得到一个 JSON 格式的列表，包含所有正在运行的 Chrome 实例及其调试信息。我们可以从中找到目标实例，并通过其 WebSocket URL 连接到 DevTools。如下图标记处，就是新打开 Page 页面的 WebSocket URL：



4、代码封装

前面几步介绍了手动操作，从打开启动 Chrome 并开启远程调试到获取 WebSocket 调试 URL。接下来我们需要将上面内容使用 Python 进行封装。

本次代码封装我们做到两个功能，第一需要一个命令来启动 Chrome 浏览器，并通过远程调试功能连接到特定页面。第二需要支持无头模式、自定义窗口大小、用户数据目录等参数。在类结构上，我们需要设计一个 Browser 类和 Page 类，Browser 类负责启动和管理 Chrome 浏览器；Page 类继承自 Browser 类，负责连接到特定页面并进行 WebSocket 通信。下面对这两个类中方法或变量进行说明：

1. Browser 类：

- 默认值：定义一些默认值，如 Chrome 的路径、远程调试端口等。
- 初始化方法：允许用户在实例化时覆盖默认值。
- 设置方法：提供动态设置参数的功能。
- 启动 Chrome：构造启动参数并使用 subprocess.Popen 启动 Chrome。
- 关闭 Chrome：确保在程序退出时关闭 Chrome。
- 获取页面 WebSocket URL：发送 HTTP 请求到远程调试端口，解析返回的 JSON 数据，找到对应的 WebSocket URL。

2. Page 类：

• 初始化方法：调用父类的初始化方法，设置 WebSocket URL，默认为第一个页面的 WebSocket URL。

•连接到页面：使用 websocket.create_connection 方法连接到指定的 WebSocket URL。

• 获取默认页面 WebSocket URL：获取第一个页面的 WebSocket URL。

根据上面逻辑实现的代码如下：

```
import atexit
import subprocess
import time
import requests
from websocket import create_connection


class Browser:
    # 设置默认值
    _chrome_path = "/Applications/Google Chrome.app/Contents/MacOS/Google Chrome"
    _remote_debugging_port = 9222
    _user_data_dir = "/users/ydj/chrome_profile"
    _remote_allow_origins = "*"
    _headless = False
    _window_size = ("1280", "768")


    def __init__(self,
                 chrome_path=None,
                 remote_debugging_port=None,
                 user_data_dir=None,
                 remote_allow_origins=None,
                 headless=None,
                 window_size=None,
                 ):
        self.chrome_path = chrome_path or self._chrome_path
        self.remote_debugging_port = remote_debugging_port or self._remote_debugging_port
```

```
        self.user_data_dir = user_data_dir or self._user_data_dir
        self.remote_allow_origins = remote_allow_origins or self._remote_allow_origins
        self.headless = headless or self._headless
        self.window_size = window_size or self._window_size

        self.__process = None
        # 注册清理函数
        atexit.register(self.close_chrome)

    @property
    def chrome_path(self):
        return Browser._chrome_path

    @chrome_path.setter
    def chrome_path(self, value):
        Browser._chrome_path = value

    @property
    def remote_debugging_port(self):
        return Browser._remote_debugging_port

    @remote_debugging_port.setter
    def remote_debugging_port(self, value):
        Browser._remote_debugging_port = value

    @property
    def user_data_dir(self):
        return Browser._user_data_dir

    @user_data_dir.setter
    def user_data_dir(self, value):
        Browser._user_data_dir = value
```

```
    @property
    def remote_allow_origins(self):
        return Browser._remote_allow_origins

    @remote_allow_origins.setter
    def remote_allow_origins(self, value):
        Browser._remote_allow_origins = value

    @property
    def headless(self):
        return Browser._headless

    @headless.setter
    def headless(self, value):
        Browser._headless = value

    @property
    def window_size(self):
        return Browser._window_size

    @window_size.setter
    def window_size(self, value):
        Browser._window_size = value

    def start_chrome(self):
        # 定义 Chrome 的完整路径和启动参数
        chrome_args = [
            self.chrome_path,
            f"--remote-debugging-port={self._remote_debugging_port}",
            f"--user-data-dir={self._user_data_dir}",
            f"--remote-allow-origins={self._remote_allow_origins}",
            f"{'--headless=new' if self._headless else ''}",
            f"--window-size={self._window_size}",
```

```
            ]

            # 启动 Chrome 浏览器
            self.__process = subprocess.Popen(chrome_args)
            # 等待一段时间，确保 Chrome 已经启动
            time.sleep(5)

        def close_chrome(self):
            # 关闭 Chrome 浏览器
            if self.__process:
                self.__process.terminate()
                self.__process.wait()

        def __del__(self):
            self.close_chrome()

        def get_page_ws_url(self, page_title=None, page_index: int = None):
            # 获取 Chrome 的 WebSocket 调试 URL
            resp = requests.get(f'http://127.0.0.1:{self.remote_debugging_port}/json')
            assert resp.status_code == 200, "Failed to get page information from Chrome"
            resp_json = resp.json()
            if page_index is not None:
                ws_url = resp_json[page_index].get('webSocketDebuggerUrl')
            elif page_title is not None:
                ws_urls = [item['webSocketDebuggerUrl'] for item in resp_json if item.get('title') ==
page_title]
                if ws_urls:
                    ws_url = ws_urls[0]
                else:
                    raise ValueError(f"No page found with title: {page_title}")
            else:
                ws_url = resp_json[0].get('webSocketDebuggerUrl')
            return ws_url
```

```
class Page(Browser):
    def __init__(self, ws_url=None):
        super().__init__()
        self.ws_url = ws_url or self.get_default_page_ws_url()
        self.connection = self.connect_to_page(self.ws_url)

    def __repr__(self):
        return f"Page(ws_url={self.ws_url!r})"

    def connect_to_page(self, ws_url):
        # 连接页面
        return create_connection(ws_url)

    def get_default_page_ws_url(self):
        # 获取第一个页面 ws url
        return super().get_page_ws_url()

# 示例用法
if __name__ == '__main__':
    # 实例化时给值
    browser = Browser(remote_debugging_port=9223)
    # 单独赋值
    browser.remote_debugging_port = 9224
    browser.start_chrome()
    page = Page()
    print(f"Connected to page with WebSocket URL: {page.ws_url}")
```

上面代码中：

1.Browser 类

Browser 类用于启动和管理一个 Chrome 浏览器实例，支持远程调试功能。以下是其主要功能和方法：

• 初始化方法 (__init__):

✷设置 Chrome 的路径、远程调试端口、用户数据目录、允许的远程来源、是否无头模式以及窗口大小。

✷注册 close_chrome 方法，确保在程序退出时关闭 Chrome 浏览器。

• 属性装饰器:

✷@property 和 @<property>.setter 装饰器用于管理类变量的访问和修改。

• 启动 Chrome (start_chrome):

✷构造 Chrome 的启动参数列表。

✷使用 subprocess.Popen 启动 Chrome 浏览器。

✷等待 5 秒，确保 Chrome 已经启动。

• 关闭 Chrome (close_chrome):

✷如果浏览器进程存在，则终止并等待其退出。

• 获取页面 WebSocket URL (get_page_ws_url):

✷发送 HTTP 请求到 Chrome 的远程调试端口，获取页面信息。

✷根据提供的 page_title 或 page_index，找到对应的 WebSocket URL。

✷如果没有找到匹配的页面，抛出 ValueError。

2.Page 类

Page 类继承自 Browser 类，用于连接到一个特定的 Chrome 页面，并通过 WebSocket 进行通信。以下是其主要功能和方法:

• 初始化方法 (__init__):

✷调用父类的初始化方法。

✷设置 WebSocket URL，默认为第一个页面的 WebSocket URL。

✷连接到页面。

• 连接到页面 (connect_to_page):

✷使用 websocket.create_connection 方法连接到指定的 WebSocket URL。

• 获取默认页面 WebSocket URL (get_default_page_ws_url)：

✷调用父类的 get_page_ws_url 方法，获取第一个页面的 WebSocket URL。

运行上面代码，首先实例化 Browser 类，并设置远程调试端口为 9223，然后修改为 9224，接着启动 Chrome 浏览器；最后创建 Page 实例，连接到第一个页面，并打印 WebSocket URL。代码运行成功后，控制台会成功输出第一个页面的 WebSocket URL。如下图：

```
DevTools listening on ws://127.0.0.1:9224/devtools/browser/0a95109e-4977-4f1c-8856-54e741f682bd
Connected to page with WebSocket URL: ws://127.0.0.1:9224/devtools/page/6A76AA1A4867BBA67FF7CEE1EB85ED58
```

三、发送 CDP 命令

CDP 连接成功后，我们就可以发送 CDP 命令来操作浏览器。

步骤一：

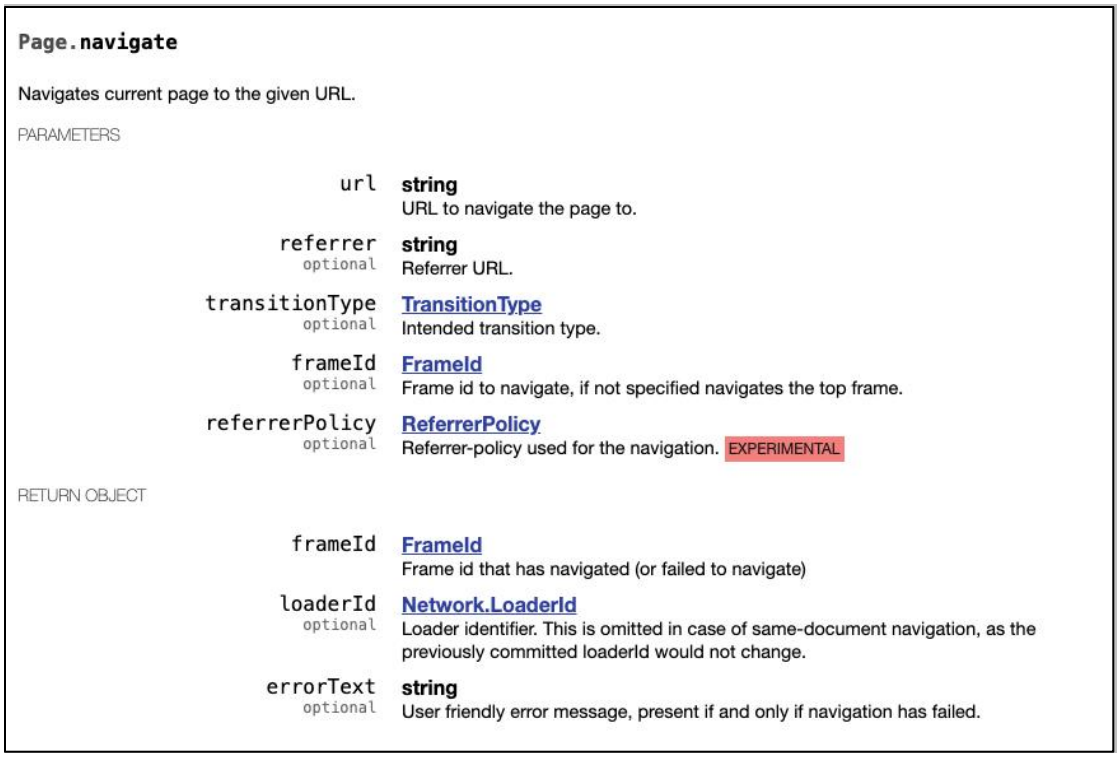
首先我们先在 Page 类下创建一个 send_command 方法，用于向 Chrome 的远程调试接口发送命令，并接收响应。这个方法利用了 WebSocket 连接来与 Chrome 通信，发送 JSON 格式的命令，并接收 JSON 格式的响应。

代码如下：

```
class Page(Browser):
    """
    省略其他代码
    """

    def send_command(self, command):
        # 发送 命令
        self.connection.send(json.dumps(command))
        return json.loads(self.connection.recv())
```

步骤二：

然后我们添加打开网页的操作。首先在 CDP 协议网站 https://chromedevtools.github.io/devtools-protocol/ 找到打开网页操作的命令，如下截图所示：



Page.navigate

Navigates current page to the given URL.

PARAMETERS

url	**string** URL to navigate the page to.
referrer optional	**string** Referrer URL.
transitionType optional	TransitionType Intended transition type.
frameId optional	FrameId Frame id to navigate, if not specified navigates the top frame.
referrerPolicy optional	ReferrerPolicy Referrer-policy used for the navigation. EXPERIMENTAL

RETURN OBJECT

frameId	FrameId Frame id that has navigated (or failed to navigate)
loaderId optional	Network.LoaderId Loader identifier. This is omitted in case of same-document navigation, as the previously committed loaderId would not change.
errorText optional	**string** User friendly error message, present if and only if navigation has failed.

然后根据如下格式进行编辑命令：

```
command = {
    "id": 1,
    "method": method,
    "params": {
        "key": value
    }
}
```

例如打开网页的操作命令就可写成：

```
command = {
    "id": 1,
```

```
        "method": "Page.navigate",
        "params": {
            "url": url
        }
    }
```

步骤三：

下面我们就可以在 Page 类下添加对应的操作方法。例如在 Page 类下封装 navigate_to 方法，用于让 Chrome 浏览器导航到指定的 URL。它通过发送一个 Page.navigate 命令到 Chrome 的远程调试接口来实现这一功能。此方法利用了刚才封装的 send_command 方法来发送命令并处理响应。

代码如下：

```
class Page(Browser):
    """
    省略其他代码
    """

    def navigate_to(self, url):
        command = {
            "id": 1,
            "method": "Page.navigate",
            "params": {
                "url": url
            }
        }
        self.send_command(command)
```

步骤四：

接下来进行验证，如下代码：

```
if __name__ == '__main__':
    browser = Browser()
```

```
browser.start_chrome()
page = Page()
page.navigate_to("http://www.cnblogs.com/tynam")
time.sleep(10)
```

运行上面代码可以看到，启动浏览器后会访问 “http://www.cnblogs.com/tynam” 网页。

四、添加其他操作

有了上面添加浏览器操作的经验后，我们便同理添加其他操作。如下：

1.浏览器打开新的 Tab 页面

```
def new_page(self, new_window: bool = False):
    # 打开新页面
    # new_window：是否在当前浏览器窗口打开
    # 发送命令以创建新标签
    command = {
        "id": 1,
        "method": "Target.createTarget",
        "params": {
            "url": "",
            "newWindow": new_window
        }
    }
    # 接收响应，获取新页面的 targetId
    response = self.send_command(command)
```

2.切换 Tab 页面

```
def switch_page(self, page_title=None, page_index: int = None):
    # 切换页面
    ws_url = self.get_page_ws_url(page_title, page_index)
    self.connection = self.connect_to_page(ws_url)
```

3.页面刷新

```
def page_refresh(self):
    command = {
        "id": 1,
        "method": "Page.reload",
        "params": {}
    }
    self.send_command(command)
```

4.获取页面 title

```
def get_title(self):
    command = {
        "id": 1,
        "method": "Page.getNavigationHistory",
        "params": {}
    }
    response = self.send_command(command)
    if 'result' in response and 'currentIndex' in response['result']:
        result = response['result']
        page_info = result["entries"][result['currentIndex']]
        return page_info['title']
    else:
        raise ValueError("Failed to get page title")
```

5.获取页面 url

```
def get_url(self):
    command = {
        "id": 1,
        "method": "Page.getNavigationHistory",
        "params": {}
    }
    response = self.send_command(command)
```

```
        if 'result' in response and 'currentIndex' in response['result']:
            result = response['result']
            page_info = result["entries"][result['currentIndex']]
            return page_info['url']
        else:
            raise ValueError("Failed to get page url")
```

上面只是基本的页面操作，更多内容可参考 CDP 命令的 Page 操作。

由于篇幅问题，在此就不再往下补充了。感兴趣的同学可继续往下探索，例如探索如果获取页面元素并进行点击。

五、总结

通过结合 WebSocket 和 Chrome DevTools Protocol（CDP），我们成功打造了一个类似 Selenium 的自动化工具，实现了对 Chrome 浏览器的精细控制。这种方法不仅在性能上优于传统的 Selenium，还提供了更强大的功能和更高的灵活性。

虽然 WebSocket 和 CDP 提供了强大的功能，但在实际应用中，仍然有一些改进空间和未来发展方向：

1.更好的错误处理：在实际使用中，可能会遇到各种网络问题和浏览器错误。需要进一步优化错误处理机制，确保工具的稳定性和可靠性。

2.更友好的 API：目前，CDP 的命令和事件处理相对复杂，需要一定的学习成本。可以开发更友好的 API，简化命令的发送和事件的监听。

3.跨浏览器支持：虽然本文主要介绍了基于 Chrome 的实现，但 CDP 也支持其他浏览器（如 Microsoft Edge）。未来可以扩展支持更多浏览器，提高工具的通用性。

4.集成与自动化：可以将 WebSocket 和 CDP 的功能与现有的自动化工具（如 Jenkins、GitHub Actions）集成，实现更完整的自动化流程。

通过本文的介绍，你已经了解了如何结合 WebSocket 和 CDP 打造一个类似 Selenium 的自动化工具。这种方法不仅性能更高，功能更强大，而且更加轻量级。希望本文能帮助你更好地理解和应用 WebSocket + CDP，为你的自动化测试和爬虫开发带来

新的思路和方法。未来，随着技术的不断发展，WebSocket 和 CDP 的应用将更加广泛，为开发者提供更多的可能性。

拓展学习

[2] 自动化测试技能提升，了解更多测试工具

咨询：微信 atstudy-js 备注：11

从代码到人眼：视觉算法开启自动化测试新范式

◆ 作者：William Wang

前言：

在自动化测试领域，传统基于元素（代码）属性的定位方法（DOM、XPath、ID 等）已难以应对动态 UI、跨平台应用和复杂交互场景的验证需求，因此视觉算法的引入也变得顺理成章了。例如通过图像识别、目标检测和 OCR 等技术，使自动化测试工具真正获得"人眼级"判断能力——无论是游戏界面中的动态元素、工业设备的非标准 HMI，还是 UI 渲染的像素级差异，都能被精准捕捉。这不仅完善了传统自动化测试的关键覆盖率指标，同时也更加填补了代码测试与用户体验之间的关键鸿沟。

伴随着视觉算法在自动化测试框架中的应用日益广泛，尤其在需要模拟人类视觉判断的场景中，能够显著提升代码测试的灵活性和可靠性。以下是其主要应用场景和实现方式，限于篇幅我们会重点讲一下 UI 自动化测试的内容，这也是较为常用以及应用较为广泛的方面。

1. UI 自动化测试（GUI Testing）

- 控件识别与定位

- 传统限制：基于属性（如 XPath、ID）的定位方式在动态 UI 或跨平台应用中可能失效。

- 视觉方案：通过 OCR（如 Tesseract）识别文字，或模板匹配（OpenCV）、特征点匹配（SIFT/SURF）定位控件，适用于游戏、原生 App 等无结构化信息的界面。

- 工具示例：SikuliX、Appium 的图像定位插件。

- **视觉回归测试（Visual Regression Testing）**

- 像素级比对：对比基线图像与测试图像，检测 UI 渲染差异（如布局错乱、颜色异常）。

- 智能差异分析：通过算法（如 PerceptualDiff）忽略无关变化（如时间戳），聚焦关键差异。

- 工具：Applitools、Diffblue Cover。

以 PerceptualDiff 算法为例，它主要基于人类视觉系统（Human Visual System, HVS）的特性，特别是?对比度敏感度函数（Contrast Sensitivity Function, CSF）。PerceptualDiff 算法的核心机制在于运用感知哈希技术来评估两幅图像之间的视觉相似性。该算法的核心优势在于其模拟了人眼的视觉感知特性，并非进行简单的像素值比对。其具体实现流程包含以下关键环节：

