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

我开发「AI 测试用例生成工具」的“失败预演”

◆ 作者：M&T.

一、问题背景：我为啥非要做这个工具？

测试同学肯定懂那种“机械重复到麻木”的感受——每次版本迭代，“新增 XX”“查询 XX”“删除 XX”这类标准化用例能占总用例的 80%：界面必填项校验、取消按钮交互、提交成功的状态反馈，流程固定到闭着眼都能默写，但架不住功能模块多啊！

上次做“作业安全管理系统”迭代，光“新增”类用例我就写了 12 条：新增作业内容、新增作业地点、新增法规模板……每个用例都要写“前置条件：用户登录且进入 XX 页”“步骤 1：点击新增按钮”“期望结果：弹出新增弹窗”，熬到晚上 9 点才把全量用例出完，结果第二天 Leader review 时直接说：“逆向场景全漏了——新增空内容、重复提交、权限不足这些怎么没写？”

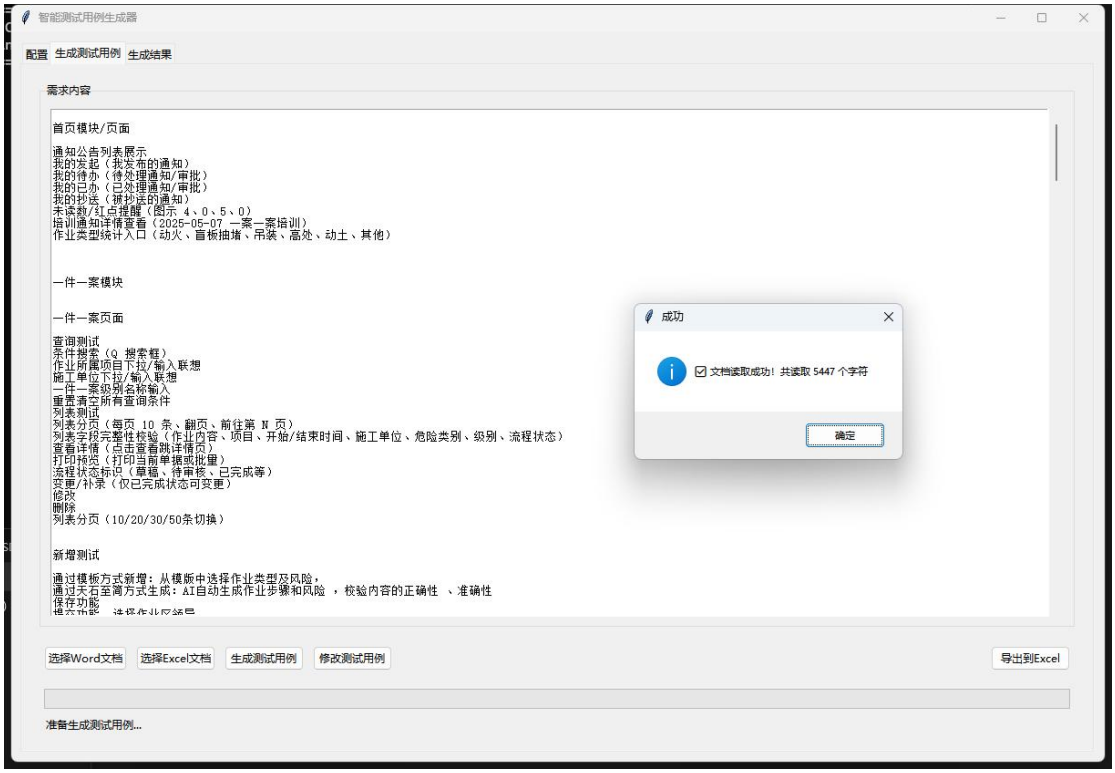
盯着屏幕上密密麻麻的“新增成功”，我突然琢磨：这类“可复制性高、场景固定”的用例，AI 肯定能接盘 80%——毕竟字段是固定的（模块名、用例编号、步骤、期望结果），场景是套路化的（正向+逆向+边界），只要把需求文档喂给 AI，让它自动生成、导出 Excel，我只需要补剩下 20%的复杂场景，岂不是能从重复劳动里解放出来？

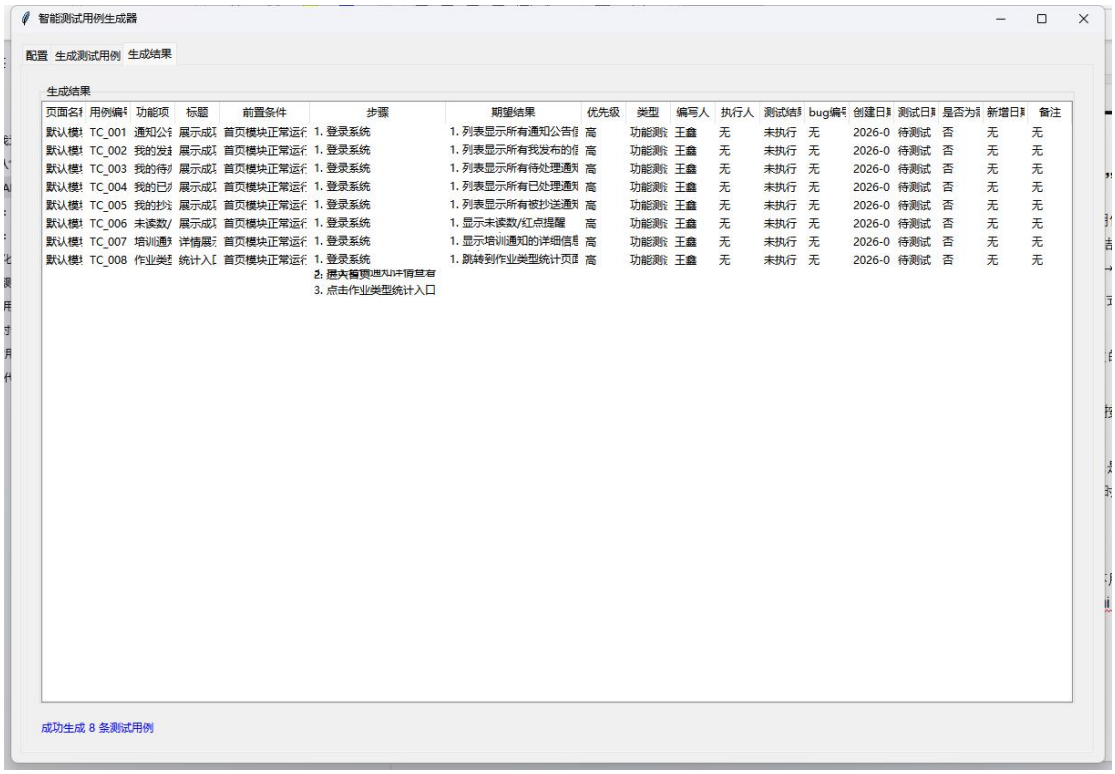
脑子一热，当天就开始搞工具：目标是做个带 GUI 的桌面端——选 Word 需求文档→AI 读取内容生成用例→支持手动修改→导出 Excel，当时甚至觉得“这波稳了，下周就能给组里同学用”。