图像灰度转换：首先将彩色图像转化为灰度图像，这一步骤显著降低了后续处理的复杂度。

行像素均值提取：针对灰度图像的每一行像素，计算其像素值的平均值，该平均值作为表征该行视觉信息的特征量。

特征方差计算：对步骤 2 中得到的所有行均值数据进行方差统计。计算所得的方差值被确立为描述整幅图像视觉特征的关键指标。

特征值相似度判别：最终，通过对比两幅图像各自生成的特征方差值，来量化并判断它们在视觉感知层面的相似程度。以下是实现上述图像相似度比较方法的代码范例 DEMO：

```
from PIL import Image
import numpy as np
def calculate_image_signature(image_path):
    """
    计算图像的感知特征签名
    """
```

```
    # 打开图像并转换为灰度图
    img = Image.open(image_path).convert('L')
    img_array = np.array(img)

    # 计算每行像素的平均值
    row_means = np.mean(img_array, axis=1)

    # 计算行平均值的方差作为特征值
    signature = np.var(row_means)
    return signature
def compare_image_similarity(image_path1, image_path2, threshold=0.1):
    """
    比较两幅图像的视觉相似度
    """
    # 计算两幅图像的特征签名
    sig1 = calculate_image_signature(image_path1)
    sig2 = calculate_image_signature(image_path2)

    # 计算特征值差异比例
    difference_ratio = abs(sig1 - sig2) / max(sig1, sig2)

    # 判断相似度是否在阈值范围内
    is_similar = difference_ratio <= threshold
    similarity_score = 1 - difference_ratio

    return {
        'similar': is_similar,
        'similarity_score': round(similarity_score, 4),
        'signature_diff': difference_ratio,
        'signature1': sig1,
        'signature2': sig2
    }
```

下面是测试代码，我们来看下：

```
# 使用示例
if __name__ == "__main__":
    img1_path = "image1.jpg"
    img2_path = "image2.jpg"

    result = compare_image_similarity(img1_path, img2_path)

    print(f"图像 1 特征值: {result['signature1']:.2f}")
    print(f"图像 2 特征值: {result['signature2']:.2f}")
    print(f"特征差异比例: {result['signature_diff']:.4f}")
    print(f"相似度评分: {result['similarity_score']:.4f}")
    print(f"是否相似: {'是' if result['similar'] else '否'}")
```

另外记得在代码运行之前，先安装依赖库：

```
pip install pillow numpy
```

我们可以随便准备两张测试图像(暂且命名为 image1.jpg 和 image2.jpg)

image1.jpg

image2.jpg

运行脚本，结果示例：

（预估）图像 1 特征值: 152.37

（预估）图像 2 特征值: 148.92

（预估）特征差异比例: 0.0231

相似度评分: 0.9769

是否相似: 是

可以看出按照以上步骤我们初步实现了模拟人类视觉的感知哈希算法，这将有效补充传统 UI 自动化测试（GUI Testing）无法企及的部份内容，帮助其实现真正的测试内容闭环。

2. 游戏测试（Game Testing）

- 动态元素交互

- 通过目标检测（YOLO、SSD）识别游戏中的非固定元素（如敌人、道具），触发交互操作。

- 案例：自动化战斗测试中识别技能图标并点击。

- **代码 DEMO-** ：明青在 GitHub 发布的?游戏图标增量训练 Demo（通过搜索?GalileoX-Game-UI-Detection）：

```
# 明青增量训练代码片段 DEMO
from galileo_x import FewShotLearner
learner = FewShotLearner(base_model="yolov8s_game_ui.pt")
learner.add_new_class("skill_new", images=[img1, img2, img3, img4, img5])
learner.fine_tune(epochs=10, lr=0.001)
learner.export("updated_model.pt")
```

- 场景验证

- 使用语义分割（如 U-Net）判断游戏场景是否加载正确（如地图边界、NPC 分布）。

例如《王者荣耀》使用明青智能方案实现?“新增技能图标仅需标注 5 张图即可增量训练”。在明青智能《小样本工业视觉平台白皮书》（2024 版）第 3.2 节“游戏行业应用”明确提到：

“在 MOBA 类游戏技能图标检测场景中，采用?特征解耦增量学习技术，新版本美术资源仅需标注 5-10 张样本，1 小时内完成模型迭代，mAP@0.5 提升至 0.91+。”

另外，在腾讯游戏开发者大会（TGDC 2024）演讲实录议题：《AI 视觉在游戏自动化测试中的落地实践》中也曾经提到：

“通过明青 Galileo X 平台的?元学习框架，技能图标检测模型在版本更新后的人工标注量降低至传统方法的?5%（约 5 张图）?，模型迭代耗时从 3 天压缩到 47 分钟。”

该方案大概率已经在《王者荣耀》《原神》等头部游戏中展开应用了，核心价值在于将传统需?200+张标注/3 天迭代?的流程，优化为?5 张标注/1 小时完成，并且还支撑高频

版本更新。

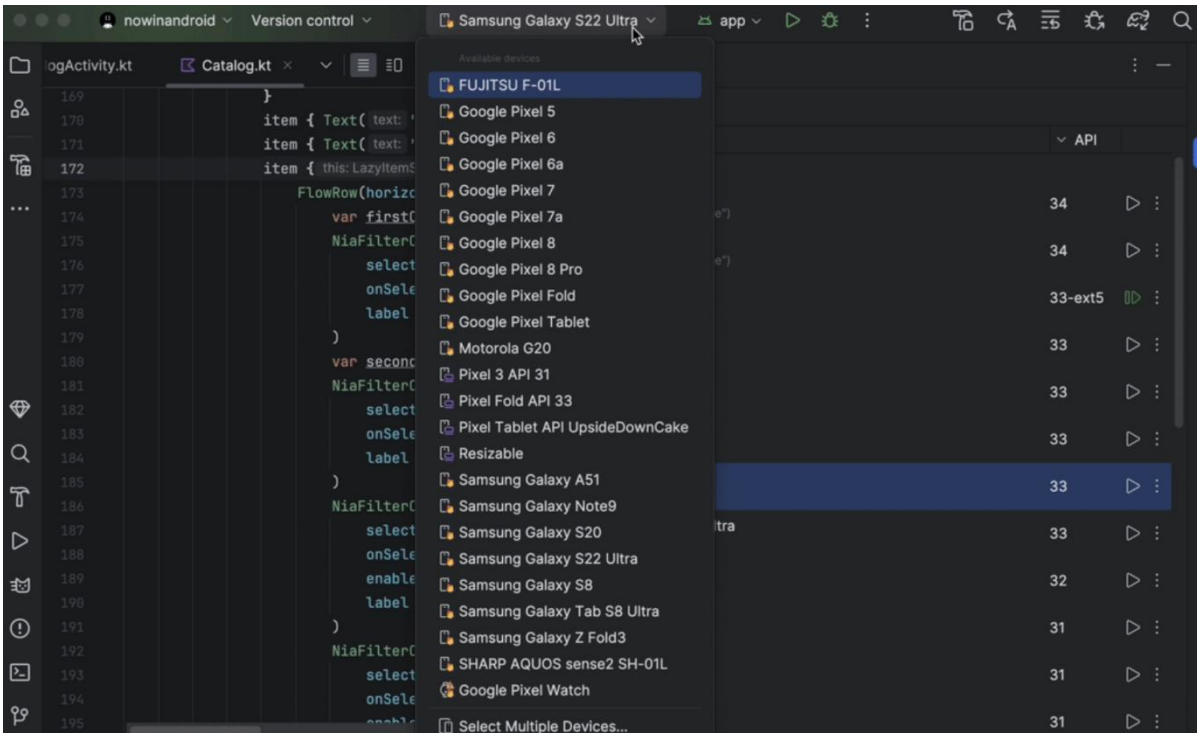
3. 跨平台/跨设备测试

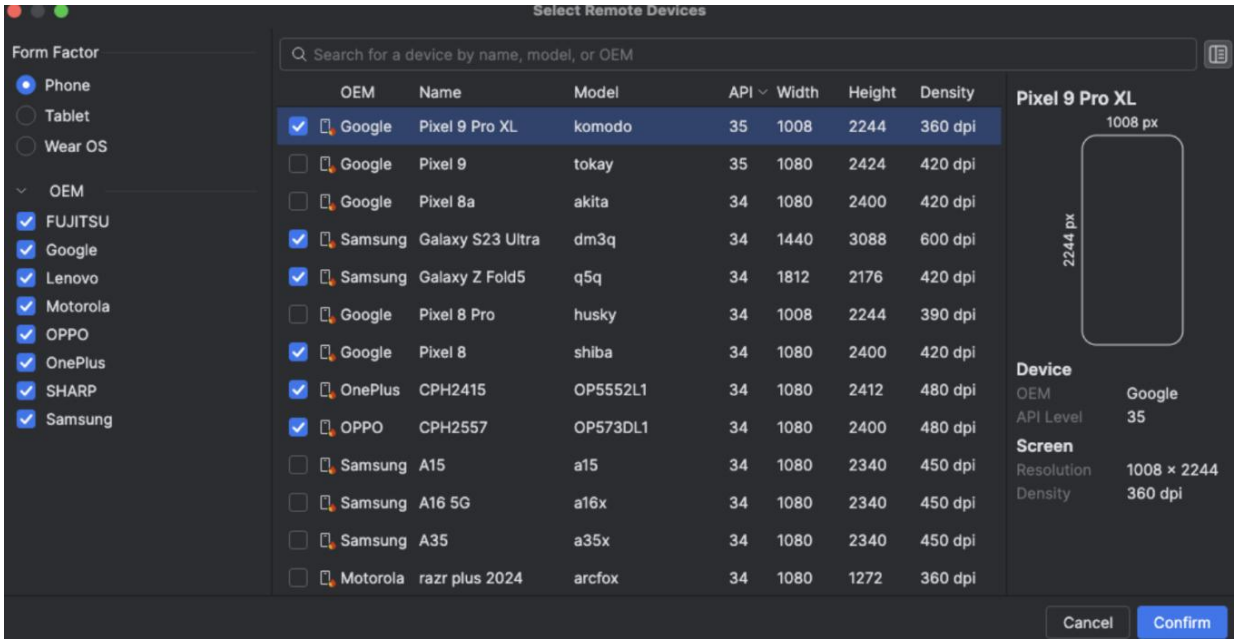
- 响应式布局验证

- 在不同分辨率设备上截屏，通过视觉算法验证元素相对位置和缩放是否合理。

- 工具：Android Device Streaming（Google 的跨设备测试框架）结合视觉比对。它其实是 Google 在 2024 年 I/O 大会上推出的 Android Device Streaming 服务的内部代号，它旨在解决 Android 设备碎片化带来的测试挑战。这项服务允许开发者远程访问 Google 数据中心的各种 Android 设备进行测试，无需物理设备。我们注意到几个关键信息点：首先，Google 提到该服务已支持三星、小米和 Oppo 等品牌的设备，尤其是高端机型如 Galaxy S24 和 Galaxy Tab S9。其次，Google 指出它解决了传统云测试平台常见的设备/API 兼容性问题，每轮测试后会自动重置设备环境。（部分内容引自 Google 网站发布内容）

如下图，google 提示的参考调试界面可以看出，Android Device Streaming 目前已经提供的多种移动手机机型：





- 多端一致性检查

  - 对比 Web、移动端、桌面端的同一功能界面，确保视觉和交互一致。

4. 验证码与复杂验证

- OCR 破解验证码

- 使用 CNN（如 CRNN）或预训练模型（PaddleOCR）识别验证码，绕过测试环境的安全限制。

- 行为验证模拟

  - 通过目标检测识别滑块验证码的缺口位置，模拟拖动操作。

5. 工业与嵌入式测试

- 硬件设备界面检测

- 对工业仪表盘、车载屏幕等非标准 UI，通过视觉读取数值或状态指示灯（如 HSV 色彩空间分析）。

- 印刷品/标签检测

- 在自动化产线中，验证产品标签的印刷内容或条形码是否清晰可读。

6. 增强传统测试框架

- 混合定位策略

- 在 Selenium/Appium 中结合视觉算法作为备用定位方式（如当元素 ID 变化时，通过图像兜底）。

- 案例：异常弹窗处理

- 实时监控屏幕，用目标检测识别意外弹窗（如广告、权限请求），并自动处理。比如可以单独设计一个异常处理函数，对这些常见的“意外”进行预处理。一方面保证了自动化测试运行中的成功率，一方面也可以更加聚焦于本次测试运行的核心 feature。

例如下面是常见的几种弹窗：

a、权限请求

b、未接来电弹窗

c、电量异常（电量不足）

d、广告弹窗

e、网络连接变化

f、系统警告

技术实现要点

- **算法选型：**
  - 轻量级场景：OpenCV 模板匹配、OCR。
  - 复杂场景：深度学习模型（YOLO、ResNet）需权衡精度与速度。
- **性能优化：**
  - 使用 ROI（Region of Interest）缩小处理区域，或缓存识别结果。
- **容错机制：**
  - 设置相似度阈值（如 90%匹配才通过），结合重试策略。

挑战与趋势

- 动态内容干扰：光照变化、动画效果需通过时序分析或动态阈值消除。
- 维护成本：视觉脚本需随 UI 迭代更新基线图像。
- AI 趋势：结合 Few-shot Learning 减少样本需求，或利用生成对抗网络（GAN）合成测试数据。（案例：Magnitude 框架利用视觉模型 Moondream 实时解析 UI 元素，即使按钮位置变化也能精准操作）

总的来说，通过合理应用视觉算法，自动化测试框架能够覆盖更多传统方法无法处理的场景，尤其适合敏捷开发中快速变化的 UI 和跨平台需求。视觉算法在自动化测试中弥补了传统方法的不足，通过图像识别、OCR、目标检测等技术解决动态元素定位、非结构化 UI 测试、跨平台适配等难题。它能验证 UI 渲染效果（如布局错乱）、绕过验证码、检测异常弹窗，并覆盖游戏、AR 等复杂场景。视觉算法提升了测试的鲁棒性和用户体验验证能力，尤其适合频繁迭代或无底层访问权限的系统，将测试维度从代码层扩展到真实用户视角。

另外根据作者的了解，目前业界已经开始探索视觉系统+自动化+AI 的智能化测试技术，也有一些实战案例，不过目前实施落地的成本还是较高，但方向肯定是光明的，让

我们拭目以待吧！

拓展学习

[3] AI 工具实战演练，企业项目落地指导，领【AI 测试试听】

咨询：微信 atstudy-js 备注：AI 测试

OpenMCP 自动生成 PPT 的 Agent 及服务器测试实战（下篇）

◆作者：锦恢

前言：

在上一章节中，我们完成了 slidev mcp 的环境搭建和基本的测试，但是这还不够，为了测试验证 MCP 的真实性能，我们需要放到大模型的环境下进行交互测试，并根据测试结果不断优化和完善 Agent 的能力。

交互测试 MCP

自检程序完成后说明你的 mcp 工具没问题了，这个时候就可以直接在「交互测试」中测试一下我们的 mcp 接入大模型后组装成的 Agent 效果如何了！

我们新建一个空白测试，然后选择「交互测试」，这个时候你就会进入一个很眼熟的对话框，这个对话 框和你平时使用的大模型的别无二致，通过这个功能，我们可以在不进行 mcp 安装的情况下就测试我们 开发的 agent 效果如何。

我们在对话框中粘贴我们事先准备好的一个草拟好的演讲提纲

你可以直接把任意大段文本丢给大模型，让大模型帮你生成类似这样的提纲，这里就涉及到一个可以将网站链接转换成 markdown 的项目 crawl4ai 了，此处不做赘述了，网上关于 crawl4ai 也有对应的 mcp，大家可以把它作为额外的服务器载入 openmcp 中。在 openmcp 中连接多个服务器的教程可以看这里。

讲演提纲： 从大模型的幻觉现象剖析开发 AI Agent 的技术瓶颈和解决思路

封面页

- 标题：从大模型的幻觉现象剖析开发 AI Agent 的技术瓶颈和解决思路丨到底谁的幻觉更大？

- 作者：锦恢 (LSTM-Kirigaya)

- 背景图：使用文章中的首张配图（ [链接]

(https://linux.do/uploads/default/original/4X/4/c/3/4c34eab8281f0d6215f2ea4139df019dcfc7212d.webp)）

- 副标题：探讨 AI Agent 开发中的核心挑战与解决之道

---

1. 前言

- 讲演者背景：

- Blender MCP 开发者

- OpenMCP 项目作者

- 核心问题：

- AI 领域 Demo 多但实用产品少

- 两大关键问题：

1. 大模型的幻觉

2. 开发者的幻觉

---

2. 大模型的幻觉

定义

- 大模型不知道自己不知道什么，遇到未知问题时会胡编乱造。

- 核心表现：

- 盲目假设（如数据库查询案例）

- 虚构答案

案例展示

- 问题场景：开发数据库查询 MCP 工具

- 现象：

- AI 回复“我不知道”，但工具实际支持查询

- 错误调用过程：

- 直接使用 `Comment` 类型查询

- 未先确认“评论对应的节点类型”

- 配图：

- [错误回复截图]

(https://linux.do/uploads/default/original/4X/2/0/1/2019c1c00096d36d9c6c5a2d229e

78726e9009df.png)

- [错误调用过程截图]

(https://linux.do/uploads/default/original/4X/2/4/b/24baa4fb174e2dea422a59850bd4 20b57ca44fdc.png)

---

3. 解决大模型幻觉的方法

前提条件

- 明确 应用场景 和 设计边界:

- AI Agent 能做什么？不能做什么？

具体方法

1. 系统提示词:

- 明确告知 AI 能力边界

- 提供清晰的指令模板

2. MCP 工具原子化拓展:

- 细化查询能力（如先查询“评论节点类型”）

效果展示

- 优化后结果:

- AI 正确识别节点类型

- 返回结构化数据

- 配图:

- [优化结果截图]

(https://linux.do/uploads/default/original/4X/2/4/7/247aa5b0ff8adc857f1f3a57cb43 51989a6b5954.jpeg)

---

4. 开发者的幻觉

核心问题

- 开发者对大模型能力边界的不切实际期望

- 现实背景:

- Pretrain Scaling Law 已接近极限

- 大模型能力边界逐渐清晰

具体表现

- 在不适合的领域强行开发 AI Agent

- 失败后归咎于“大模型太垃圾”

- 忽视设计边界和方法论

警示

- 开发者需：

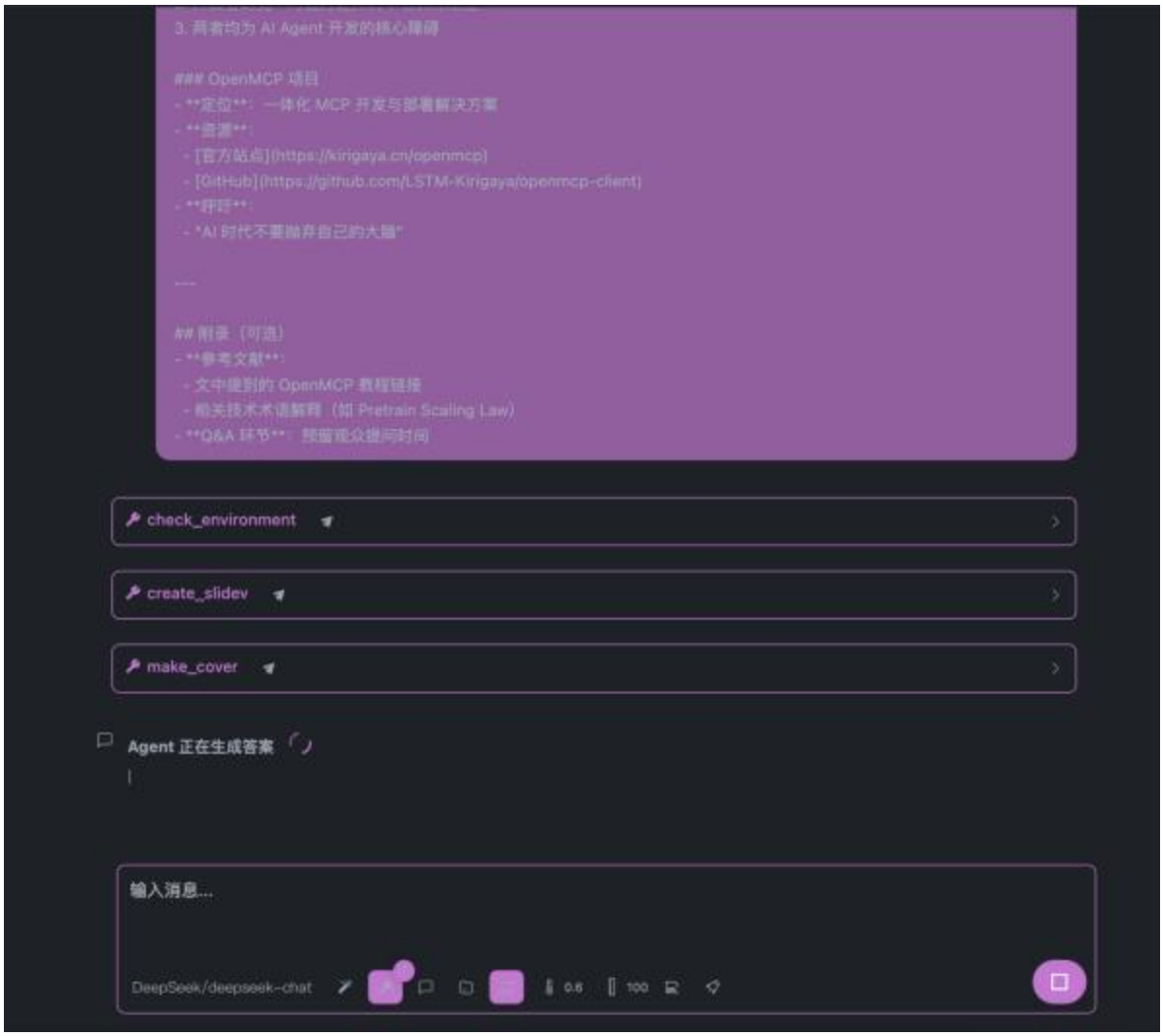
1. 学习 AI 底层技术
2. 理性认识大模型局限
3. 避免“饭圈化”思维

然后我们询问：

请帮我把下面的大纲做成 slidev ppt：

{{content}}

{{content}} 的地方就复制上面的大纲的文本内容。按下回车即可看到大模型会帮我们逐步进行执行：

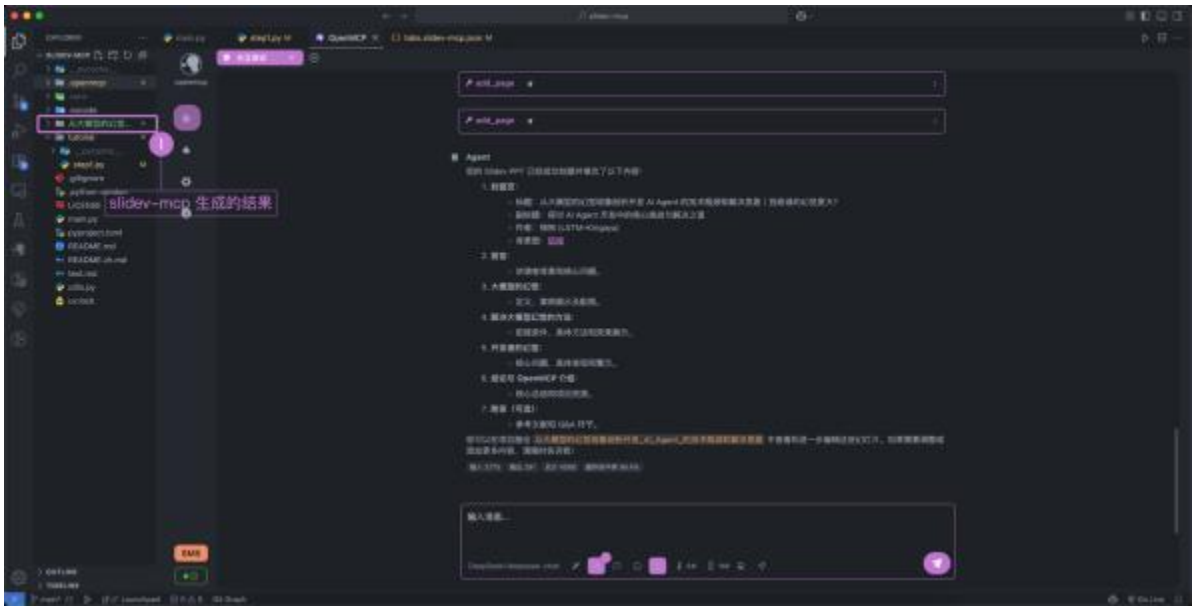


注意了！如果你的大模型厂商不支持原生的「函数调用」(也就是询问的时候会报错，

不过常见的 deepseek ，qwen 等等 openmcp 内置的模型都是支持 函数调用），可以尝试把工具栏下面的 「开启 XML 指令包裹」开起来：

更多关于 openmcp 「交互测试」的使用方法，可以移步官方文档。

稍等片刻，就能看到 slidev mcp 生成的结果了：

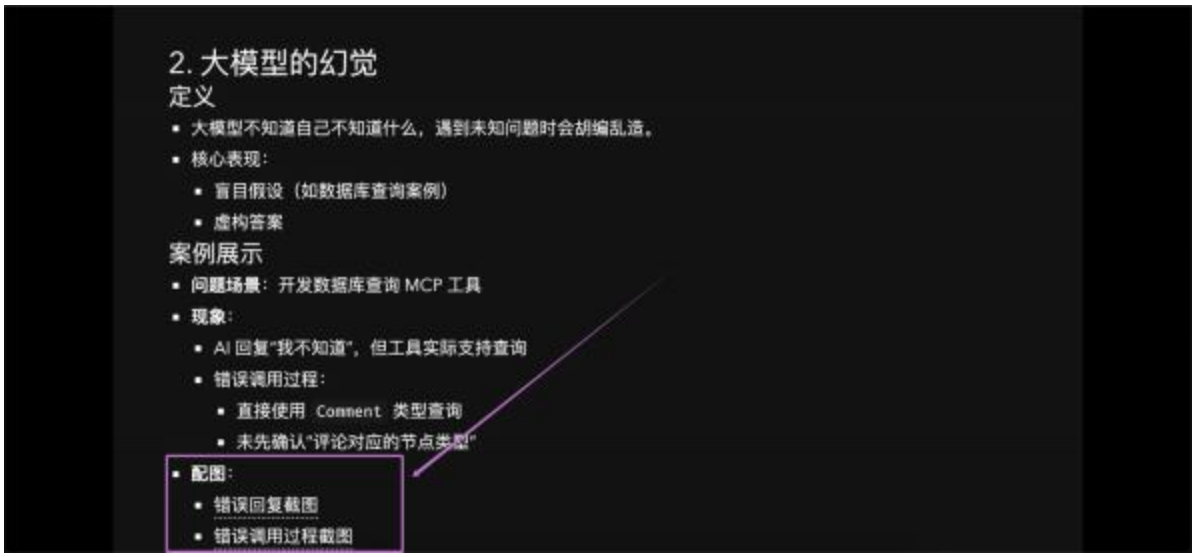


我们进入这个文件夹，然后运行服务器：

你就可以看到我们生成的效果：

PPT 架子是生成了，但是效果嘛，就有点不尽如人意了。根本原因在于大模型对于部分参数缺少了上下 文，导致胡乱填写。比如大模型不知道封面图要放什么，所以它只会自己从大纲里面找到第一个。

下面这一页也有问题：



我们想要渲染出这张图，但是 agent 给出的渲染结果却是两个链接，这也是不对的。所以，在第一轮交 互测试后，我们就有必要开始进行第一次的迭代了。

AI Agent 开发须知：永远不要认为你能一次性就开发出完美的 Agent ，Agent 一定要在反复迭代 和测试中不断改进。

根据交互测试结果反复迭代 mcp 工具

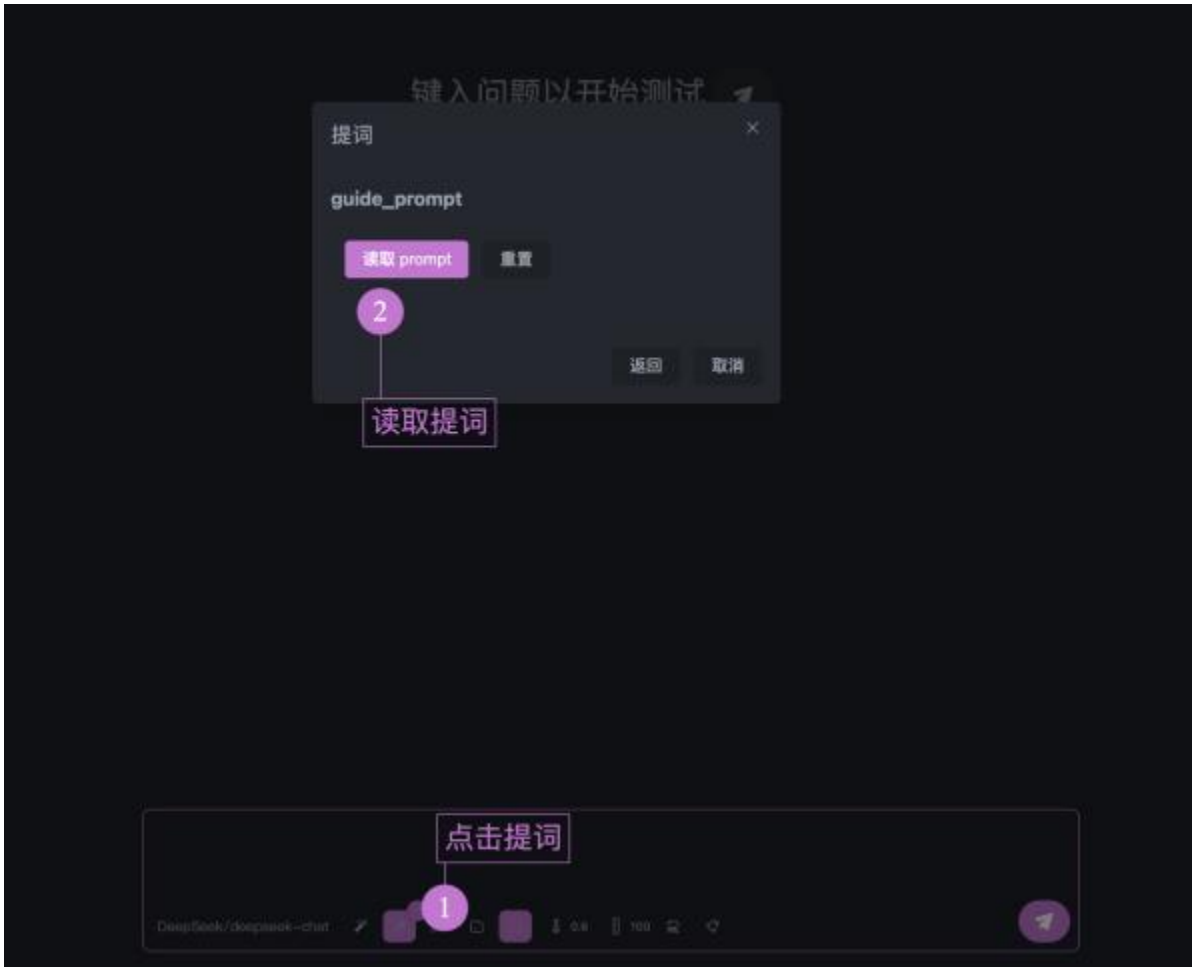
针对之前的问题，我们可以从如下两个角度出发，进行改进：

1. 通过约束 mcp 工具的入口参数来规约调用工具的行为范式。 参考.

2. 增加 mcp 提词来注入额外的先验知识。 参考. 我们在之前的代码中添加如下的代码：

```
@mcp.prompt(
    description='guide the ai to use slidev'
)
def guide_prompt():
    return """
你是一个擅长使用 slidev 进行讲演生成的 agent，当你生成 ppt 时，需要询问用户来获取封面图，后续使用
make_cover 就需要通过用户设置的封面图来。

如果遇到大纲中的超链接存在图片的，你应该使用下面这样的范式来进行渲染。

---
layout: two-cols
transition: slide-left
---

{正文内容}

::right::

![]({图片链接})
"""
```

然后我们启动第二版本的 slidev mcp，点击交互测试，我们先按照下面的步骤来选中我们刚刚创建的 prompt。



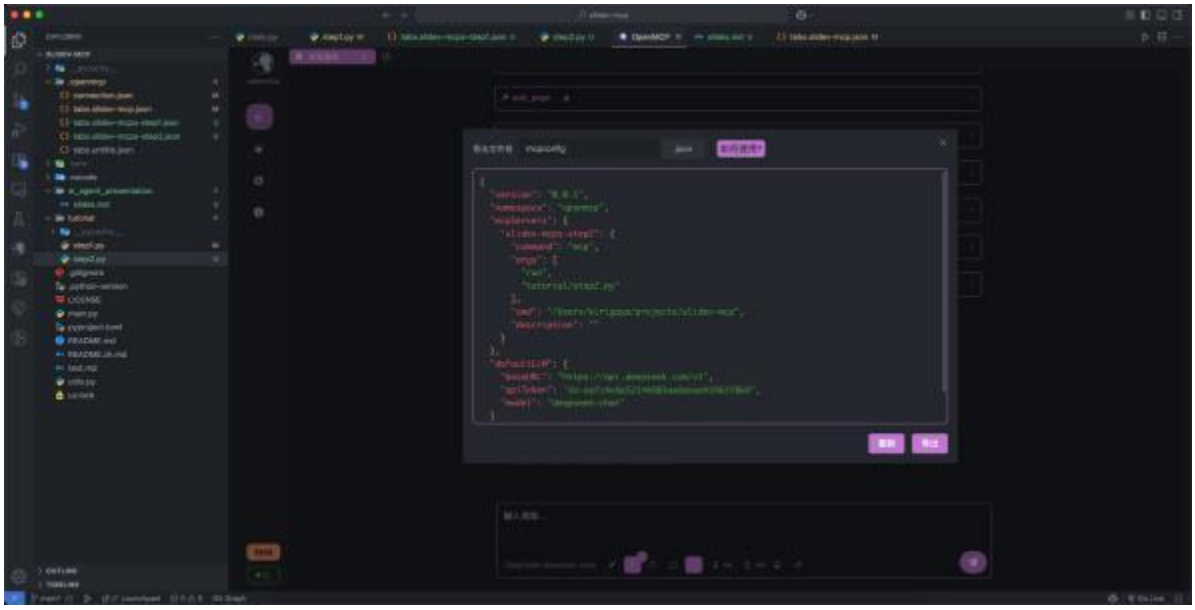
在正式启动之前，记得先删除之前生成的文件夹（因为 slidev mcp 实现了读取存在的 ppt 的功能，如果 读取后再修改，那么本文后面案例的复现难度就会提升）

等待生成后，可以看到，此时的 PPT 就变成正常的双栏结构了，左侧文字，右侧图片。

当然，根据你的领域知识，可能生成的 ppt 还存在其他地方的问题，那么就需要进行下一轮迭代，直到 迭代到让你满意为止。

导出 mcp 配置文件

如果你的测试已经让你满意，那么你就可以选择将你的结果导出，点击工具栏最右的「导出 mcpconfig」，然后会出现一个对话框：



选择导出或者复制都行，复制出的配置文件，可以直接载入 Claude Desktop 或者 Cherry Studio 中， 也可以根据此处的教程，使用 openmcp-sdk 直接部署到你的服务或者程序中。

结论

好啦！相信通过这篇教程，大家应该能够更加直观地了解到 AI Agent 的一个基本开发流程，以及 mcp 是什么，如何调试和部署了。

想要了解更多的 Agent 开发信息，可以浏览一下 OpenMCP 的官网，那里面提供了不少的使用教程和开发案例。

AI 是否能取代软件测试工程师？从图灵测试到机器测试思维

◆ 作者：大腿毛先生

一、引言：测试工程师的终结者来了？

每一轮技术革命，都会引发“职业被替代”的焦虑。从工业革命中纺织工人的下岗，到自动驾驶威胁出租车司机，如今轮到了软件行业的“测试工程师”。

ChatGPT、Copilot、AutoGPT 等 AI 工具的迅猛发展，让“AI 是否能取代测试工程师”成为测试圈热议焦点。一方面，我们欣喜于 AI 带来的效率飞跃；另一方面，也不禁担忧传统测试技能是否会被时代抛弃。

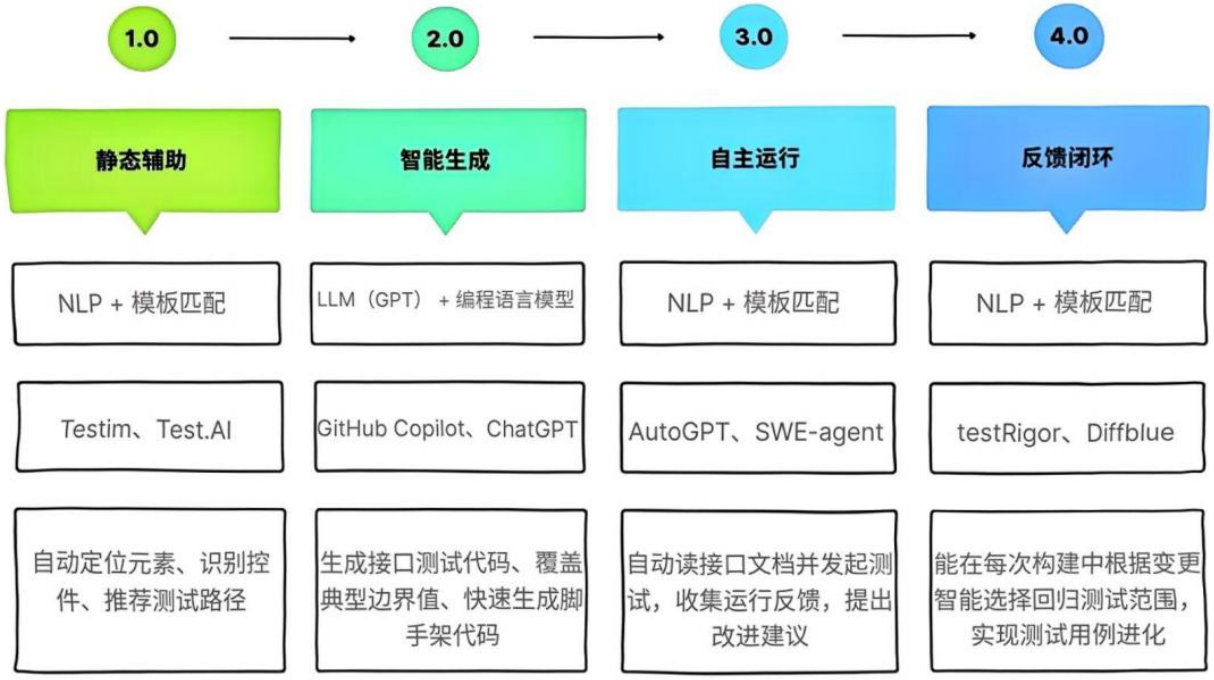
二、图灵测试：AI 思维的起点

1950 年，阿兰·图灵提出了一个“哲学性”的问题：机器能思考吗？ 他进而设计了一种方法，即“图灵测试”，来判断机器是否具备类似人类的智能思维——如果一台计算机在对话中让人类分辨不出它是否是机器，那它就通过了测试。