二、过程难点：从“信心满满”到“半成品上线”

结果现实给了我结结实实的一巴掌：工具能跑，但生成的用例根本没法用，活脱脱

一个“半成品”。


智能测试用例生成器
配置
生成测试用例
生成结果
需求内容
首页模块/页面
通知公告列表展示
我的发起（我发布的通知）
我的待办（待处理通知/审批）
我的已办（已处理通知/审批）
我的抄送（被抄送的通知）
未读数/红点提醒（图示 4、0、5、0）
培训通知详情查看（2025-05-07 一案一案培训）
作业类型统计入口（动火、盲板抽堵、吊装、高处、动土、其他）
一件一案模块
一件一案页面
查询测试
条件搜索（Q 搜索框）
作业所属项目下拉/输入联想
施工单位下拉/输入联想
一件一案级别名称输入
重置清空所有查询条件
列表测试
列表分页（每页 10 条、翻页、前往第 N 页）
列表字段完整性校验（作业内容、项目、开始/结束时间、施工单位、危险类别、级别、流程状态）
查看详情（点击查看跳详情页）
打印预览（打印当前单据或批量）
流程状态标识（草稿、待审核、已完成等）
变更/补录（仅已完成状态可变更）
修改
删除
列表分页（10/20/30/50条切换）
新增测试
通过模板方式新增：从模版中选择作业类型及风险，
通过天右至简方式生成：AI自动生成作业步骤和风险 ，校验内容的正确性 、准确性
保存功能
成功
文档读取成功！共读取 5447 个字符
确定
选择Word文档
选择Excel文档
生成测试用例
修改测试用例
导出到Excel
准备生成测试用例...


智能测试用例生成器
配置
生成测试用例
生成结果
生成结果
成功生成 8 条测试用例

1. 工具逻辑：AI 写的代码“能跑，但没灵魂”

我一开始是这么跟 AI 提需求的：“帮我写个带图形界面的测试用例生成工具，字段要包含模块名、用例编号、功能项、标题、前置条件、步骤、期望结果；核心逻辑是选择 Word 文档→读取内容→调用 Kimi 大模型生成用例→支持修改→导出 Excel。”

AI 确实秒回了代码（就是我之前那版），甚至帮我加了“文件格式校验”“日志打印”这些细节，但调试时才发现：“没定标准”=“全是隐形 BUG”：

• 没定义“Word 需求文档的格式”：纯文字的需求被拆成零散的句子，带表格的需求直接把“模块名”和“功能点”混在一起；

• 没给“用例输出模板”：AI 一会儿把“步骤”写成“1. 点击对应按钮”，一会儿写成“点击新增”，甚至出现“操作对应功能”这种模糊到没法执行的描述；

这个环节调试生成的模板符合要求属于一个小难点 。

2. 大模型选择：“免费+长文本”的甜蜜陷阱

选 Kimi 是奔着两个“优势”去的：一是每月送 15 元免费额度，不用额外花钱；二是支持 128K 长文本，能完整读取我的需求文档——而且我之前用 Kimi 网页版生成过测试用例，质量还不错，本以为“API 调用和网页版是一个水平”。

```
# 定义支持的模型类型
MODEL_TYPES = {
    "Kimi": {
        "models": [
            "moonshot-v1-8k",
            "moonshot-v1-32k",
            "moonshot-v1-128k"
        ],
        "base_url": "https://api.moonshot.cn/v1/"
    }
}

# 定义默认模型
DEFAULT_MODEL_TYPE = "Kimi"
DEFAULT_MODEL_NAME = "moonshot-v1-8k"
```

结果工具里调用 Kimi 时，生成的用例直接“降智”：

• 调 temperature 参数、 max_tokens=1200：从 0.2（输出太简略，步骤只有 1 句话）调到 0.7（逻辑混乱，把“新增”写成“删除”），中间试了 0.5，结果是“步骤写了一半突然停了，只留了个‘1. 点击新增按钮’”；max_tokens 跳到了 4000+，这两个参数是大模型生成内容是否精细的关键