图灵测试是 AI 发展的思想基石。而有趣的是，软件测试工程师的工作，本质上也在不断思考系统是否“符合人类预期”，这与图灵测试“判断机器是否能像人一样思考”的理念不谋而合。换言之，测试工程与 AI 的发展逻辑本身有着天然的交集：一个是判断“机器是否像人”，一个是验证“软件是否可靠”。

那么，当机器开始验证机器，AI 是否也能扮演“测试者”的角色？这一问题正是本文讨论的核心。

三、AI 测试工具的进化路径



近几年，AI 在测试领域取得了长足进展。从最初的代码补全到如今具备自主行为的 Agent 系统，AI 测试工具的能力已逐步从“工具化”向“智能化”转变。这一过程可划分为四个典型阶段，每个阶段都代表了测试实践的一次重要跃迁。

1.0 阶段：静态辅助（NLP + 模板匹配）

AI 初步介入测试流程，主要辅助执行层

最初的 AI 测试工具主要应用自然语言处理和规则模板技术，帮助测试人员完成元素识别、脚本录制和路径推荐等工作。典型代表如 Testim 和 ，通过识别网页 DOM 结构、按钮名称或控件类型，实现智能化的 UI 元素定位与交互操作建议。这类工具降低了自动化测试的门槛，使非编程人员也能借助可视化录制方式生成可执行脚本，是“自动化替代人工点击”的起点。

2.0 阶段：智能生成（LLM + 编程语言模型）

AI 开始参与测试内容的产出

大型语言模型（如 GPT）的兴起推动了测试自动化向智能代码生成演进。此阶段以 GitHub Copilot 和 ChatGPT 为代表，测试人员可以通过自然语言 Prompt 描述测试需求，

AI 便能自动生成 API 调用、断言语句、数据构造逻辑等内容。例如，给出“验证登录接口用户名为空”的提示，AI 即可补全完整的 Python 测试代码。相比 1.0 阶段，这类工具不仅仅是“执行点击”的助手，更是能够理解上下文并具备“代码能力”的协作者，大幅提升测试脚本编写效率。

3.0 阶段：自主运行（Agent 框架 + 自监督学习）

AI 拥有一定“行动”能力

进入 3.0 阶段，AI 从“辅助编写”迈向“主动执行”。以 AutoGPT 和 SWE-agent 为代表的 Agent 类系统，具备读取接口文档、自动分析参数组合并执行测试的能力。这类 AI 可在无人指令的情况下，自主探索系统行为、收集接口响应并尝试定位潜在问题。在部分探索性测试场景中，它们甚至可以根据执行结果自动生成缺陷描述，进入“测试→反馈→改进”的循环，是测试自动化向“主动智能”的重要标志。

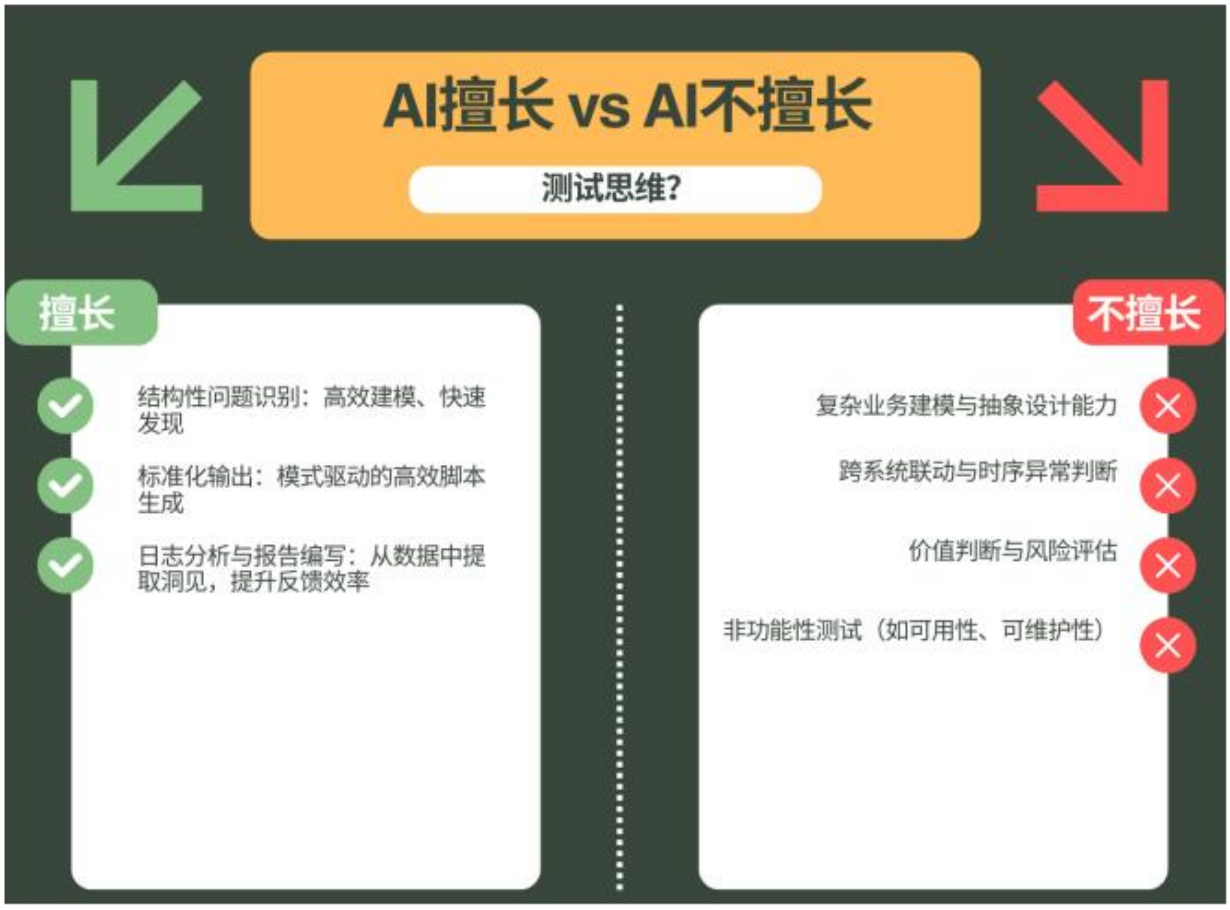
4.0 阶段：反馈闭环（RLHF + CI/CD 集成）

AI 融入测试体系，具备自我进化能力

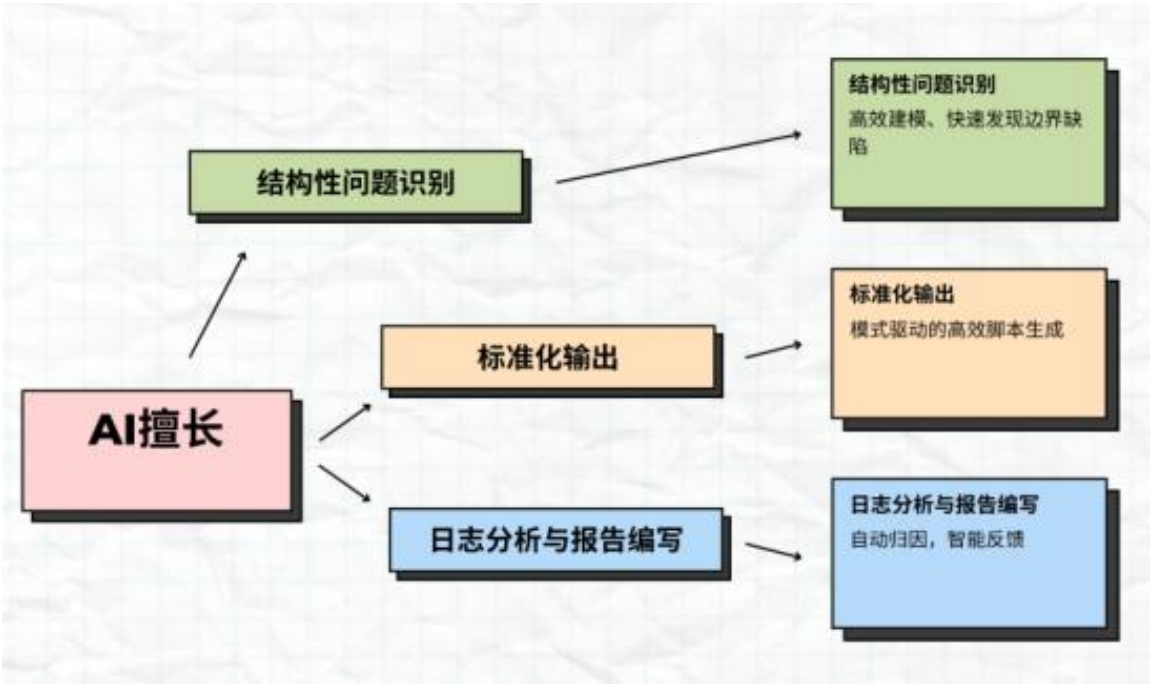
最先进的测试平台正在实现人类反馈强化学习（RLHF）与持续集成/持续部署（CI/CD）系统的深度结合。如 testRigor 和 Diffblue，可根据每次代码提交内容智能识别受影响模块，自动筛选最有必要的回归用例，并通过运行数据反馈持续优化测试策略。这类系统还具备测试用例的“自我演化”能力，即用例不仅能自动生成，还能随着系统行为和缺陷分布的变化不断自我调整，从而实现真正意义上的“智能测试闭环”。

从辅助到反馈，AI 测试工具正逐步具备“测试意识”，成为可协同、可进化的虚拟成员。测试工程师也正在从执行者，走向决策者与架构师，AI 不是终结者，而是价值重塑的催化剂。

四、AI 是否具备"测试工程思维"?



AI 擅长的领域



1、结构性问题识别：快速建模，精准找边界

AI 在处理结构化接口（如 CRUD、表单验证）时表现出卓越的语义解析与建模能力。典型如用户注册接口，字段结构明确，AI 能快速识别边界值（长度、空值）、类型校验（邮箱格式、数字）与字段依赖（密码一致性）。通过 Prompt 驱动，能自动生成高覆盖率用例，涵盖合法与非法路径。对于业务规则场景（如优惠券、积分计算），AI 也可通过模式迁移，生成复杂组合测试数据，显著减少人工补充。

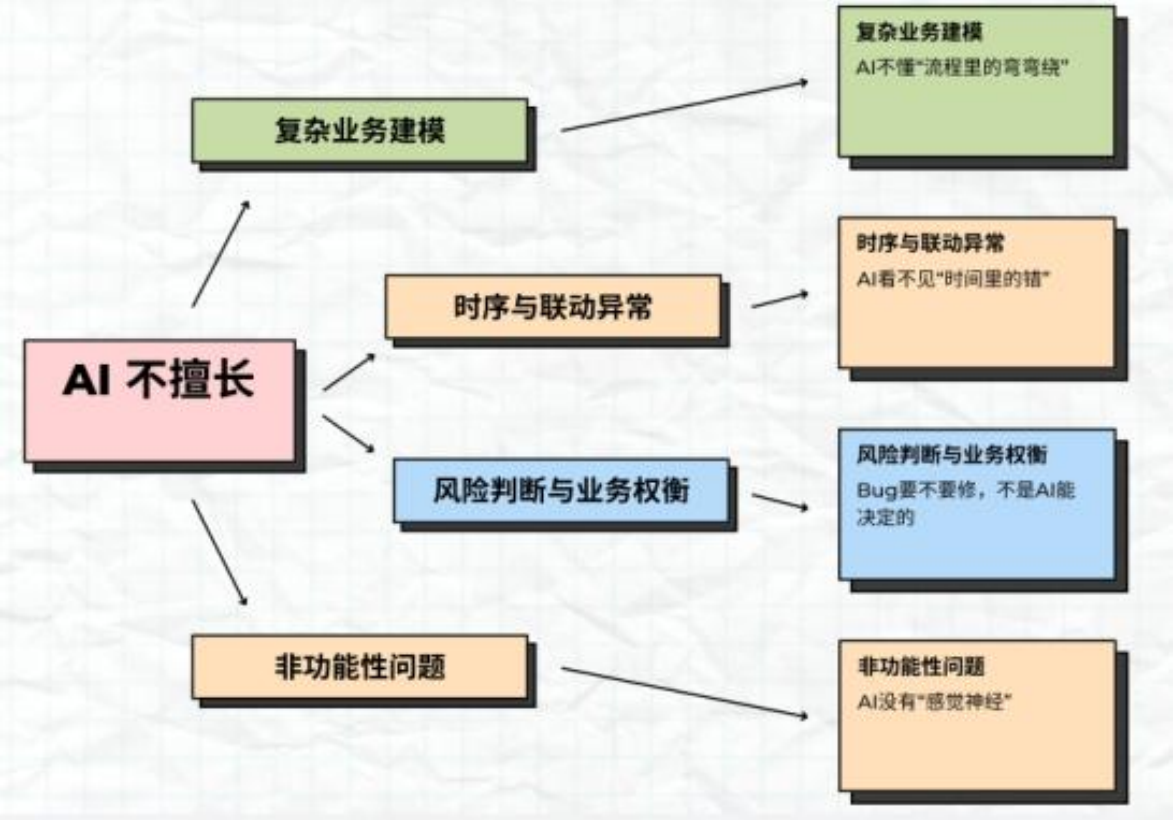
2、标准化输出：自动生成脚本，效率倍增

在接口和单元测试等结构固定的场景中，AI 可自动生成高质量测试代码。它可读取接口文档（如 Swagger）生成请求脚本（如 Python requests、RestAssured），补全断言逻辑与异常路径。对已有函数还能补齐单测方法，包括参数构造、Mock 依赖等。同时，AI 还能根据字段类型与历史样本自动构造测试数据，覆盖边界值与异常值。整体呈现出强模板化生成能力，极大提升测试脚本产出效率。

3、日志分析与报告撰写：自动归因，智能反馈

在回归阶段，AI 可对海量日志进行聚类分析，归类常见异常（如空指针、SQL 错误），提高排障效率。它还能基于状态码/响应结构，识别异常模式并提供初步归因建议（如 token 失效、参数缺失）。测试完成后，AI 可自动生成 Markdown 或 HTML 报告，包含通过率、新失败用例、与上轮对比及建议关注模块。借助 NLP 能力，AI 成为反馈环节中的高效“总结者”。

AI 不擅长的四个关键领域



1、复杂业务建模：AI 不懂“流程里的弯弯绕”

复杂流程的关键，往往不在接口，而在背后的“隐性规则”。比如退款流程，涉及物流核实、财务方向、会员等级、支付通道等多系统协同。AI 或许能测接口通不通，却无法抽象出整个流程的依赖链与判断逻辑。更别提退货路径、发票、返点等边角场景。AI 无法建模这些业务“潜规则”，更无法像测试专家一样识别隐藏在流程缝隙里的风险。

2、时序与联动异常：AI 看不见“时间里的错”

系统问题不全是状态码和报错，更隐蔽的常是“时序错”——任务延迟、消息丢失、状态未同步等。AI 的判断大多是“快照式”的：调一次接口、看一次返回，它无法理解异步逻辑、处理多日延迟或判断“昨天通，今天卡”的根因。跨系统联动中的“时间相关性”仍然是 AI 测试的盲区。

3、风险判断与业务权衡：Bug 要不要修，不是 AI 能决定的

测试不只是判断对错，更要评估“是否值得修”。一个提示语 Bug，也许放着无伤大雅；而一个微小逻辑错误，可能影响大客户核心路径。AI 缺乏业务背景、用户画像和场景感知能力，无法判断 Bug 的优先级与影响面。真正的判断，需要项目经验和对上下文的深度理解。

4、非功能性问题：AI 没有“感觉神经”

不是所有问题都有明确断言。界面模糊、交互别扭、文档对不上接口，这些非功能性瑕疵，AI 看得见页面但感觉不到用户的“卡顿感”与“违和感”。真正的资深测试工程师，往往一眼就能发现“这个功能上线会被投诉”。这靠的是直觉、经验、和对用户的共情，而不是逻辑和语法。

五、实践案例：ChatGPT 自动生成测试代码

我们以一个常见的登录接口为例，Prompt 如下：

请写一个 Python 接口测试代码，验证用户名为空和密码为空场景。

ChatGPT 的代码输出如下：

```
import requests
import pytest
# 假设接口地址
LOGIN_URL = "http://example.com/api/login"
# 通用请求头
HEADERS = {
  "Content-Type": "application/json"
}
@pytest.mark.parametrize("username,password,expected_message", [  ("", "123456", "用户名不能为空"),
("test_user", "", "密码不能为空"),
])
def test_login_empty_fields(username, password, expected_message):    payload = {
```

```
"username": username,
"password": password
}
 response = requests.post(LOGIN_URL, json=payload, headers=HEADERS)  assert response.status_code == 400, "期望状态码应为 400"
 json_data = response.json()
assert json_data.get("message") == expected_message, f"提示信息应为：{expected_message
```

完整输出如图：

以下是一个使用 Python 编写的接口测试示例代码，验证用户名为空和密码为空的登录接口场景，使用了 requests 库和 pytest 测试框架：

前置准备

假设套录接口格式如下：

- 接口地址：H:http://esmple.con/apl/tngin
- 请求方式：POST
- 请求参数（JSON 格式）：

```
json
{
  "username":"xxx",
  "password":"xxx"
}
```

- 响应格式（示例）：

```
json
{
  "code":400
  "message":"用户名不能为空"
}
```

接口测试代码示例

```python
import requests
import pytest

# 假设接口地址
LOGIN_URL = "http://example.com/api/login"

# 通用请求头
HEADERS = {
    "Content-Type": "application/json"
}

@pytest.mark.parametrize("username,password,expected_message", [
    ("", "123456", "用户名不能为空"),
    ("test_user", "", "密码不能为空"),
])
def test_login_empty_fields(username, password, expected_message):
    payload = {
        "username": username,
        "password": password
    }

    response = requests.post(LOGIN_URL, json=payload, headers=HEADERS)
    assert response.status_code == 400, "期望状态码应为 400"

    json_data = response.json()
    assert json_data.get("message") == expected_message, f"提示信息应为: {expected_message}"
```

✅ 运行方法

1. 安装依赖：

```bash
pip install pytest requests
```

2. 保存代码为 `test_login.py`，在终端中运行：

```bash
pytest test_login.py -v
```

📌 小提示

- 如果接口返回码是 `200` 但业务失败，请将 `assert response.status_code == 400` 改为：

```python
assert response.json().get("code") == 400
```

- 你也可以加入更多边界场景，如：
  - 用户名和密码都为空
  - 用户名是空格
  - 长度超过最大限制

需要我继续加场景或封装为测试类风格也可以告诉我。

ChatGPT 回答优点

1. 结构清晰、易于维护

- 使用了 pytest.mark.parametrize 参数化测试，让多个输入场景复用一个测试函数。
- 清晰地分离了输入、预期结果和断言逻辑。

2. 语义明确

- 测试用例名称和断言语句中使用了中文提示信息，便于定位失败原因。

3. 符合实际项目需求

- 典型接口测试中“用户名为空”“密码为空”的校验是常见用例，示例具有实战参

考价值。

• 使用 requests 模拟真是 HTTP 请求，贴近实际后端接口调用方式。

4. 可扩展性好

• 可以轻松添加更多异常/边界用例，只需增加参数化项。

• 可以方便集成到 CI 流水线（例如 pytest + Jenkins）。

ChatGPT 回答缺点

1. 未封装为类或模块

• 在实际项目中，测试代码往往按功能模块或接口进行组织，这里是脚本式的单文件写法和简单测试，不适合大型项目维护。

2. 未包含前置清理/初始化逻辑

• 比如登录前可能需要清除某些缓存、初始化数据库状态，这里没有体现（但对“空用户名/密码”测试通常也不需要）。

3. 未对响应结构做健壮性判断

假设接口总是返回 JSON 并包含 message 字段，若后端异常返回非 JSON，程序将报错，可优化如下：

```
try:
 json_data = response.json()
except ValueError:
 pytest.fail("响应不是合法的 JSON 格式")
```

4. 硬编码了 URL 和响应结构

•接口地址 LOGIN_URL 是硬编码的，应支持通过配置文件读取以适配不同环境（如测试、预发布、正式）。

•响应字段 message 也是假设约定好的字段，建议定义统一响应协议模型进行断言更稳健。

种种说明：AI 生成的是模板级测试，而非语义驱动的高质量测试设计。

六、哪些测试岗位更容易被 AI 替代？

功能测试执行岗：替代风险极高（★★★★）

这个岗位是 AI 最容易接管的部分。为什么？因为功能测试的很多工作本质上就是：根据需求文档或用例步骤，反复去执行操作，验证系统是否按照预期响应。这类工作具备三个显著特征：

• 高度重复性：无论是页面点选、接口调用、表单提交，流程基本固定，每次执行过程区别不大。

• 可标准化：测试用例和预期结果通常是提前定义好的，适合被自动化工具识别并判断通过/失败。

• 结果清晰可判定：返回值对不对、界面跳转对不对，是非逻辑明确，AI 可以训练模型准确判断。

例如，输入用户名密码登录成功跳转首页，或接口返回状态码 200，AI 完全可以根据模板、规则、历史记录判断是否符合要求，不需要太多“思考”。只要输入数据、运行环境和路径固定，AI 可以做到高效率、低出错、连续执行。未来这类岗位极可能被“低代码自动化平台+AI 回归调度”所取代。

自动化测试工程师：替代风险较高（★★★）

表面上看，自动化测试工程师做的是“技术活”，写脚本、搞框架，看起来复杂一些。

但别忘了——AI 最擅长的事情之一就是写代码，特别是有明确目标、固定格式的代码，比如测试用例、断言逻辑、数据构造等。

目前市面上的 Copilot、ChatGPT 等工具已经可以自动生成不少高质量的测试脚本，比如用 pytest 测试一个 API 接口、用 selenium 模拟登录操作、生成测试数据工厂等。而且写完之后还能解释逻辑、优化结构，有时比初级测试开发还细心。

当然，自动化测试并不只是“写脚本”这么简单，更重要的部分是测试框架的设计、公共组件的抽象、异常场景的恢复处理、数据清理策略等。这些内容需要对系统结构、业务逻辑有深刻理解，而目前 AI 尚不具备这类“架构抽象”能力，因此这个岗位的中高级职责仍然较安全。但低阶“写脚本、填参数”层面的工作，已经开始逐步被 AI 蚕食。

性能测试工程师：替代风险中等（★★）

性能测试属于相对工具化较强的一类工作，比如用 JMeter、LoadRunner、Locust 等工具进行并发压测、TPS 统计、响应耗时分析、瓶颈定位等。执行层面的很多流程，AI 可以部分参与，甚至在 CI 流程中实现自动化加载与结果输出。

但问题在于，性能问题并不总是“结果数字异常”那么简单，更大的难点在于：

- 测试黄精的设计是否科学？
- 压测模型是否模拟了真实用户行为？
- 数据增长趋势是否模拟合理？
- 瓶颈到底是数据库？缓存？线程数？网络？还是第三方调用？

这些问题涉及到系统结构、部署架构、硬件配置、业务高峰特征等复合因素。AI 目前缺乏系统级的调优能力，也没有经验做出判断：“这个 TPS 掉了，是不是因为缓存击穿？”、“接口耗时高，是不是某个表没加索引？”

因此，性能测试的执行环节可被 AI 辅助，但诊断、调优、建模、评估报告撰写这类分析性、策略性的工作，依然需要有经验的工程师完成。

安全测试工程师：替代风险较低（★）

安全测试不是简单地调用几个接口去验证权限是否正确。真正的安全测试工程师，干的是逻辑绕过、身份冒用、权限提升、攻击路径构造、代码注入与逆向分析等工作。

这类岗位最大的特点是：思维非常非标准化，没有固定路径可走。你要尝试构造最奇怪的请求串，设计绕过验证的序列，甚至做一些“越界”的事情。

AI 目前在已知的 OWASP Top10、安全漏洞规则里，可以做些静态扫描、简单模拟攻击（如 SQL 注入测试），但要它去找出业务逻辑漏洞，比如：

- 怎么骗系统多发一张优惠券？
- 能不能通过换包请求跳过支付？
- 多线程下有没有并发抢占漏洞？

这种 “不按套路出牌”的测试方法，AI 几乎没有能力掌握。而高端安全测试甚至涉及对 APP 逆向、WebShell 植入、加密协议破解等技术，更是 AI 无法胜任的高度专业领域。

测试架构师 / 测试经理：替代风险极低（★）

这是当前最安全的测试岗位，短期内几乎没有被 AI 取代的可能。

原因很简单：这个角色不是执行测试，也不是写脚本，而是在“测试工作该怎么做、做成什么样、如何协作落地”层面进行整体设计与组织推动。

比如：

- 测试流程应该怎么建？在哪些阶段做自动化？哪些要人工补充？
- 平台需要支持哪些测试类型？要不要做测试数据隔离？测试环境该怎么搭？
- 团队怎么搭建？用哪些工具？测试规范怎么落地？和研发怎么协作？

这些工作牵扯到人、组织、资源、技术、文化、目标之间的关系，是高度综合的工程活动。AI 可能能给你一些参考建议，但它无法处理多角色之间的博弈、优先级决策和人心管理。

而作为测试经理，不仅要懂技术，还要负责进度协调、风险控制、质量评估和组织

赋能，这些内容本质上是“人际沟通 + 项目感知 + 主观判断”的混合体，AI 根本无法真正进入。

总之：

AI 能替代“怎么做”的岗位，但替代不了“做什么、为什么做、做得好不好”的人。

未来的趋势很明确：执行型测试岗位会逐步被 AI+自动化工具压缩，但懂得思考、设计、决策的测试人，反而会越来越重要。

AI 不会一夜替代你，但可能会淘汰 “只会执行、不懂思考”的你，因为真正被替代的，从来不是岗位，而是 “拒绝进化的思维方式”。

七、测试工程师的新角色：AI 协同与测试设计者

未来的测试工程师，不再是“写测试用例的人”，而是“设计质量策略的人”，其能力模型将发生质变：

未来测试人的新核心能力维度

能力维度	原测试岗要求	未来测试岗要求
测试技术	编写用例、自动化脚本	编排测试流、设计Prompt、定义验证流程
系统建模能力	理解需求、制定测试点	具备建模能力、业务抽象能力
AI协同能力	使用工具辅助生成测试代码	驱动AI完成测试链路、自定义反馈逻辑
沟通与影响力	提交Bug、配合研发改进	协调多角色达成质量目标、参与产品决策过程

未来测试工程师的能力，不再只是“写测试用例”那么简单

随着 AI 介入测试流程，测试工程师这个角色正在快速转型。过去我们强调“执行力”，未来我们需要“控制力”“设计力”“协同力”。下面我们从四个核心能力维度，来看一位合格的未来测试工程师，究竟需要具备怎样的新技能。

1、测试技术：从写脚本到设计测试链路

以前的测试工程师，技术能力主要体现在编写测试用例、写自动化脚本。你能写个 Selenium 脚本跑 UI 测试，能用 pytest 写接口验证，就算不错的测试开发。

但在 AI 时代，这种“点对点”的技术操作将变得基础甚至廉价。未来的测试工程师，技术核心是：能不能搭建一整条“验证流程”的自动化链路？

你要懂得：

• 如何将多个微服务的测试打通形成一个“端到端场景”；

• 如何将 CI/CD、测试框架、AI 提示指令（Prompt）整合到一个“可重复验证体系”中；

• 如何设计参数化、断言逻辑、失败重跑、mock 策略等“工程化机制”；

你不是一个“写 case 的人”，你是一个“测试系统的设计师”。

2、系统建模能力：从点状用例到业务抽象理解

传统测试员的建模能力，是“看一条需求，想出几个测试点”。比如，用户注册功能要测哪些字段输入是否合法，返回码对不对，页面跳转是否正确。

但未来的要求是：你能不能把一个业务流程抽象成状态机、流程图、或事件驱动模型？

比如，一个订单从创建到支付到履约，涉及多个系统协同（库存、财务、营销、配送），状态可能有 8~10 种变化路径。测试工程师要能：

• 建立出完整的流程图、路径图；

• 找出其中最复杂的边界条件；

• 识别哪些路径最容易被忽视但风险最高（例如异步回调失败后的补偿机制）；

简单说，你不再是拿“测试用例模板”去对照，而是自己能建出“业务规则模型”再来设计验证路径。

AI 可以帮你生成代码，但建模抽象这种需要业务逻辑理解+系统结构感知的事，依然是测试人员的核心竞争力。

3、AI 协同能力：从“用工具”到“调 AI 做事”

过去你会用一点 Postman、JMeter、Pytest，就算“技术测试”。但未来，这远远不够了。

未来的测试工程师必须具备一种新能力：你不仅是用 AI 工具，而是要“调教 AI、驱动 AI”干活。这意味着你要懂得：

•如何用 Prompt 设计思路，指挥 AI 批量生成测试代码、Mock 数据或边界条件用例；

• 如何把 AI 输出接入自己的测试框架，形成一个完整链路；

• 如何校准 AI 输出，评估它的准确性、稳定性，以及如何设计 fallback 机制（AI 写错了怎么办？有没有 review 策略？）；

举个例子，你不再是“让 ChatGPT 写个用例”，而是“设计一个 Prompt 模板，驱动 AI 高效协同完成测试链路”，能自动提取文档中的所有 API，生成接口测试脚本，并在失败时附上运行截图和上下文信息”。

AI 协同的能力，决定了你是在“用 AI”，还是被“被 AI 替代”的人。

4、沟通与影响力：从提 Bug 到推动质量结果

很多测试人以前的角色是“问题发现者”——写用例、提 Bug、配合修复。技术不错，但影响力不足，决策链条很靠后。

未来，这种状态必须改变。

真正有价值的测试工程师，要从“测试参与者”变成“质量设计者”，从执行链底端走到产品策略讨论的中前段。

你要能做到：

- 面对复杂版本节奏时，帮助产品评估“哪些模块风险高、需要留更多测试时间”；
- 和研发一起讨论接口协议的可测性与易维护性；
- 和 DevOps 一起优化 CI 流程，让自动化跑得更快、更稳定；
- 在质量红线问题发生时，有能力组织质量回溯、提出流程改进建议；

说到底，测试不是找问题的人，而是交付信心的人。这就要求你不仅能测，还能讲清楚“我们为什么这样测、测了有什么用、不测会出什么事”。

八、结语：AI 测试，不是终点，而是重生

图灵测试的提出，是人类第一次认真思考：“如果机器足够聪明，它是否可以像人一样去‘思考’？”

而我们作为测试工程师，肩负的正是另一重意义上的图灵测试 —— 在软件系统日趋复杂的今天，我们不断地用人类的思维和经验，去挑战一个系统是否“像用户一样在使用中正常运行”。我们用质疑、用假设、用破坏性的想象力，为整个软件交付过程建立一层最后的“理性防火墙”。

今天，AI 进入测试领域，它确实让我们变得更高效、更自动化、更智能——我们再也不用花几天时间一条条造数据、手写冗长脚本、点击重复流程。AI 可以帮我们构建测试数据、生成用例、扫描风险、做回归追踪，它已经是我们测试流程里的“第二大脑”。

但请注意：AI 只能成为“大脑的外挂”，不能成为“大脑的替身”。

真正的“第一大脑”，始终是那个能：

- 看出需求里的模糊点；
- 理解系统架构里的风险链条；
- 用一句话把 Bug 解释得让开发、产品都无法反驳；
- 在最后一分钟提出“这块别上线”的人。

那是你。是你这个拥有经验、判断力、技术感知和用户共情能力的测试工程师。

未来最有价值的测试人，不是“会执行”的人，而是懂 AI 但不被 AI 牵着走、会用工具但不会被工具替代、能抽象业务又能落地实现的人。

他们不是在担心 AI 会不会取代自己，而是在思考：“我怎么把 AI 用得比别人更好？”

所以，读到这里，你可以问问自己一个非常真实的问题：

你，是还在害怕被 AI 替代

还是已经开始，用 AI 去重塑你自己的测试思维和能力边界？

这是一个终点，但更是一次重生。

因为测试这个职业，从来都不是“写多少个用例”，而是能不能站在系统和用户之间，守住那一道我们叫做“信任”的质量底线。

拓展学习

[4] AI 工具实战演练，企业项目落地指导，领【AI 测试试听】

咨询：微信 atstudy-js 备注：AI 测试

几则 Oracle 数据库查询性能分析和优化案例

◆作者：fhh192

【案例 1】

某系统后台采用 Oracle 数据库作为数据存储，该系统的某业务需要对表 Table 执行查询操作，该查询操作执行时间不稳定，有时出现执行时间较长的问题。

业务执行的 SQL 语句较为简单，只对表 Table 这一单表执行了查询操作，如下所示：

```
SELECT COLA,COLF
FROM TABLE
WHERE COLA=$1 AND COLB=$2 AND COLC=$3 AND COLD=$4
ORDER BY COLE;
```

对该 SQL 语句的执行计划进行分析，其执行计划如下所示：

```
-------------------------------------------------------------------------------------
| Id  | Operation                    |Name      |Rows |Bytes |Cost(%CPU)|Time     |
-------------------------------------------------------------------------------------
|   0| SELCT STATEMENT              |          |     |      |  9   (100)| 00:00:01 |
|   1|  SORT ORDER BY               |          |    2|   980|  8    (20)| 00:00:01 |
|  *2|   TABLE ACCESS BY INDEX ID   | TABLE    |    2|   980|  7     (1)| 00:00:01 |
|  *3|    INDEX RANGE SCAN          | IDX_COLC |    2|   835|  5     (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
 2- Filter(("COLA"=:1 AND "COLB"=:2 AND "COLD"=:4))
 3-Access("COLC"=:3)
```

通过执行计划可以发现，该 SQL 语句执行时，首先使用了 COLC 列的单列索引 IDX_COLC 执行了过滤操作，之后进行了回表并按 COLA、COLB、COLD 列进行了二次过滤。

对于这种通过索引再回表的查询，其执行效率与索引过滤需要访问的数据量有关。对索引列 COLC 的数据分布情况进行分析，如下所示：

```
SELECT COUNT(*),COLC
FROM TABLE
GROUP BY COLC
ORDER BY 1 DESC;

COUNT(*) | COLC
--------------------
 5687854 | 02
  507891 | 05
  267787 | 01
  135879 | 03
   67672 | 07
   40068 | 04
   15965 | 06
```

通过以上的查询结果可知，索引列 COLC 的数据存在严重的分布不均衡的问题，总量最多的 02 值与总量最少的 06 值之间相差达到 356 倍，表 TABLE 共有 6723116 条记录，如果查询条件为 COLC=02，通过该索引访问后需要执行 5687854 次的回表再过滤操作，回表的记录数已经达到全表记录的 84%，理论上，此时更适合直接采用全表扫描的方式。

由此可知，索引列 COLC 的数据分布问题是导致该查询执行时间不稳定的直接原因。

对于该查询的优化，需要重新选择索引列。一个列的数据分布严重不均，从另一个角度反映了该列存在大量的重复数据，存在大量重复数据的列的选择性是较差的，选择性较差的列不适合建立索引。

对该 SQL 语句的其他过滤列的分布情况进行分析，并按选择性从高到低进行排序，结果为 COLD>COLA>COLB，因此，可以按选择性从高到低为这三个列创建一个复合索引 IDX_DAB(COLD,COLA,COLB)。

设置复合索引后，每次执行 SQL 语句时，通过该复合索引可以过滤掉表 TABLE 中的大量记录，需要读取的索引块较之前大量减少，只需执行少量的回表后再按 COLC 列的过滤操作，执行耗时稳定在 1 秒左右。

【案例 2】

某系统后台采用 Oracle 数据库作为数据存储，该系统的某业务需要对表 Table 执行查询操作，该查询操作执行时间不稳定，有时出现执行时间长达数十分钟的问题。

业务执行的 SQL 语句对表 Table 执行了自关联的 NOT EXISTS 子查询操作，如下所示：

```
SELECT    COUNT(*)
FROM TABLE T1
WHERE INSERTDATE>=sysdate AND INSERTDATE<sysdate+1
        AND UID='21'
        AND NOT EXISTS(
                        SELECT PID
                        FROM TABLE T2
                        WHERE INSERTDATE<sysdate-1
                                AND T2.UID='21'
                                AND T1.PID=T2.PID
                      );
```