• 补提示词关键词：我加了“必须包含逆向场景，比如新增空内容提交、重复提交”，结果 AI 把“点击取消按钮”当成“逆向场景”写了进去；

• 加参考案例：我把之前手写的“新增作业内容”用例当示例喂进去，AI 倒是模仿了格式，但把“作业内容输入框”写成了“作业标题输入框”，张冠李戴得离谱。

以下是我的关键词

```
# 角色定位
你现在是【资深高级软件测试工程师】，拥有 10 年以上测试经验，精通等价类划分、边界值分析法、场景法、错误推测法，严格遵循 ISO/IEC/IEEE 29119 等软件测试行业规范。

# 任务目标
根据以下需求文档内容，**先识别出所有模块，然后识别每个模块下的所有页面，再分析每个页面下的所有功能点**，为**每个功能点**生成**高质量、可直接执行**的标准化测试用例。
```

```
**特别注意：** 文档采用以下层次结构：
- **模块级**：如"三、现场管理模块"
- **页面级**：如"（一）作业内容页面"
- **功能点级**：包括带序号的功能点（如"1. 新增作业内容功能"或"- 安全交底功能"）和不带序号的功能点（如下列示例）
```

```
**功能点识别示例**（请确保识别以下类型的功能点）：
- 通知公告列表展示
- 我的发起（我发布的通知）
- 我的待办（待处理通知/审批）
- 我的已办（已处理通知/审批）
- 我的抄送（被抄送的通知）
- 未读数/红点提醒（图示 4、0、5、0）
- 培训通知详情查看（2025-05-07 一案一案培训）
- 作业类型统计入口（动火、盲板抽堵、吊装、高处、动土、其他）
```

```
请确保**识别并处理所有层级**的结构，**不要遗漏任何页面或功能点**，无论是带序号的还是不带序号的功能点。
```

```
# 测试用例规范
1. **必填字段**（字段顺序严格不变，无内容填"无"）：
   {','.join(case_fields)}

2. **字段取值规则**：
   - 优先级：仅允许填写「高」「中」「低」
   - 类型：仅允许填写「功能测试」「接口测试」「边界值测试」「异常测试」「正向测试」「反向测试」「UI 测试」
   - 测试结果：仅允许填写「未执行」「通过」「失败」「阻塞」
   - 是否为需求变更新增用例：仅允许填写「是」「否」
   - 创建日期/新增日期：格式为「YYYY-MM-DD」
   - 测试日期：默认填「待测试」
   - bug 编号：默认填「无」
   - 编写人：固定为「xx」
   - 执行人：默认填「无」

3. **用例设计要求**（**必须严格遵守，生成的用例必须包含所有这些类型**）：
   - 【模块识别】：**必须先从文档中识别出所有模块**
   - 【页面识别】：**每个模块下必须识别出所有相关页面**
   - 【功能点识别】：**每个页面下必须识别出所有功能点**（如新增、查询、修改、删除、导出等）
   - 【功能点覆盖】：**每个功能点均需有对应的测试用例**
   - 【操作覆盖】：每个操作（如编辑 YAML 的查看、上传、下载、更新，新增、取消新增，修改、取消修改，删除、取消删除，审批功能的提交审批、审批通过、审批删除等）都应有单独的测试用例
   - 【正向测试】：正常流程的测试用例
   - 【反向逻辑】：**必须**包含所有操作的反向逻辑测试（如新建取消、删除取消、修改取消等）
   - 【字段校验】：**必须**包含表单字段的必填项校验、格式校验、长度校验等异常情况测试
   - 【边界值测试】：**必须**包含输入参数的边界值测试（如最小值、最大值、空值、特殊字符等）
   - 【异常测试】：**必须**包含各种异常情况的测试（如网络异常、权限不足、数据冲突等）
   - 【创建界面】：创建界面功能（一般为新增页面）需要添加用例
   - 【图形测试】：页面有图形的需追加一条"图形区域加载正常，无异常显示"的测试用例
   - 【列表测试】：列表字段需要查看校验、列表字段排序倒叙排列
   - 【编号规范】：用例编号从 TC_001 开始，按模块顺序递增
   - 【步骤要求】：步骤字段必须包含详细的操作过程，每个步骤前标有数字序号（1、2、3等），步骤之间用换行分隔，每个步骤仅描述一个操作，使用明确的动作词
   - 【期望结果】：期望结果必须与步骤一一对应，标有数字序号，结果之间用换行分隔，描述清晰明确
   - 【覆盖率要求】：必须完整覆盖等价类划分、边界值分析、场景法、错误推测法
```

```
   - 【质量要求】：用例无冗余、无重复、无遗漏，逻辑严谨，步骤清晰可执行，期望结果明确无歧义
   - 【智能发现】：除了上面的功能点外，发现可测试点也可以生成
4. **用例数量和覆盖要求**（**必须严格遵守**）：
   - 【场景覆盖】：**必须**覆盖正常场景、逆向场景（含非法输入、流程跳转异常、权限越界等至少 5 类逆向场景）、边界场景
   - 【功能点覆盖密度】：**每个功能点必须对应 3-5 条用例**（如新增功能至少 3 条、查询功能至少 3 条等），确保充分覆盖各种测试场景
   - 【无详细描述处理】：**当功能点没有详细描述时，至少编写 3 条测试用例**，包括正常场景、边界场景和异常场景的测试
   - 【逆向场景要求】：逆向场景**必须明确异常触发条件和预期报错信息**

**特别注意：生成的测试用例必须包含足够数量的反向逻辑、字段校验、边界值测试和异常测试用例，不能仅仅是正向功能的测试用例，必须符合真实测试人员的写作内容和测试思维。**
{module_prompt}

5. **输出格式要求**：
   - 每个测试用例独立展示，字段名与字段值对应清晰
   - 步骤和期望结果字段内的多个条目必须用换行分隔
   - 禁止添加任何额外的解释、开场白或结束语
   - 禁止使用任何特殊格式（如 Markdown、表格等）

# 测试用例模板示例
请严格按照以下格式生成测试用例：

页面名称：单位管理
用例编号：TC_001
功能项：新增功能
标题：新增成功
前置条件：单位字典管理模块正常运行
步骤：
1.登录系统
2.进入单位管理界面
3.点击新增
4.填写基本信息，点击确定
期望结果：
1.弹出新增页
2.成功创建单位，列表中显示新创建的单位
3.列表中数据更新（每个字段）
4.总记录数据更新
优先级：高
类型：功能测试
编写人：xx
```

```
执行人：无
测试结果：未执行
bug 编号：无
创建日期：{current_date}
测试日期：待测试
是否为需求变更新增用例：否
新增日期：无
备注：无

页面名称：单位管理
用例编号：TC_002
功能项：新增功能
标题：取消新增
前置条件：单位字典管理模块正常运行
步骤：
1.登录系统
2.进入单位管理界面
3.点击新增
4.填写基本信息并提交，点击取消
期望结果：
1.弹出新增页
2.取消后新增页关闭，未创建新单位
优先级：中
类型：功能测试
编写人：xx
执行人：无
测试结果：未执行
bug 编号：无
创建日期：{current_date}
测试日期：待测试
是否为需求变更新增用例：否
新增日期：无
备注：无

# 需求文档内容
{require_content}

# 固定参数
编写人：xx
执行人：无
创建日期：{current_date}
"""
```

```
    if modify_prompt:
        base_prompt += f"""
现在需要对你生成的测试用例进行修改，修改要求如下：{modify_prompt}
修改要求：
1、仅修改符合要求的用例，其他用例保持不变；
2、修改后依然要满足上述所有用例规范和字段要求；
3、修改后的用例编号保持不变，仅更新内容。
"""
```

3. 功能点识别：Word/Excel 都救不了的“睁眼瞎”

我的核心需求是“让 AI 从需求文档里自动抠出模块名、页面名、功能点”，结果 AI 是“瞎眼式提取”：

• 喂 Word 文档：需求里写“在作业内容页点击‘新增’按钮，输入内容后提交”，AI 能识别“作业内容页”是页面名，但把“新增”识别成了“查询”；

• 喂 Excel 模板：我特意把“模块名、页面名、功能点”列成表格（比如“作业管理模块-作业内容页-新增作业内容”），结果 AI 直接把“模块名”和“功能点”串在一起，生成的用例变成“作业管理模块-新增作业内容”，页面名直接消失了。

后来才想明白问题在哪：我没给 AI“功能点的提取规则”——比如“功能点必须是‘动词+名词’的格式，如‘新增作业内容’‘查询作业列表’”；而且需求文档里的功能点藏得太散，有的在段落里，有的在表格里，AI 没有“统一提取”的能力。

三、失败里的“优化种子”：这不是结束，是迭代的开始

虽然工具没做成，但这次“失败预演”反而让我摸清了“AI 生成测试用例”的正确方向——不是“喂需求→等结果”的“一键生成”，而是“喂知识库+定模板+多轮调试”的“协同生成”。

方向 1：投喂“测试用例知识库”，让 AI 学我的“测试脑”

我打算把之前积累的高质量用例整理成“测试用例知识库”，让 AI 先“学”会我的测试思维：

• 按“功能类型”分类：比如“新增类功能必测场景”“查询类功能必测场景”“界

面校验必测规则”；

• 给每个场景加“详细说明”：比如“新增类逆向场景：空内容提交（期望结果：页面提示‘请输入内容’）、重复提交（期望结果：页面提示‘请勿重复提交’）、权限不足提交（期望结果：页面提示‘无操作权限’）”；

• 加“优先级标签”：比如“新增空内容提交”标“高优先级、必测”，“新增超长内容提交”标“中优先级、选测”。

AI 学完这些“测试经验”，生成的用例才会有“测试思维”，而不是随便写几句“正确的废话”。

方向 2：定死“用例生成模板+功能清单模板”，把 AI 框在“标准里”

不再让 AI“自由发挥”，而是给它两个强制模板，把输出“框死”：

1. 功能清单模板：我先把“模块名、页面名、功能点”按“模块-页面-功能点”的层级填好（比如“作业管理模块-作业内容页-新增作业内容”），AI 只需要“按清单生成用例”，不用自己识别功能点——从根源上解决“识别错误”的问题；

2. 用例生成模板：把每个字段的格式写死，比如：

• 步骤：必须是“数字+动作+元素”，如“1. 点击作业内容页的‘新增’按钮”；

• 期望结果：必须是“可观察的状态/提示”，如“页面弹出新增作业内容的弹窗，包含‘内容输入框’‘提交按钮’‘取消按钮’”；

• 测试类型：只能选“功能测试、逆向测试、边界测试、异常测试”中的一个。

AI 只能按模板填内容，格式再也不会乱。

方向 3：多轮对话调试，让 AI“边生成边改”

不再是“一次生成等结果”，而是让工具支持“多轮对话式调试”：

• 生成后先让 AI 自检：“请检查这个用例是否包含逆向场景，如果没有，请补充‘新增空内容提交’的用例”；

• 发现格式问题直接追问：“请把步骤改成‘数字+动作+元素’的格式，比如‘1.

点击 XX 按钮’”；

• 补充场景：“请给这个用例加一个‘输入超长字符（200 字）’的边界场景”。

AI 能实时调整，用例质量会越来越贴近我的要求——相当于让 AI 当我的“助理”，我当“指挥”。

最后附“优化后流程”（可以直接画成流程图）：

【优化后流程】

准备阶段：整理测试用例知识库 + 制定功能清单模板 + 制定用例生成模板

第一步：导入功能清单模板（我填好模块名、页面名、功能点）

第二步：投喂测试用例知识库，让 AI 学习测试规则与场景

第三步：AI 生成初始用例，多轮对话调试（补充场景、调整格式、去重）

第四步：导出 Excel，少量手动微调（补复杂场景）

最后：AI 不是“替代测试”，是“测试的助理”

这次失败让我明白：AI 生成测试用例不是“一键出结果”的“魔法”，而是“先给 AI 喂经验、定标准，再让它帮我干活”的“协同”。

现在我已经开始整理“测试用例知识库”了——先从“新增类功能”开始，把逆向场景、边界规则都写清楚；等知识库整理完，就迭代工具的模板和多轮对话功能。

下次迭代后，AI 生成的用例肯定能“又快又准”——毕竟，测试的核心是“质量”，AI 只是帮我们更快达到质量标准的工具，这次失败，不过是“优化路上的预演”而已。如果你也想自己打造自己的测试小工具，相信这篇文章会有所启发。

拓展学习

[1] 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：51

虚幻引擎应用程序性能测试

◆作者：枫叶

一、术语

UE：虚幻引擎（Unreal Engine），是综合性游戏开发平台，涵盖核心技术、数据生成工具及基础支持功能，是将渲染、碰撞检测、AI、图形、网络和文件系统集成为一体的引擎，适用于 PC、主机及移动端游戏开发，也是许多行业领域比如地理信息不可或缺的工具。

UE 像素流：虚幻引擎 UE 提供的一种云渲染技术，通过 WebRTC 协议将服务器端渲染的 3D 画面实时编码为适配流传输至客户端，实现跨平台低延迟访问。

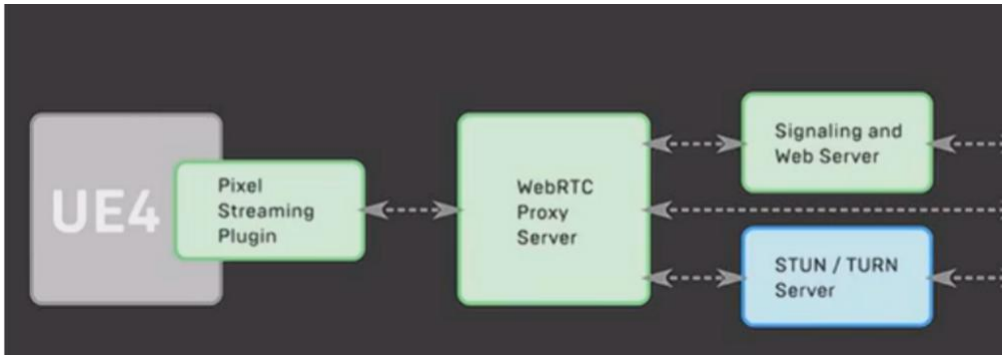
UE 像素流送是 Unreal Engine 官方推出的云推流解决方案，通过显卡服务器将 UE 模型实时渲染并推流至终端设备（如手机、平板、网页等），实现无显卡设备的实时交互体验。

D3D：Direct 3D 的简称，专门用在 windows 系统上用于高性能 3D 图形编程的应用程序接口。

GPU：GPU（Graphics Processing Unit），即图形处理器，核心功能是图形渲染，三大厂商：NVIDIA、AMD、Intel。

二、技术栈及原理

UE 涉及的技术栈包含 Node.js 服务器和 JS 技术。核心的像素流素插件（pixel streaming plugin），集成在 UE4 中，也是集成 NVIDIA NVENC 的地方。它会将视频流进行编码，并将其发送给 WebRTC 代理，将视频流和音频流从虚幻引擎客户端分发到网页浏览器(用户端)。Signaling and web server 信令服务器，负责连接 WebRTC 代理和客户浏览器，相当于 WebRTC 代理的守门员，如下连接图：