对该 SQL 语句的执行计划进行分析，其执行计划如下所示：

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                    |Name          |Rows |Bytes |Cost(%CPU)|Time     |
-------------------------------------------------------------------------------------------------
|   0| SELCT STATEMENT               |              |    1 |   52|  45   (10)| 00:00:07 |
|   1|   SORT GROUP BY               |              |    1 |   52|  45   (10)| 00:00:06 |
|  *2|     FILTER                    |              |      |     |  43   (1) | 00:00:05 |
|  *3|       HASH JOIN ANTI          |              |  128|  6616|  40   (0) | 00:00:05 |
|  *4|         INDEX RANGE SCAN      |IDX_UID_IDA|  323|  12781|  35   (0) | 00:00:01 |
|  *5|         INDEX RANGE SCAN      |IDX_UID_IDA|  981K|  13M|  20   (0) | 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
 2- Filter(SYSDATE@!<SYSDATE@!+1)
 3-Access("T1"."PID"="T2"."PID")
 4-Access("UID"="21" AND "ISERTDATE">=SYSDATE@! AND  "ISERTDATE"<SYSDATE@!+1)
 5-Access("T2"."UID"="21" AND "ISERTDATE"<SYSDATE@!-1)
   Filter("T2"."PID" IS NOT NULL)
```

通过执行计划可以发现，该 SQL 语句执行时，优化器（CBO）已经将子查询展开为 HASH 反连接操作，所以不存在子查询未展开导致的性能问题。

分别单独执行该 SQL 语句中的主查询部分语句和子查询部分语句，其中主查询返回 2087 条记录，子查询返回 2017894 条记录，执行计划中，主查寻结果集与子查询结果集执行了 HASH 反连接操作，HASH 连接适用于较大表或结果集的关联，因此，优化器选择的 HASH 反连接操作是正确的。

继续分析对表 Table 的访问操作，对表 Table 执行过滤时，优化器采用了 INDEX RANGE SCAN 索引范围扫描的方式，且没有回表访问。

表 Table 共有 250 万左右的记录，查询访问的 IDX_UID_IDA 索引是一个复合索引(uid,insertdate,pid)，其大小超过 2GB，子查询通过 INDEX RANGE SCAN（索引范围扫描）返回了 2017894 条记录，已经达到全表记录的 80%，需要访问大量的索引数据块。当 IDX_UID_IDA 索引数据块从 ORACLE 数据库的 BUFFER CACHE 缓冲区中置换出去后，执行该查询时就需要从磁盘读取索引数据块，INDEX RANGE SCAN（索引范围扫描）是单块读，其效率较差，特别是当所读取的索引数据块较多时，索引数据块的读取消耗了大量的时间，导致查询的执行时间较长。当 IDX_UID_IDA 索引数据块存在于 BUFFER CACHE 缓冲区中时，执行 INDEX RANGE SCAN（索引范围扫描）无需再读取磁盘，直接从缓冲区获取，此时执行相同的查询，其执行时间就是较快的。这就是该 SQL 语句执行时间不稳定的原因。

对于该查询的优化，因为其对表 TABLE 访问后需要返回该表 80%的数据且没有回表操作，其完全可以通过访问索引即可获取到所需的数据，所以，可以使用 INDEX FAST FULL SCAN（索引快速全扫）替代优化器采用的 INDEX RANGE SCAN（索引范围扫描），INDEX FAST FULL SCAN（索引快速全扫）采用多块读的方式，适合于大量数据块的读取，在 IDX_UID_IDA 索引数据块需要从磁盘中读取的情况下，其读取效率高于 INDEX RANGE SCAN（索引范围扫描）。

使用/*+ index_ffs(T2) */提示器强制 exists 子查询对表 TABLE 访问时采用 INDEX FAST FULL SCAN（索引快速全扫），即：... AND NOT EXISTS(SELECT /*+ index_ffs(T2) */ PID ... ...)。

优化后，该 SQL 语句的执行时间控制在 5 秒左右，即使发生了从磁盘读取索引数据块的操作，整体执行时间也降低至 60 秒以内。

【案例 3】

某系统后台采用 Oracle 数据库作为数据存储，该系统的某业务需要对表 PRPLXXXX，即：别名表 A 和表 PRPLREXXXX，即：别名表 B 执行关联查询操作，某日，该查询操作执行时间突然变长，超过 60 分钟。

业务执行的 SQL 语句如下所示：

```
SELECT 'NOCAR' AS PLATFORMTYPE,
       'CLAIM' AS SUBPLATFORMTYPE,
        CLAIMNO,
       A.CASENUM AS CASENO,
       A.RISKCODE,
       A.CLASSCODE,
       A. ... ...
FROM  PRPLXXXX A
LEFT JOIN PRPLREXXXX B ON A.REGISTNO = B.REGISTNO
WHERE SUBSTR(A.RISKCODE, 1, 2) NOT IN ('05', '31', '32', '25', '40');
```

对该 SQL 语句的执行计划进行分析，其执行计划如下所示：

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |     1 |   981 | 34937   (1)| 00:07:00 |
|   1 |  NESTED LOOPS OUTER          |                 |     1 |   981 | 34937   (1)| 00:07:00 |
|*  2 |   TABLE ACCESS FULL          | PRP[redacted]   |     1 |   951 | 34935   (1)| 00:07:00 |
|   3 |   TABLE ACCESS BY INDEX ROWID| PRPLR[redacted] |     1 |    30 |     2   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN         | PK_LREG[redacted]|     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(SUBSTR("A"."RISKCODE",1,2)<>'05' AND SUBSTR("A"."RISKCODE",1,2)<>'31' AND
              SUBSTR("A"."RISKCODE",1,2)<>'32' AND SUBSTR("A"."RISKCODE",1,2)<>'25' AND
              SUBSTR("A"."RISKCODE",1,2)<>'40')
   4 - access("A"."REGISTNO"="B"."REGISTNO"(+))
```

通过执行计划可知，表 A 与表 B 采用了嵌套循环关联，表 B 作为被驱动表，且被驱动表在关联列上使用了索引扫描，该执行计划看似是“正确”的。

进一步分析参与关联的表 A 与表 B 的结果集的大小，发现关联的两表结果集的记录数基本在 95 万左右，其中作为驱动表的表 A 使用 WHERE 条件过滤后的记录数达到 94 万条左右，这将导致被驱动表表 B 被访问 94 万次，显然，这种关联方式的性能是低下的，优化器选择了错误的关联方式。

应将表 A 与表 B 的方式调整为 HSAH 关联，使用提示器强制使用 HASH 关联，即：/*+ use_hash(A,B) */。

调整后使用 HASH 关联，该 SQL 语句的执行时间降低至 4 分左右。

进一步分析优化器选择错误关联方式的原因，表 A 中数据随系统的运行不断增长，但未及时收集统计信息，优化器一直使用的是数据较少时的旧的统计信息，导致优化器认为表 A 的数据较少，因此仍然使用之前的嵌套循环关联方式。

对表 A 重新收集统计信息，去掉语句中的强制使用 HASH 关联的提示器，重新执行查询，优化器在执行该查询语句时可以正确地选择采用 HASH 关联的方式对表 A 和表 B 进行连接。

【案例 4】

某系统后台采用 Oracle 数据库作为数据存储，该系统的某业务需要同时对表 t1、t2、t3 执行查询操作，该查询的执行时间达到 1 分钟左右。

业务执行的 SQL 语句如下所示：

```
select count(1)
from  t1
where t1.object_type = 'TABLE'
      or exists (
                 select 1
                 from t2
                 where t1.object_id = t2.object_id
                )
      or exists (
                 select 1
                 from t3
                 where t1.object_id = t3.object_id
                );
```

该 SQL 语句中存在两个 exists 关联子查询，且它们之间的关系为 or，对该 SQL 语句的执行计划进行分析，其执行计划如下所示：

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name               | Rows  | Bytes | Cost (%CPU)| T
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                    |     1 |    15 |     183   (1)|
|   1 |  SORT AGGREGATE      |                    |     1 |    15 |              |
|*  2 |   FILTER             |                    |       |       |              |
|   3 |    TABLE ACCESS FULL | T1                 |  108K |  1585K|     183   (1)|
|*  4 |    INDEX RANGE SCAN  | IND_T2_OBJECT_ID   |     1 |     5 |       1   (0)|
|*  5 |    INDEX RANGE SCAN  | IND_T3_OBJECT_ID   |     1 |     5 |       1   (0)|
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
2 - filter("T1"."OBJECT_TYPE"='TABLE' OR  EXISTS (SELECT 0 FROM "T2" "T2"
    WHERE "T2"."OBJECT_ID"=:B1) OR  EXISTS (SELECT 0 FROM "T3" "T3" WHERE
    "T3"."OBJECT_ID"=:B2))
4 - access("T2"."OBJECT_ID"=:B1)
5 - access("T3"."OBJECT_ID"=:B1)
```

该查询的执行统计信息如下所示：

```
统计信息
----------------------------------------------------------
0  recursive calls
0  db block gets
184940  consistent gets
0  physical reads
0  redo size
349  bytes sent via SQL*Net to client
472  bytes received via SQL*Net from client
2  SQL*Net roundtrips to/from client
0  sorts (memory)
0  sorts (disk)
```

通过执行计划和统计信息可知，该查询执行时，因存在对两个 exists 相关子查询结果集的 or 操作，使得优化器未能消除 exists 子查询，执行计划中产生了 FILTER 操作，该查询的访问的主表 t1 返回的记录较多，使得 FILTER 操作的性能较低，查询执行时的逻辑读（consistent get）达到 1849940，这是导致该查询执行效率不高的主要原因。

对该 SQL 语句的优化，需要通过等价改写的方式，使用 union/union all 替代对 exists 子查询结果的 or 操作。

首先将对表 t2 和 t3 的 exists 关联子查询更改为表 t2 与表 t1，表 t3 与表 t1 的连接查询，将这两个查询结果使用 union 连接（去掉两个结果集中的重复数据），该部分语句如下所示：

```
select t1.*
from t1, t2
where t1.object_id = t2.object_id and t1.object_type <> 'TABLE'
union
select t1.*
from t1, t3
where t1.object_id = t3.object_id and t1.object_type <> 'TABLE'
```

之后，对表 t1 执行查询，并将查询结果集与上述查询的结果集执行 union all 操作（保留两个结果集中的重复数据），最终修改后的查询语句如下所示：

```
select count(1)
from
      (
       select t1.*
       from t1
       where t1.object_type = 'TABLE'
       union all
       (
        select t1.*
        from t1, t2
        where t1.object_id = t2.object_id and t1.object_type <> 'TABLE'
        union
        select t1.*
        from t1, t3
        where t1.object_id = t3.object_id and t1.object_type <> 'TABLE'
       )
    );
```

改写后的查询，其执行时间降低到 2 秒左右，以下展示了改写后的查询语句的执行计划：

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |     1 |       |   571   (1)|
|   1 |  SORT AGGREGATE              |                   |     1 |       |            |
|   2 |   VIEW                       |                   | 29274 |       |   571   (1)|
|   3 |    UNION-ALL                 |                   |       |       |            |
|*  4 |     TABLE ACCESS FULL        | T1                |  2303 | 87514 |   183   (1)|
|   5 |     SORT UNIQUE              |                   | 29274 |  1218K|   571  (69)|
|   6 |     UNION-ALL                |                   |       |       |            |
|*  7 |      HASH JOIN               |                   | 25971 |  1090K|   201   (1)|
|   8 |       INDEX FAST FULL SCAN   | IND_T2_OBJECT_ID  | 26007 |   126K|    17   (0)|
|*  9 |       TABLE ACCESS FULL      | T1                |   105K|  3931K|   183   (1)|
|* 10 |      HASH JOIN               |                   |  1000 | 43000 |   187   (2)|
|  11 |       INDEX FAST FULL SCAN   | IND_T3_OBJECT_ID  |  1000 |  5000 |     3   (0)|
|* 12 |       TABLE ACCESS FULL      | T1                |   105K|  3931K|   183   (1)|
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
4 - filter("T1"."OBJECT_TYPE"='TABLE')
7 - access("T1"."OBJECT_ID"="T2"."OBJECT_ID")
9 - filter("T1"."OBJECT_TYPE"<>'TABLE')
10 - access("T1"."OBJECT_ID"="T3"."OBJECT_ID")
12 - filter("T1"."OBJECT_TYPE"<>'TABLE')
```

通过改写后的执行计划可知，查询执行时，已经消除了 FILTER 操作，统计信息如下所示：

```
统计信息
----------------------------------------------------------
1  recursive calls
0  db block gets
2027  consistent gets
0  physical reads
0  redo size
349  bytes sent via SQL*Net to client
472  bytes received via SQL*Net from client
2  SQL*Net roundtrips to/from client
1  sorts (memory)
0  sorts (disk)
```

改写后的查询，执行时的逻辑读（consistent get）由之前的 1849940 降至 2027。

【案例 5】

某系统后台采用 Oracle 数据库作为数据存储，该系统的某业务需要对表 XXX_REMINDER_ORGCUST 和表 XXX MCMODEI_REMINDER（B）执行关联，并进行分页操作，最后排序后获取前 50 条记录，该查询执行较慢，平均执行用时 16 秒左右。

业务执行的 SQL 语句如下所示：

```
select * from(
        select tmp.*,rownum row_id
        from(
            select a.mid,a.cno,a.cname,b.stu
            from [illegible]_MODEL_REMINDER_ORGCUST a inner join [illegible] MODEI_REMINDER b
                on a.model_id=b.model_id and b.status='01'
            where a.procod='24' and a.assign_orgno in ('992400000000') and
                (b.role_id_id='BASEPF0015' or b.role_id_id is null) and b.model_id='xncs1000000008101'
            order by a.reach_date desc
            ) tmp
        where rownum<=50
    )
where row_id>=1;
```

表 XXX_MODEL_REMINDER_ORGCUST 记录较多，达到 3 千万左右。

经对执行计划进行分析，执行计划其中存在 SORT ORDER BY ROWNUM 排序操作，该 SQL 执行表表关联后，返回的结果集较大，达到 300 万条记录，对 300 万条进行排序，造成了较大的开销，且因为 PGA 空间有限，排序占用了临时表空间，同时造成了一定的 I/O 开销（出现 direct path readtemp 和 direct path write temp 事件）。

对该 SQL 查询调优的首要就是消除排序，因排序列为表 XXX_MODEL_REMINDER_ORGCUST，所以，为该表创建一个联合索引，按“(等值过滤列,关联列,排序列)”的顺序依次创建，需要注意的是，该 SQL 查询按表 XXX_MODEL_REMINDER_ORGCUST 的 reach_date 列降序排序，reach_date 列需要创建降序索引，创建索引的语句如下：

```
create index idx1 on [illegible]_MODEL_REMINDER_ORGCUST(procod,assign_orgno,model_id,reach_date desc);
```

创建索引后，该 SQL 语句的执行计划如下所示:

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Times
0	SELECT STATEMENT		50	131K	150	00:00:01
*1	VIEW		50	131K	150	00:00:01
*2	COUNT STOPKEY					
3	VIEW		50	131K	150	00:00:01
4	NESTED LOOP		50	10600	149	00:00:01
*5	NESTED LOOP		52	10600	149	00:00:01
*6	TABLE ACCESS BY INDEX ROWID	[illegible]_MODEL_REMINDER_ORGCUST	222	40781	50	00:00:01
*7	INDEX RANGE SCAN	IDX1	51		5	00:00:01
*8	INDEX RANGE SCAN	R_MODEL_INDEX	1		3	00:00:01
*9	TABLE ACCESS BY INDEX ROWID	[illegible]MODEL_REMINDER	1	26	1	00:00:01

通过执行计划可以发现，排序操作已被消除，优化器选择了嵌套循环关联，并以排序列所属的表 XXX_MODEL_REMINDER_ORGCUST 作为嵌套循环关联的驱动表。

实际执行该 SQL 语句，查询用时降至 0.6 秒左右。

继续对该查询进行优化，通过进一步分析表 XXX_MODEL_REMINDER_ORGCUST 的数据分布，发现该表的记录按 procod 列的数据分布，整个表中 procod 列的值约有 30 个，每个值对应的记录数在 100 万-300 万条左右，查询执行时对表 XXX_MODEL_REMINDER_ORGCUST 按 procod 的值进行了过滤。因此，可以将表 XXX_MODEL_REMINDER_ORGCUST 按 procod 的值进行分区，创建非 HASH/LIST 分区。此外，每次查询时 procod 只指定一个值，不存在跨分区查询，所以，为 procod 列创建本地索引即可。

经过第二次调优后，该 SQL 语句的执行计划如下所示：

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Times	Pstart	Pstop
0	SELECT STATEMENT		50	131K	120(00:00:01)			
*1	VIEW		50	131K	120(00:00:01)			
*2	COUNT STOPKEY							
3	VIEW		50	131K	120(00:00:01)			
4	NESTED LOOP		50	10600	120(00:00:01)			
5	NESTED LOOP		52	10985	6 (00:00:01)			
6	PARTITION LIST SINGLE		292K	50M	6 (00:00:01)	22	22
7	TABLE ACCESS BY LOCAL INDEX ROWID	[illegible] MODEL_REMINDER_ORGCUST	292K	50M	5(00:00:01)	22	22
*8	INDEX RANGE SCAN	INDEX ORGCUST3	51		5(00:00:01)	22	22
*9	INDEX RANGE SCAN	R_MODEL_INDEX	1		3(00:00:01)		
*10	TABLE ACCESS BY INDEX ROWID	[illegible] MODEL_REMINDER	1	26	1(00:00:01)		

Predicate Information (identified by operation id):

1 - filter("RNUM">=1)
2 - filter(ROWNUM<=10)
8 - access("A"."ASSIGN_ORGNO"='992400000000'AND "A"."MODEL ID"='xncs100000000810')
9 - access("B"."MODEL_ID"='xnC8100000000810')
10 - filter(("B"."ROLE_ID_ID" IS NULL OR "B"."ROLE_ID_ID"='BASEPFOO15') AND "B"."STATUS"='00')

通过以上的执行计划可以发现，对表 XXX_MODEL_REMINDER_ORGCUST 的访问只扫描了一个分区，该 SQL 语句的执行时间进一步降低至 10 毫秒左右。

实例讲解测试 Web 应用防火墙

◆ 作者：小猪

前言

Web 应用防火墙也称 WAF（Web Application Firewall），是部署在 Web 服务器前端、基于 HTTP/HTTPS 流量分析的网络安全防护设备，通过规则匹配、行为分析和机器学习技术识别 SQL 注入、XSS 攻击等恶意流量并进行拦截。（这段内容摘自百度百科）

在生产环境上线的 Web 系统，都会部署 Web 应用防火墙或其他类似的软硬件产品，来避免或降低各种恶意流量的威胁。既然是一款如此重要的产品，我们就需要知道如何评价它的好坏，毕竟市面上的同类产品这么多，各家都王婆卖瓜，所以作为用户总要敲一敲，或者试吃一个，才能判断这个瓜到底甜不甜。

当然，本文仅从软件测试中安全测试的角度来测试 Web 应用防火墙，也就是针对漏洞防御这块进行测试。

首先，要测试 Web 应用防火墙，我们需要一款好的模糊测试（FUZZ）工具。何谓好的模糊测试工具，笔者认为只要符合代码开源、分类管理、灵活加载三原则的 FUZZ 工具，就是一款好的模糊测试工具。在本文中会向读者展示笔者根据这三原则而设计的模糊测试工具。

代码开源其实很好理解，尤其是做安全或渗透测试，最好知道所使用的工具都做了什么，测试逻辑是什么，才好判断测试效果，也能灵活调整，适应不同情况。

分类管理什么，灵活加载什么？当然是漏洞测试脚本了，这些漏洞测试脚本可以称之为“弹”，“子弹”的“弹”，可以“连发”，也可以“单发”。

模糊测试（FUZZ）工具的“连发”功能针对 Web 应用防火墙的某一种策略库，而“单发”功能针对 Web 应用防火墙对新漏洞的响应功能，毕竟现在的 Web 应用防火墙都对接

了 AI 功能，AI 可以实时监控各大漏洞发布平台最新的漏洞发布情况并作出适当的处理，但是这个功能还是要验证一下。

正文

一、模糊测试工具

模糊测试工具的设计原则是代码开源、分类管理和灵活加载。

首先，启动开源免费的安全测试工具，在工具的菜单栏选择“渗透测试”->”扫描平台”，如图 1-1 所示。

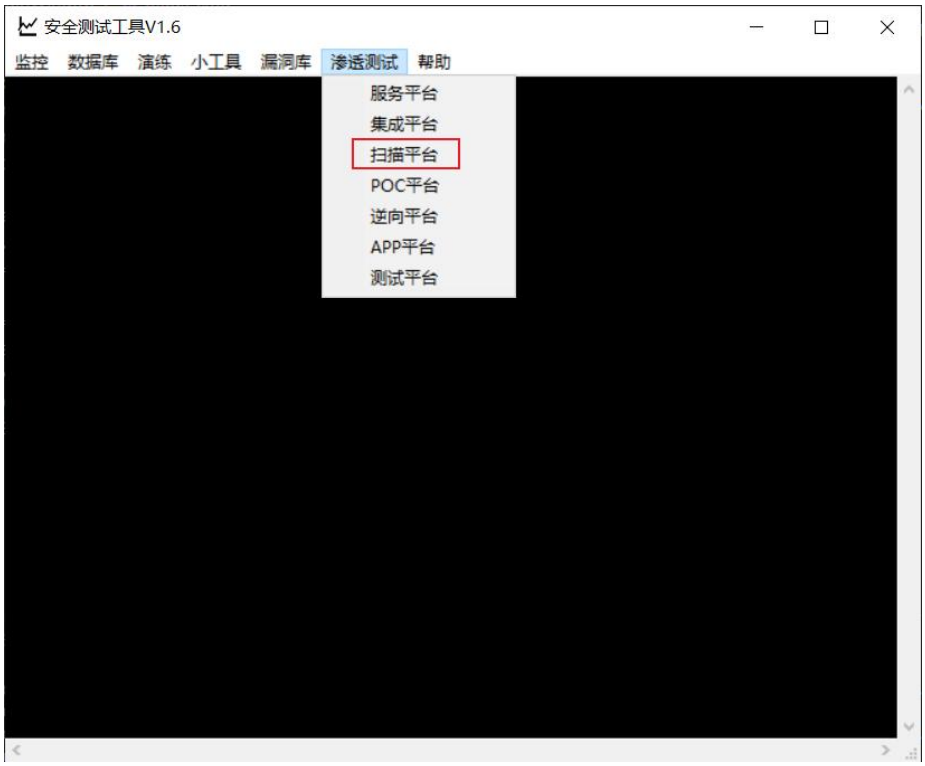


图 1-1

(该工具的使用教程链接在结尾，有兴趣的读者可以参考。)

打开扫描平台窗口，如图 1-2 所示。

图 1-2

在扫描平台窗口中输入命令“help”，可以查看漏洞分类，如图 1-3 所示。

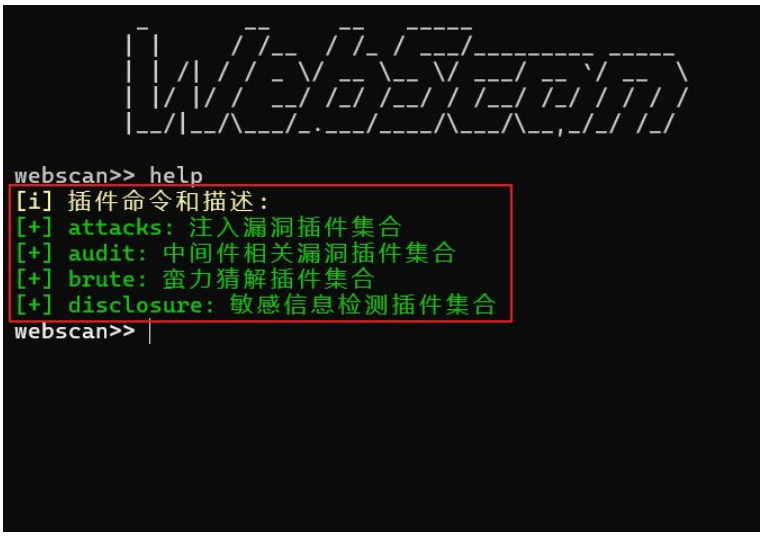


图 1-3

图 1-3 所示的漏洞分类可以根据实际情况进行更改，这个实际情况指的是可以对应待测防火墙的策略库，更改的地址在安全测试工具源代码根目录下“scan\plugins”，如图 1-4 所示。

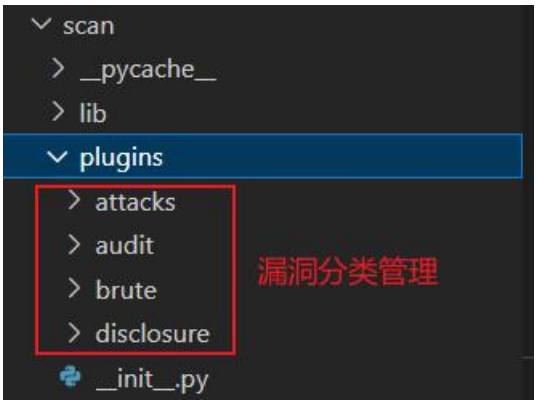


图 1-4

在扫描平台窗口中输入命令“info + 漏洞分类模块名称”，可以查看具体的测试漏洞脚本，如图 1-5 所示。

图 1-5

在具体的漏洞目录下可以增加、更改或减少漏洞测试脚本，以适应实际的测试需求，谓之灵活加载。

漏洞测试脚本的具体地址为安全测试工具源代码根目录“scan\plugins\漏洞目录名称”，有兴趣的读者可以参考，如图 1-6 所示。

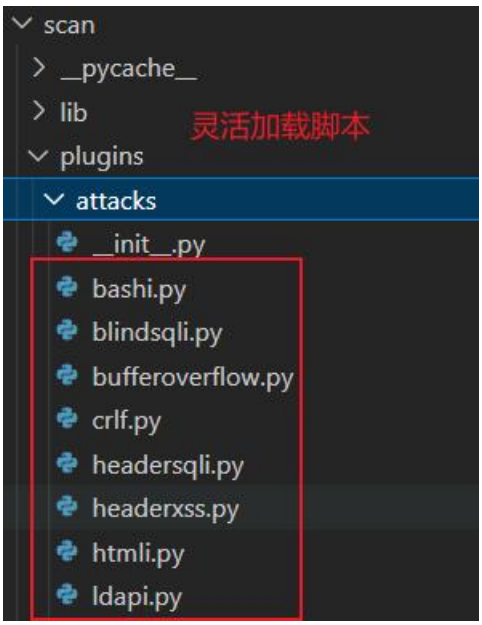


图 1-6

在扫描平台窗口中输入命令“exec + 漏洞分类模块名称”，可以执行该漏洞分类模块下所有的漏洞测试脚本，也就是所谓的“连发”测试，如图 1-7 所示。

图 1-7

在扫描平台窗口中输入命令“exec + 漏洞分类模块名称.漏洞脚本名称”，可以执行单一漏洞测试脚本，也就是所谓的“单发”测试，如图 1-8 所示。

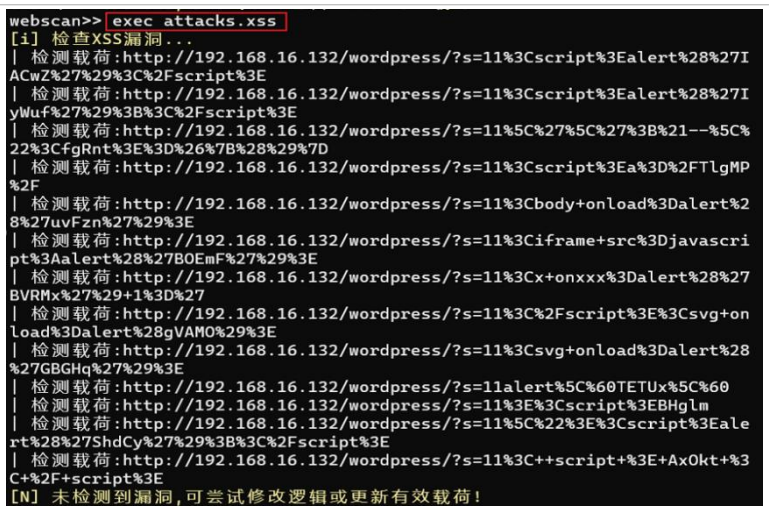


图 1-8

好了，到这里我们有了一款分类管理，灵活加载并且代码开源的模糊测试（FUZZ）工具，并且该工具即可以“连发”测试 WAF 指定策略库，也可以“单发”测试具体的新发布漏洞，那么我们离完成测试就只差一个简单的测试方法了。

二、针对 WAF 策略库进行测试

有了称手的工具，那测试方法就很简单了，总结起来就五步。

第一步，FUZZ 工具加载漏洞模块。

第二步，WAF 设置拦截恶意流量的策略库，这里读者要根据 WAF 的实际情况进行设置。

第三步，启动 FUZZ 工具执行漏洞测试脚本。

第四步，检查 WAF 防火墙的拦截日志，并与 FUZZ 工具所执行的漏洞测试脚本进行对比。

第五步，根据对比结果，评估拦截效果。

接下来，我们简单模拟一下执行流程。

首先，我们启动设计好的模糊测试（FUZZ）工具，并进行基础设置，如图 1-9 所示。

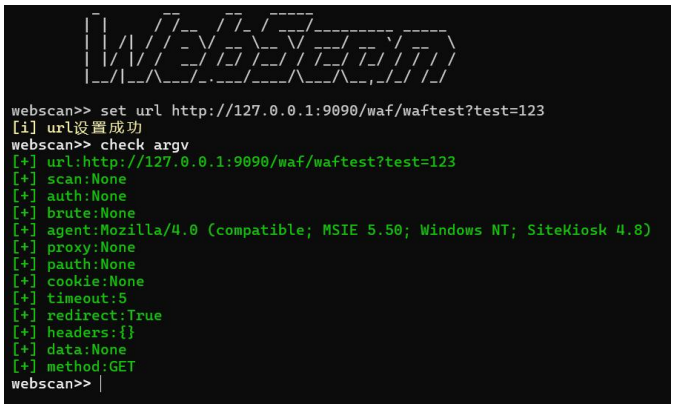


图 1-9

接下来，启动模拟 WAF 功能，因为是模拟 WAF 功能，所以策略库就当作是默认策略库，如图 1-10 和图 1-11。

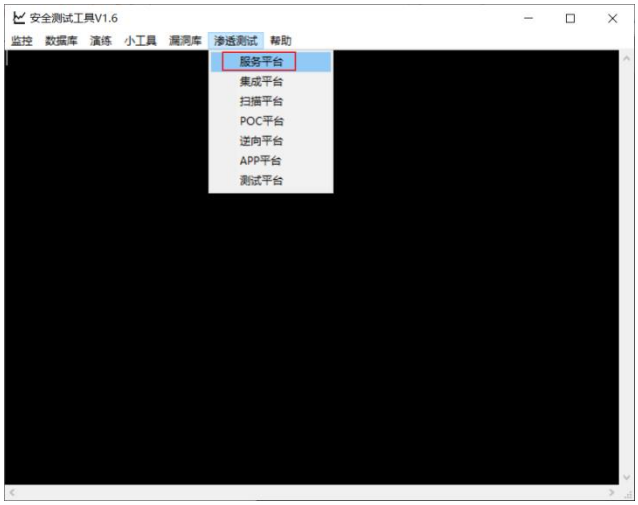


图 1-10

图 1-11

接下来，在模糊测试工具启动执行漏洞测试，如图 1-12 所示。

图 1-12

接下来，检查 WAF 防护墙拦截日志，如图 1-13 所示。

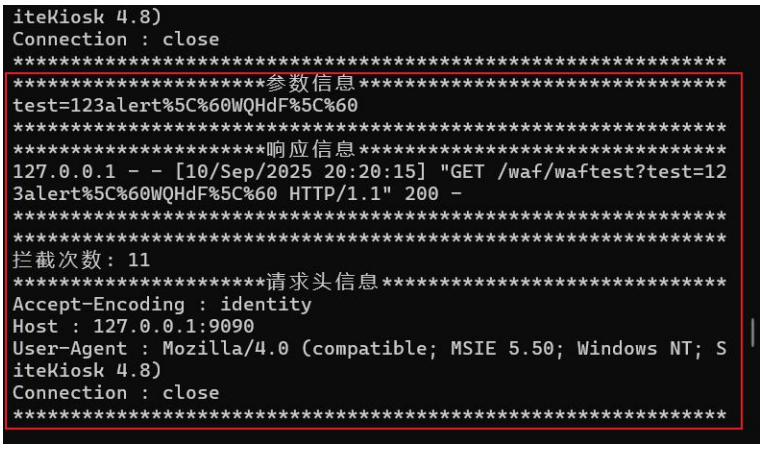


图 1-13

最后对比结果，评估拦截效果，如图 1-14 所示。

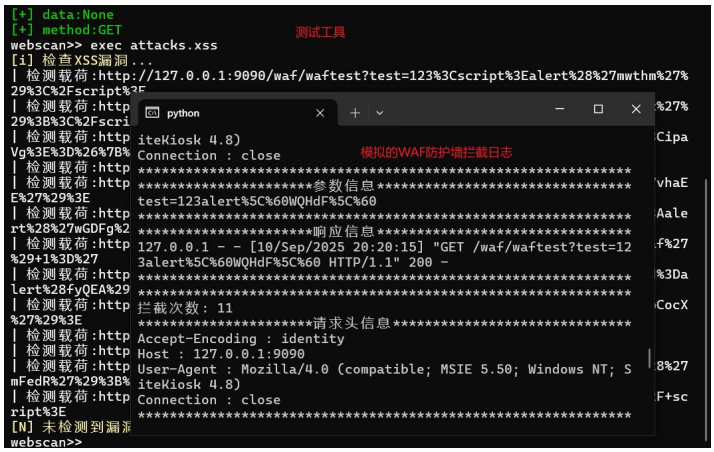


图 1-14

三、针对新发布漏洞进行测试

新发布漏洞，不是 0-day 漏洞，应该是 1-day，2-day 漏洞。

针对新发布漏洞，我们需要验证 Web 应用防护墙的响应时间是否及时，毕竟现在已经接入了 AI，如果响应时间太慢，就有点说不过去了。

具体延迟多少天算慢，这个就看 Web 应用防火墙的使用说明了，如果它宣传的是即时响应，那 3 天以上就算太慢了。

对于新发布漏洞，我们需要关注比较有名的漏洞发布平台，结合实际系统的架构情况，找到漏洞的详细说明，并根据提供的漏洞验证脚本，进行修改后（大部分情况都需要修改），加入到模糊测试工具中，这样也方便回归测试。

以我们的测试工具进行举例，现在要编写新发布漏洞的测试脚本，简单步骤如下：

第一、配置漏洞管理模块路径，新增的漏洞分类统一按下面的格式配置，如图 1-15 所示。

```
#插件配置
##集成检测注入漏洞模块,相关的注入性漏洞放在这里
P_ATTACKS_PATH = os.path.join(PLUGIN_PATH,"attacks")
##集成检测默认配置漏洞，中间件配置相关的漏洞放在这里,中间件[Apache,weblogic,Tomcat,nginx,iis等]
P_AUDIT_PATH = os.path.join(PLUGIN_PATH,"audit")
##集成检测蛮力猜解模块,例如猜解后台管理地址、敏感的备份文件等
P_BRUTE_PATH = os.path.join(PLUGIN_PATH,"brute")
##集成检测页面或响应内容中的敏感信息等
P_DISCOSURE_PATH = os.path.join(PLUGIN_PATH,"disclosure")
```

图 1-15

第二步、配置漏洞管理模块描述信息，如图 1-16 所示。

```
#插件模块信息
##插件描述
plugs_info ={
    'attacks':"注入漏洞插件集合",
    'audit':'中间件相关漏洞插件集合',
    'brute':'蛮力猜解插件集合',
    'disclosure':"敏感信息检测插件集合"
}
plugs = {
    '1':'attacks',
    '2':'audit',
    '3':'brute',
    '4':'disclosure',
    '5':'fullscan'
}
```

图 1-16

第三步、配置漏洞验证脚本描述信息，如图 1-17 所示。

```
#注入漏洞检测配置 attacks
##检测模块描述配置
attacks_info = {
    'bashi':"检测bashi命令注入,破壳漏洞(ShellShock)",
    'blindsqli':"检测SQL盲注",
    'bufferoverflow':'检测缓冲区溢出检测',
    'crlf':"检测crlf注入 即回车换行注入",
    'headersqli':"HTTP的头信息header,检测是否可能存在sql注入漏洞",
    'headerxss':"检测Headers头信息中XSS漏洞",
    'htmli':"检测html代码注入",
    'ldapi':"检测LDAP[轻量级目录访问协议]注入",
    'lfi':"检测本地文件包含漏洞",
    'oscommand':"检测操作系统命令注入漏洞",
    'phpi':"检测PHP代码注入漏洞",
    'sqli':"检测SQL注入漏洞",
    'ssi':"检测服务端包含注入漏洞",
    'xpathi':"检测xpath[xml路径语言]注入漏洞",
    'xss':"检测XSS跨站点脚本漏洞",
    'xxe':"检测XXE XML外部实体注入漏洞",
    'xst':"#检测XST跨站跟踪漏洞"
}
```

图 1-17

第四步、配置漏洞测试脚本的路径信息，如图 1-18 所示。

```
##检测模块路径配置
attacks = {
    #检测bash命令注入,破壳漏洞(ShellShock)
    'bashi': P_ATTACKS_PATH + os.sep + 'bashi.py',
    #检测SQL盲注
    'blindsqli':P_ATTACKS_PATH + os.sep + 'blindsqli.py',
    #检测缓冲区溢出检测
    'bufferoverflow':P_ATTACKS_PATH + os.sep + 'bufferoverflow.py',
    #检测crlf注入 即回车换行注入
    'crlf':P_ATTACKS_PATH + os.sep + 'crlf.py',
    #HTTP的头信息header,检测是否可能存在sql注入漏洞
    'headersqli':P_ATTACKS_PATH + os.sep + 'headersqli.py',
    #检测Headers头信息中XSS漏洞
    'headerxss':P_ATTACKS_PATH + os.sep + 'headerxss.py',
    #检测html代码注入
    'htmli':P_ATTACKS_PATH + os.sep + 'htmli.py',
    #检测LDAP[轻量级目录访问协议]注入
    'ldapi':P_ATTACKS_PATH + os.sep + 'ldapi.py',
    #检测本地文件包含漏洞
    'lfi':P_ATTACKS_PATH + os.sep + 'lfi.py',
    #检测操作系统命令注入漏洞
    'oscommand':P_ATTACKS_PATH + os.sep + 'oscommand.py',
    #检测PHP代码注入漏洞
    'phpi':P_ATTACKS_PATH + os.sep + 'phpi.py',
    #检测SQL注入漏洞
```

图 1-18

第五步、加载漏洞测试脚本的方法编写，如图 1-19 所示。

```
#加载注入漏洞插件
def attacks_plugins():
    _plugins = []
    for root,dirs,files in os.walk(P_ATTACKS_PATH):
        files = filter(lambda x: not x.startswith("__")
        and x.endswith(".py"),files)
        _plugins.extend(map(lambda x: os.path.join(root,x),files))
    return _plugins
def spec_attacks_plugins(key:str):
    _plugin = []
    if key in attacks.keys():
        _plugin.append(attacks[key])
    return _plugin
#加载审计漏洞插件
def audit_plugins():
    _plugins = []
    for root,dirs,files in os.walk(P_AUDIT_PATH):
        files = filter(lambda x: not x.startswith("__")
        and x.endswith(".py"),files)
        _plugins.extend(map(lambda x: os.path.join(root,x),files))
    return _plugins
def spec_audit_plugins(key:str):
    _plugin = []
    if key in audit.keys():
        _plugin.append(audit[key])
    return _plugin
```