WebRTC 含捕获模块和编码。使用 WebRTC 优势在于带宽自适应，即帧数少了，带宽会小。它将信息发回 UE4，发回到像素流插件中，最终回到 NVIDIA NVENC 系统。

三、UE 核心性能监控指标

1.帧率和帧时间

• FPS：平均帧率、最低帧率，监控的目标是稳定在目标帧率（如 60/90/120 FPS）。

• 帧时间：渲染一帧所需的毫秒数，比 FPS 更直观反映卡顿。

理想值的帧时间分别为 16.67ms (60FPS), 11.11ms (90FPS), 8.33ms (120FPS), 33.33ms (30FPS)。

帧时间方差：波动越小，体验越流畅，出现突然的峰值就是卡顿。

2.CPU 性能

• Game Thread：游戏逻辑、蓝图执行时间，通常是首个瓶颈。

• Draw Call Thread（UE4） / CPU Render Thread（UE5）：准备渲染命令的耗时。

• Render Thread：向 GPU 提交命令的耗时。

• GPU Time / RHI Thread：CPU 等待 GPU 或处理渲染硬件接口的时间。

• 各线程 CPU 使用率：核心负载是否均衡。

3.GPU 性能

GPU Time：GPU 渲染一帧的总时间。是判断是否受显卡限制的关键。

GPU Time 是指 GPU 执行一帧中所有渲染命令所花费的总时间，通常以毫秒为单位。它是评估渲染性能的关键指标。

重要阈值：

- 60 FPS 目标：GPU Time ≤ 16.67ms
- 30 FPS 目标：GPU Time ≤ 33.33ms
- VR 目标（90Hz）：GPU Time ≤ 11.11ms

GPU 利用率：通常越高越好，但若接近 100%且帧率不达标，说明 GPU 是瓶颈。

4.内存

- 系统内存：整个应用占用的 RAM，需防泄漏和过度占用。
- GPU 显存：纹理、网格、帧缓冲占用，显存溢出会导致严重性能下降。
- Streaming Pool / Texture Memory：流送纹理池使用情况，溢出会导致纹理弹出。
- 内存分配/释放速率：高频分配可能引发卡顿。

5.渲染与画面

- Draw Call 数量：每帧提交的渲染调用次数，过多会加重 CPU（渲染线程）负担。
- Triangle Count：每帧渲染的三角形数量。
- Shader 复杂度：通过 Shader 复杂度视图定性分析。

6.内容流送与加载

- Streaming Load/Unload Times：场景流送、纹理流送的加载时间。

• Disk I/O：硬盘读取速度，对开放世界至关重要。

• Level Load Times：关卡加载时间。

7.物理与网络

• 物理时间：物理模拟的 CPU 耗时。复杂碰撞和破坏会消耗大量资源。

• 网络带宽与延迟：对于多人游戏，监控网络更新频率、数据包大小及 RTT。

8.平台特定指标

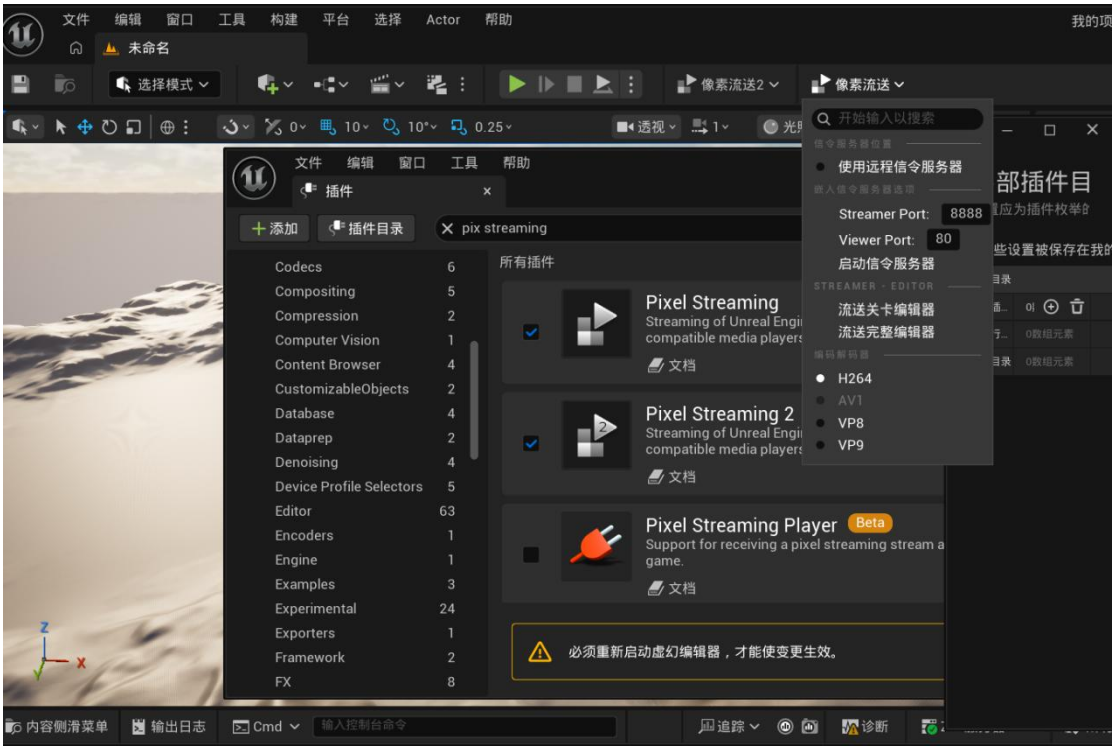
Screen Percentage / Resolution Scaling：动态分辨率缩放效率。