图 1-19

第六步、编写漏洞测试脚本，如图 1-20 所示。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
from re import search,I
from lib.utils.printer import *
from lib.request.request import *
from lib.utils.payload import bash

class bashi(Request):
    """
    bash命令注入
    破壳漏洞(ShellShock)
    """
    get = "GET"
    def __init__(self,kwargs,url,data):
        Request.__init__(self,kwargs)
        self.url = url
        self.data = data
        self.result = {}
        self.result['bashi'] = None
```

图 1-20

最后就是执行脚本测试，可参考前面的执行流程。

结尾

本篇内容中所使用的安全测试工具，在 51testing 测试圈中有详细的安装和使用教程。

有兴趣的读者可以访问：http://quan.51testing.com/pcQuan/lecture/117。

自动化测试 POM 常见陷阱：四大 Anti-Pattern 解析

◆作者：TestCraft

在自动化测试框架中，我们经常使用 POM（Page Object Model）模式对被测试页面进行封装抽象。然而，在实际开发过程中，我们常常看到由于对 POM 理解不深或实践方式不当，导致测试代码结构逐渐偏离其初衷，使得自动化测试框架反而变得臃肿、难以维护。

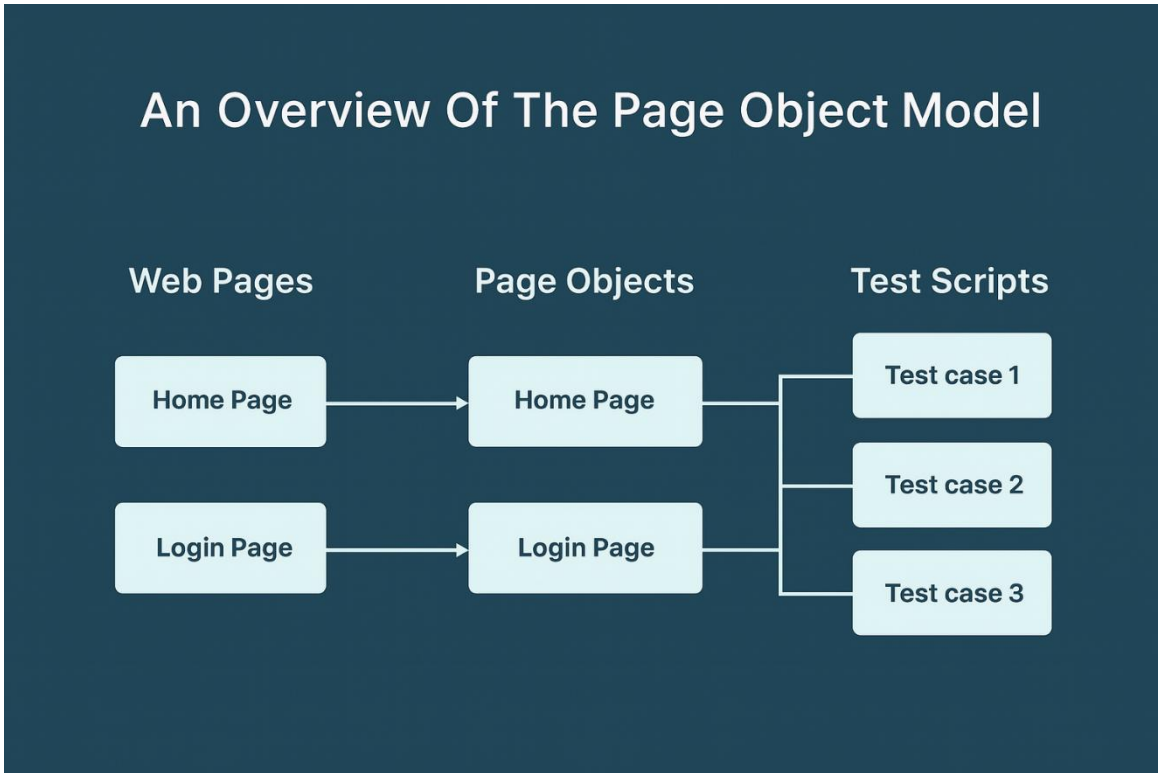
本文将梳理 POM 的常见反模式（Anti-Pattern），帮助你在项目中更好地实践和落地 POM 模式。

What is the Page Object Model (POM)?

POM 本质上是一种设计模式，用于创建网页元素的对象库。其核心目的是建立机制，为框架提供更健壮、稳定和灵活的设计。

在页面对象类的开发时，我们为不同的页面创建一个对应的页面类，每个类中封装该页面上的元素定位器以及常用操作方法。

这种方式实现了页面元素与测试逻辑的解耦，多个测试脚本可以复用同一页面类中的元素与方法，从而大大减少了代码重复，提升了整体的可维护性与扩展性。



为什么使用 POM

1、可维护性

在 Web 应用的开发过程中，如果测试代码直接去操作页面上的对象，一旦 UI 发生变动测试用例就可能全部失效，就会造成散弹式的修改，维护成本很高。

假设有多个不同的登录测试用例（如不同用户名、密码错误、验证码验证等）。一旦页面中的某个元素发生改变，或者登录页面多了一个验证框时、所有涉及登录的测试用例都需要修改。

POM 对被测试页面进行抽象和封装，将页面元素和操作方法交互逻辑集中在一个类中。当 UI 发生变化时，只需要修改对应的 Page 类，从而提升测试用例代码的可维护性。

2、代码复用性

POM 通过封装页面上的操作可以使不同的测试脚本可以复用相同的交互逻辑，减少代码重复。

例如，多个测试用例可能都需要登录功能，如果每个测试都手动输入用户名和密码，代码会大量冗余。但使用 POM，可以封装 login() 方法，在不同的测试中直接调用，提

高代码复用性。

3、让测试逻辑与 UI 结构解耦

使用了 POM 设计后，就间接的为测试框架引入了分层设计的概念，上层的测试逻辑和底层的页面解耦。这种解耦方式使得测试层的代码可以专注于测试本身，而不依赖具体的操作或元素。

POM 的设计原则

1.单一职责原则（SRP）： 每个页面对象类应只负责一个页面或页面的一小部分功能。这样可以保持代码结构清晰、有条理，便于维护，同时明确不同功能区域的职责边界。

2.抽象性（Abstraction）： 页面对象应屏蔽页面交互的细节，例如元素定位和具体操作方法。通过提供更直观、可读性强的接口，抽象能够降低测试代码的脆弱性，并提升其可维护性。

3.封装性（Encapsulation）： 页面对象应封装页面的状态与行为，使开发者可以轻松理解当前页面的状态以及可执行的操作。这有助于保持关注点分离，并提高测试代码的可读性。

4.可复用性（Reusability）： 页面对象应被设计为可在多个测试中复用，从而减少代码重复，提高测试开发的效率。通过构建模块化、自包含的页面对象，可以像积木一样快速组合测试逻辑。

5.易读性（Easy to Understand）： 页面对象中的方法和变量命名应具备良好的可读性和表达力，让开发者无需阅读底层代码即可理解其作用。清晰的命名有助于提升测试代码的可读性与可维护性。

6.关注点分离（Separation of Concerns）： 测试脚本应专注于描述页面的高层行为，而页面对象则负责底层的页面操作细节。实现这种职责分离可以让测试代码更清晰、更易组织，同时构建出更具可扩展性的测试框架。

POM 中的常见的 Anti Pattern

1、在 Page Object 中添加断言或验证逻辑

最最常见的一个习惯，估计就是在 POM 中加入断言了，比如登录方法中直接写上判

断登录是否成功。因为这样似乎就能复用了，但是等一下！

在页面对象方法中包含断言模糊了关注点。页面对象的职责应仅限于封装页面上的元素操作，而断言属于测试验证逻辑，应该由测试用例负责。将断言写在页面类中，会导致职责混乱，降低测试用例的灵活性与可维护性。

✖不推荐的做法：在页面对象中断言

```
# page_objects.py
class LoginPage:
    def __init__(self, driver):
        self.driver = driver
        self.username_input = driver.find_element_by_id("username")
        self.password_input = driver.find_element_by_id("password")
        self.login_button = driver.find_element_by_id("login_button")
        self.welcome_message = driver.find_element_by_id("welcome_message")
    def login(self, username, password):
        self.username_input.send_keys(username)
        self.password_input.send_keys(password)
        self.login_button.click()
        assert "Welcome" in self.welcome_message.text, "Login failed"
```

☑推荐的做法：将断言放在测试用例中

```
# page_objects.py
class LoginPage:
    def __init__(self, driver):
        self.driver = driver
        self.username_input = driver.find_element_by_id("username")
        self.password_input = driver.find_element_by_id("password")
        self.login_button = driver.find_element_by_id("login_button")
    def login(self, username, password):
        self.username_input.send_keys(username)
        self.password_input.send_keys(password)
        self.login_button.click()
```

```
        def get_welcome_message(self):
            return self.driver.find_element_by_id("welcome_message").text
# test_login.py
def test_login_success():
    login_page = LoginPage(driver)
    login_page.login("user123", "password123")
    welcome_message = login_page.get_welcome_message()
    assert "Welcome" in welcome_message, "Login failed"
```

将断言逻辑混入页面对象 POM 中主要带来的问题：

1. 限制用例扩展，影响多场景测试

就拿登录举例，当需要测试多个登录场景（成功、密码错误、用户名为空等），但断言已经写死在 login() 方法中，导致你不得不复制代码或创建多个方法（如 login_with_valid_credentials()、login_with_invalid_password()），这反而增加了代码重复和维护成本。

2. 造成不必要的失败

假设你有一个测试流程是：创建用户 → 删除用户。测试的重点是验证删除功能是否正常，但如果在“创建用户”这一步就加了断言（验证欢迎信息等），一旦这个验证因为意外失败(比如延迟)，就会阻断后续步骤，让真正想测试的删除功能无从验证。

```
def test_create_and_delete_user():
    login_page.login("admin", "admin123")
    user_page.create_user("new_user")  # 创建时断言意外失败
    user_page.delete_user("new_user")  # 没有执行到这里
```

3. 引入复杂的验证逻辑或外部依赖

有时验证结果需要调用接口、查数据库、检查后端日志等，这些属于测试逻辑范畴，不应混入页面类中。如果你把这类逻辑塞入 Page 类，不但代码臃肿，还会造成页面类对测试环境有依赖，违背了测试框架的分层设计。

```
def login(self, username, password):
```

```
    # ✗ 页面对象中引入数据库验证
    db_user = query_user_from_db(username)
    assert db_user.active, "User not activated"
```

2、在测试函数中直接操作元素

在 Page Object 模式中，一个关键的最佳实践是：不要在测试脚本中直接使用 locator 来操作页面元素。相反，应当在页面对象类中封装好所有的元素定位与相关操作逻辑。

✗不推荐示例：没有封装元素或操作逻辑

```
# Playwright 示例
class LoginPage:
    def __init__(self, page):
        self.page = page
        self.email_field = page.locator("id=input-email")
        self.password_field = page.locator("id=input-password")
        self.login_button = page.locator('input:has-text("Login")')
        self.welcome_message = page.locator("#welcome")
# test_login.py
def test_login_success(page):
    login_page = LoginPage(page)
    login_page.email_field.fill("user@example.com")      # ✗ 操作在测试中进行
    login_page.password_field.fill("password123")        # ✗
    login_page.login_button.click()                      # ✗
    assert login_page.welcome_message.is_visible()       #
```

这种方式会破坏 POM 对象的封装性和抽象性，背离了设计初衷。我们会将所有的操作细节暴露或转移到了测试层中，会让测试用例不仅要描述“测试逻辑”，还必须处理“操作细节”，从而使测试代码变得冗长、难以阅读和维护。

想象一下，当一个测试用例需要执行多个步骤、操作多个页面元素时，这种将所有操作展开在测试层的方式会让测试脚本变得非常臃肿。

☑推荐做法：封装元素定位与操作逻辑

Selenium 示例：

```
# login_page.py
from selenium.webdriver.common.by import By
from selenium.webdriver.remote.webdriver import WebDriver
class LoginPage:
    def __init__(self, driver: WebDriver):
        self.driver = driver
        self._username_locator = (By.ID, "username")
        self._password_locator = (By.ID, "password")
        self._login_button_locator = (By.ID, "login")
    def login(self, username, password):
        self.driver.find_element(*self._username_locator).send_keys(username)
        self.driver.find_element(*self._password_locator).send_keys(password)
        self.driver.find_element(*self._login_button_locator).click()
    def is_logged_in(self) -> bool:
        return "Welcome" in self.driver.page_source
# test_login.py
def test_login_success(driver):
    login_page = LoginPage(driver)
    login_page.login("user@example.com", "password123")
    assert login_page.is_logged_in()
```

、Playwright 示例

```
# login_page.py
from playwright.sync_api import Page
class LoginPage:
    def __init__(self, page: Page):
        self.page = page
        self._email_field = page.locator("id=input-email")
        self._password_field = page.locator("id=input-password")
        self._login_button = page.locator('input:has-text("Login")')
        self._welcome_message = page.locator("#welcome")
```

```
        def enter_email(self, email: str):
            self._email_field.fill(email)
        def enter_password(self, password: str):
            self._password_field.fill(password)
        def click_login(self):
            self._login_button.click()
        def login(self, email: str, password: str):
            self.enter_email(email)
            self.enter_password(password)
            self.click_login()
        def is_logged_in(self) -> bool:
            return self._welcome_message.is_visible()
# test_login.py
def test_login_success(page):
    login_page = LoginPage(page)
    login_page.login("user@example.com", "password123")
    assert login_page.is_logged_in()
```

这边说明一下，在 Python 中虽然没有通过关键字强制私有属性的设计。可以通过在属性前加下划线（如 _email_field）表示该属性是私有内部使用的，调用者应该遵守约定不应该类外直接访问。

3、直接存放测试数据或环境变量

在自动化测试中，测试数据的管理与页面逻辑应保持解耦。良好的实践是将测试数据从代码中抽离出来，可以使用 YAML、JSON、CSV 等方式进行存放和管理再配合参数化使用，以提高灵活性与可维护性。

✕不推荐的做法：

```
# login_page.py
from playwright.sync_api import Page
class LoginPage:
```

```
    def __init__(self, page: Page):
        self.page = page
        self._username_field = page.locator('#username')
        self._password_field = page.locator('#password')
        self._login_button = page.locator('#login')
    def login(self):
        # ✕ 硬编码测试数据到页面类中
        self._username_field.fill('testuser')
        self._password_field.fill('password123')
        self._login_button.click()
```

为什么不推荐这样做：

- 页面对象类中包含了与页面无关的测试数据，违反了设计初衷
- 无法支持灵活测试（如错误账号、空密码、多账号测试等）
- 当数据变更时需要改动页面类，破坏封装性

☑推荐做法（数据解耦 + 参数传入）

```
from playwright.sync_api import Page
class LoginPage:
    def __init__(self, page: Page):
        self.page = page
        self._username_field = page.locator('#username')
        self._password_field = page.locator('#password')
        self._login_button = page.locator('#login')
    def login(self, username: str, password: str):
        self._username_field.fill(username)
        self._password_field.fill(password)
        self._login_button.click()
# test_login.py（测试用例）
# 测试用例中读取数据并传入
```

```
import json
import pytest
from login_page import LoginPage
@pytest.fixture
def test_data():
    with open('data/login_user.json', encoding='utf-8') as f:
        return json.load(f)
def test_login_success(page, test_data):
    login_page = LoginPage(page)
    login_page.login(test_data["username"], test_data["password"])
    assert page.locator("#welcome").is_visible()
```

通过数据与页面逻辑的解耦，不仅可以提升测试代码的清晰度，还能更方便地做数据驱动测试、只需要在 login_user.json 中更换或添加新的测试数据就可以测试多个不同的账号。

4、页面对象之间互相继承或依赖

如果你发现多个页面类之间存在大量的继承关系，那往往是一个信号：需要停下来对代码进行适当的重构。

虽然在实际项目中，我们常常会通过 BasePage 抽象封装一些通用的基础功能（例如导航、等待、截图等），这是一种合理且推荐的做法。但这并不意味着具体页面对象之间的继承也是一种好的实践。

✕不推荐的继承方式：页面继承页面

页面对象之间应当避免互相之间建立继承关系。例如，不建议让 SignInPage 继承 AuthenticationPage，因为这样会导致类结构混乱、职责不清晰，进而影响可维护性与可读性。每个页面对象类应保持独立，只负责封装自己页面的元素与行为。

```
# ✕ 页面类之间继承 - 不推荐
class AuthenticationPage:
    def __init__(self, page):
        self.page = page
```

```
            self.email = page.locator('#email')
            self.password = page.locator('#password')
        def fill_credentials(self, email, password):
            self.email.fill(email)
            self.password.fill(password)
    class SignInPage(AuthenticationPage):   # ? 页面继承页面
        def __init__(self, page):
            super().__init__(page)
            self.login_button = page.locator('#login')
        def login(self, email, password):
            self.fill_credentials(email, password)
            self.login_button.click()
    class SignUpPage(AuthenticationPage):   # ? 页面继承页面
        def __init__(self, page):
            super().__init__(page)
            self.signup_button = page.locator('#signup')
        def signup(self, email, password):
            self.fill_credentials(email, password)
            self.signup_button.click()
```

☑推荐的方式

当多个页面存在功能重叠时，更推荐将这些通用逻辑抽象为独立的模块或组件，并在需要的页面类中通过组合的方式引入使用。就像搭积木一样，将功能模块拼装在页面对象中，更贴近实际页面的结构与行为，也更有利于代码的灵活性和复用性。

示例：组件组合方式

```
# credential_form.py
class CredentialForm:
    def __init__(self, page):
        self.page = page
        self.email = page.locator('#email')
        self.password = page.locator('#password')
```

```
        def fill(self, email: str, password: str):
            self.email.fill(email)
            self.password.fill(password)
# sign_in_page.py
from credential_form import CredentialForm
class SignInPage:
    def __init__(self, page):
        self.page = page
        self.credential_form = CredentialForm(page)
        self.login_button = page.locator('#login')
    def login(self, email: str, password: str):
        self.credential_form.fill(email, password)
        self.login_button.click()

# sign_up_page.py
from credential_form import CredentialForm
class SignUpPage:
    def __init__(self, page):
        self.page = page
        self.credential_form = CredentialForm(page)
        self.signup_button = page.locator('#signup')
    def signup(self, email: str, password: str):
        self.credential_form.fill(email, password)
        self.signup_button.click()
```

总结

Page Object Model（POM）是一种设计模式，它通过将页面对象与测试逻辑解耦，显著提升了自动化测试的可维护性、可扩展性和代码复用性。然而，实践中一旦忽视其设计初衷，往往容易让测试框架变得臃肿、脆弱甚至难以扩展。

本文归纳了 POM 实践中最常见的 4 个反模式：

1.在页面对象中写断言 —— 违反关注点分离原则，降低可复用性；

2.在测试中直接使用元素定位器 —— 破坏封装，导致测试冗余难维护；

3.页面类中硬编码测试数据 —— 缺乏灵活性，无法支持数据驱动；

4.页面对象之间互相继承 —— 类结构混乱，职责不清，推荐使用组合。

拓展学习

[5] 领取【自动化测试全链路】资料包

咨询：微信 atstudy-js 备注：11

-------- END ---------