主机：严格的内存预算和帧时间限制。当然还有移动端的功耗与发热等。

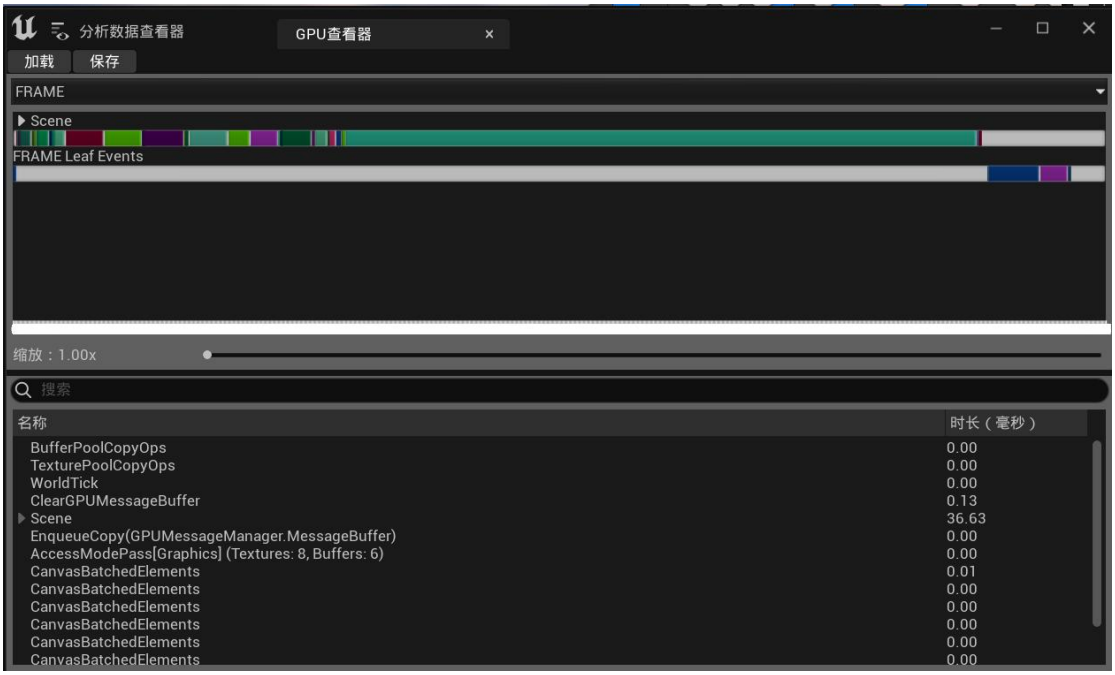
四、UE 内置性能分析工具

1.图形界面性能分析工具

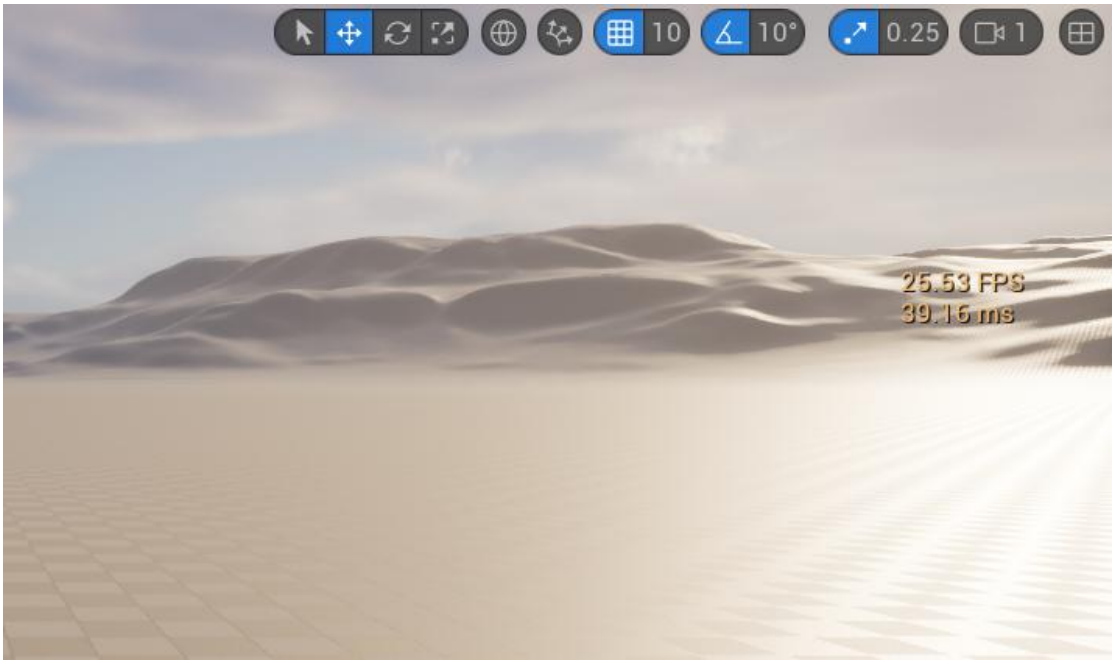
UE 打包像素流出来的系统，可以用 pixel streaming 测试。



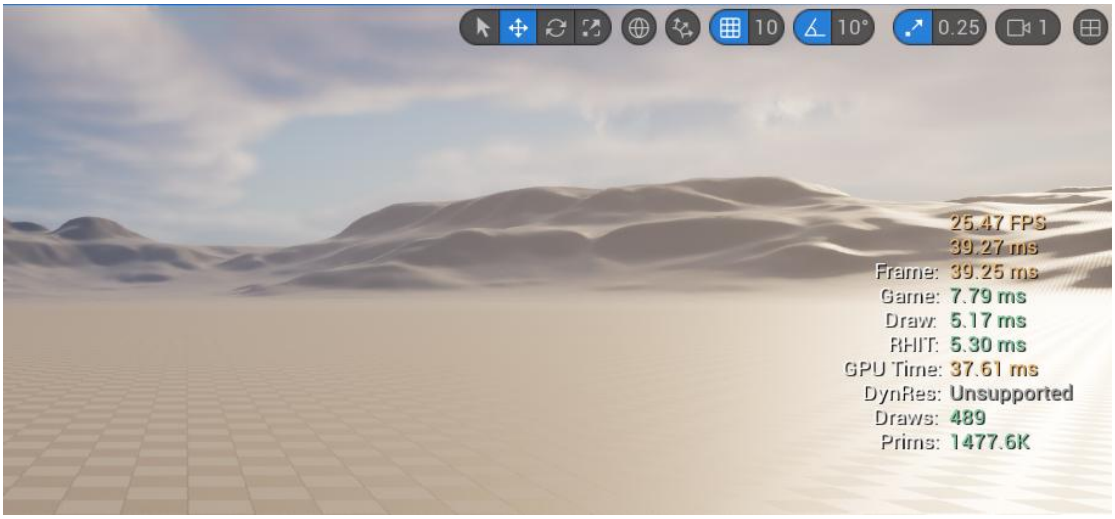
然后按住 Ctrl+shift+逗号键打开 GPU visiualizer 查看 GPU 性能:



输入 stat fps，或者同时按住 ctrl+shift+H 可以看 FPS 帧率，图中 25.53FPS:



stat unit 打开控制台命令，看到更多指标如帧时间数据，图中 39.27ms：



如果是 5.0 版本，搜索 Unreal Insight Reference 打开所有检测通道，可添加“ueproject.exe -statnamedevents -trace=cpu,gpu,frame,log,bookmark,file,loadtime”，注意 5.0 以前版本没有这个功能，可以一键查看帧率、CPU、GPU 等这些核心性能指标。

2.控制台性能分析工具

还可以通过控制台命令监控性能（在编辑器中按 ` 键调出）：

stat unit：最核心命令，显示帧时间、Game 线程、Draw 线程、GPU 时间。

stat fps：显示帧率。

stat memory：详细的内存使用情况。

stat rhi：显示渲染硬件接口数据。

stat scenerendering：场景渲染各阶段耗时。

stat game：游戏线程开销。

stat particle：粒子系统性能。

stat streaming：流送系统状态。

profilegpu：生成 GPU 各阶段的详细时间轴报告。

五、UE 应用程序启动脚本常用命令

1.显示与窗口模式

-windowed 以窗口模式运行游戏。 MyProject.exe -windowed

-fullscreen 以独占全屏模式运行游戏。 MyProject.exe -fullscreen

-ResX=WIDTH -ResY=HEIGHT 设置游戏分辨率。 MyProject.exe -ResX=3840 -ResY=2160

-ForceRes 强制使用指定的分辨率，即使它不被显示器支持。 MyProject.exe -ResX=800 -ResY=600 -ForceRes

-naked 无边框窗口运行（等同于 -windowed -noborder）。 MyProject.exe -naked

-benchnmark 运行一个内置的基准测试，通常在指定帧数后退出。 MyProject.exe -benchmark

2.性能与统计

-fps=MAX_FPS 限制游戏的最大帧率。 MyProject.exe -fps=60

-vsync 开启垂直同步。 MyProject.exe -vsync

-novsync 禁用垂直同步，通常用于性能测试。 MyProject.exe -novsync

-stat unit 显示帧时间和线程统计信息（Game, Draw, GPU）。 MyProject.exe -stat unit

-stat fps 在屏幕上显示帧率计数器。 MyProject.exe -stat fps

-stat scenerendering 显示详细的场景渲染统计信息。 MyProject.exe -stat scenerendering

3. 地图与游戏模式

MAP_NAME 直接启动到指定地图。这是最常用的命令之一。 MyProject.exe MyMapName

-game 直接以游戏模式启动，跳过任何开场电影或菜单。 MyProject.exe MyMapName -game

-server 以专用服务器模式启动。不会加载图形界面。 MyProject.exe MyMapName -server -log

4. 日志与调试

-log 将日志输出到控制台窗口。 MyProject.exe -log

-NoSound 完全禁用声音系统。 MyProject.exe -NoSound

-NoLoadingScreen 禁用加载屏幕。 MyProject.exe -NoLoadingScreen

-TIMEOUT=SECONDS 在崩溃时，等待指定的秒数后再退出，便于收集日志。MyProject.exe -TIMEOUT=30

5.用法举例

专用服务器高性能基准测试

MyProject.exe MyBattleMap -server -log -novsync -fps=0 -benchnmark

MyBattleMap：直接加载地图。

-server：以无图形界面的服务器模式运行。

-log：输出日志到控制台。

-novsync -fps=0：解除所有帧率限制，测试最大性能。

-benchnmark：运行基准测试。

在特定分辨率下进行性能问题诊断

MyProject.exe -windowed -ResX=1280 -ResY=720 -stat unit -stat fps -ExecCmds="r.VSync 0; t.MaxFPS 0"

-windowed -ResX=1280 -ResY=720：在 1280x720 的窗口模式下运行。

-stat unit -stat fps：显示详细的性能统计和帧率。

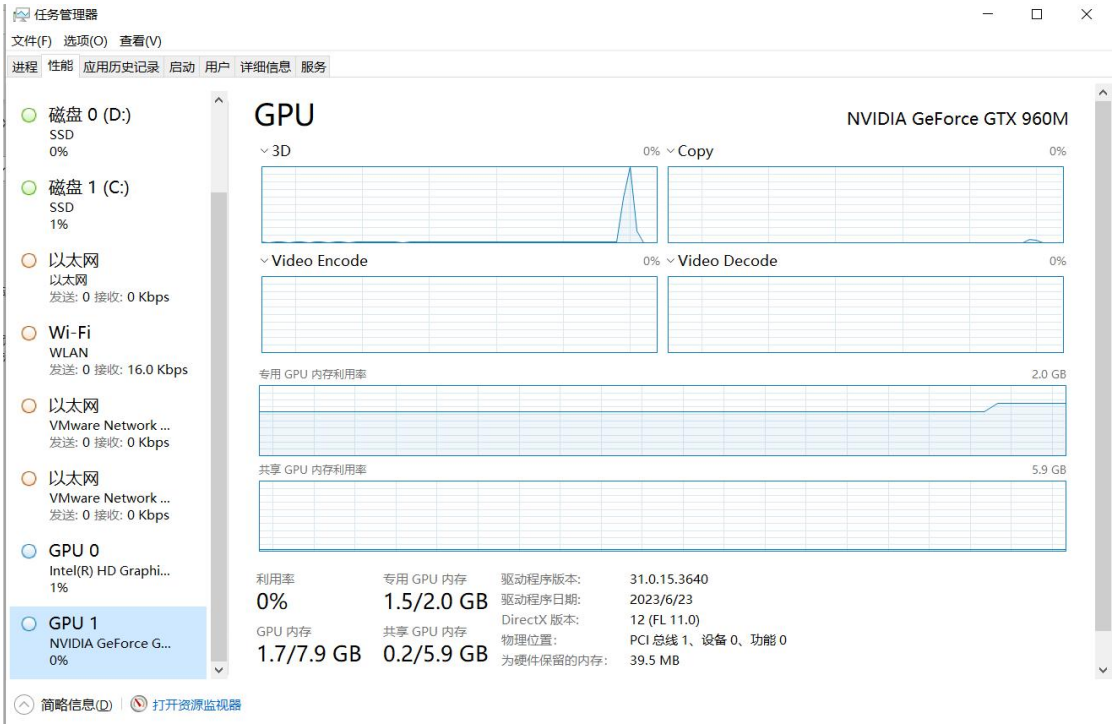
-ExecCmds="..."：一次性执行多条控制台命令，用分号分隔，这里强制关闭垂直同步并解除帧率限制。

六、企业中测试需求

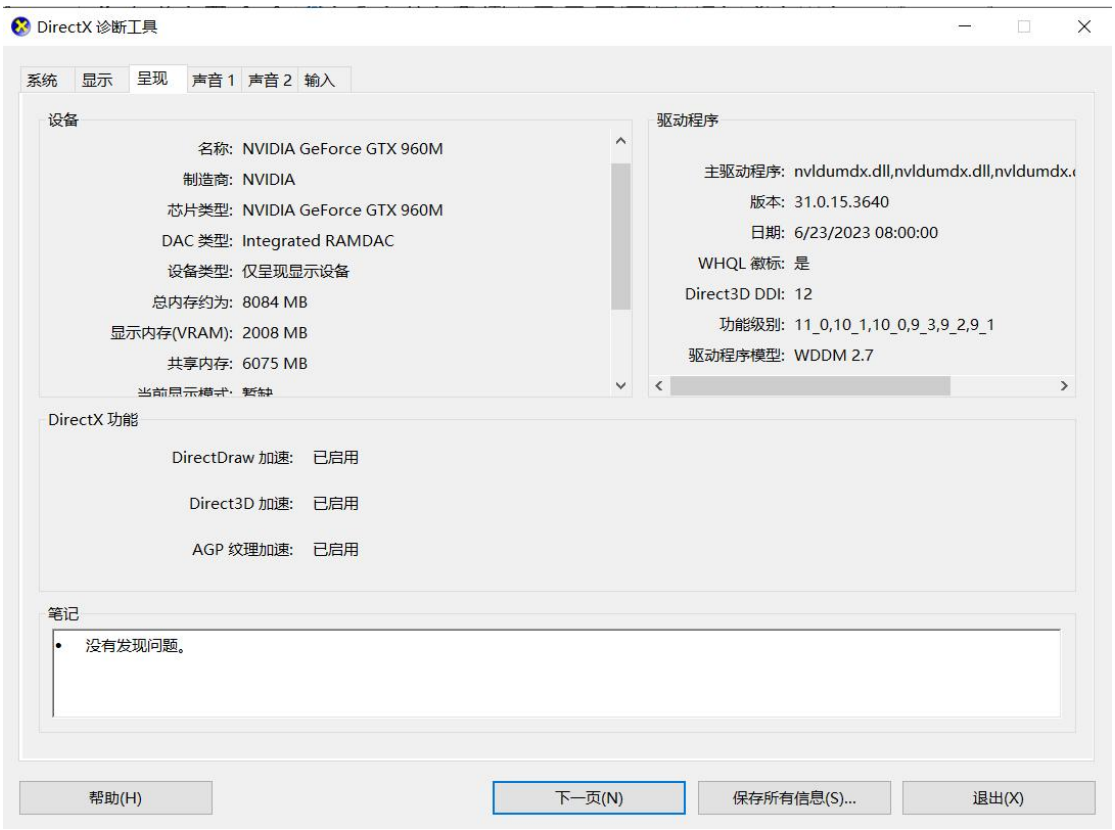
测试目标：验证一台服务器最多可以运行几套 UE 场景。

测试对象：一台部署 UE 系统的 Windows 服务器。

查看未加载 UE 场景时，服务器性能良好利用率=0%，按住 Ctrl+Shift+Esc 打开任务管理器——性能：



还可以通过 CMD，输入 dxdiag，打开 DirectX 诊断工具——“呈现”页签：



七、测试方法

为避免前端修改影响测试结果，我们采用修改像素流的端口，分别启动场景服务和像素流服务后，用浏览器访问相应的端口，已达到模拟 1 个用户 UE 场景的目的。再分别修改不同的端口来启动场景服务和像素流服务，并用浏览器访问，这样逐个增加 UE 场景，来测试 UE 像素流的极限值。

监控指标：GPU 专用内存≤80%，帧率≥40FPS。

监控手段：nvidia-smi 指令查看服务器专用 GPU 内存利用率，结合 stat fps 查看实时帧率和帧时间。

八、性能测试执行

用研发提供的系统部署包，修改场景脚本（以.bat 结尾），并双击启动，场景脚本修改如下：

xxx.exe（UE 可执行主程序） -RenderOffscreen -PixelStreamingIP=127.0.0.1 -

PixelstreamingPort=38888 -InterfaceCallPlatform=cloud -nosound -StartScene=xxxxx

再修改像素流启动脚本，如下所示：

start powershell ".\SignallingWebServer\platform_scripts\cmd\Start_SignallingServer.ps1" --PublicIP=localhost --HttpPort=38887 --StreamerPort=38888 --SFUPort=38889

同时查看 GPU，可以得到运行一个 UE 场景的性能执行结果：GPU 的专用内存内存=11.0G，使用率在 24.7%左右，帧率约 40FPS：

我们修改场景脚本和像素流脚本，再新增 1 个端口，使 UE 场景=2，并查看此时 GPU 的专用内存=21.9G，利用率在 49.2%左右，帧率约 29FPS。

逐步增加 UE 场景，让场景=3，GPU 的专用内存=32.8G，利用率在 73.7%左右，帧率约 20FPS。

在增加 UE 场景，场景=4，GPU 的专用内存=43.3G，利用率在 97.3%左右，已经接近满值，帧率约 14FPS。

初步可以推断，系统服务器（单张显卡）可以支持 3 个 UE 场景此时 GPU 利用率中 73 左右接近阈值 80%，但是帧率还需要优化下。

本文还原了完整的 UE 性能测试全过程，朋友们可以实际操作起来。

基于大语言模型的代码审查助手设计与实现

◆ 作者：blues_C

1. 引言

在软件研发活动中，代码审查始终是保障质量的重要环节。无论是需求迭代中的增量开发，还是历史系统的维护重构，团队都需要通过审查识别潜在缺陷、发现安全隐患、评估可维护性，并尽可能在缺陷进入测试甚至生产之前完成纠偏。但在实际工作中，代码审查往往面临两类典型矛盾：一方面，技术团队希望审查尽量深入，能够覆盖正确性、性能、安全、测试充分性等多个维度；另一方面，审查本身又受到人力、时间和上下文复杂度的限制，尤其在跨语言、多文件或项目级分析场景下，人工审查成本会迅速上升。

传统代码审查流程通常依赖开发工程师或测试开发工程师手工完成。执行者需要先阅读文件内容，再理解模块边界、推断上下文关系、提炼关注点，最后给出结构化建议。这一过程高度依赖个人经验和注意力稳定性。对于简单文件，人工审查尚可快速完成；但在文件数量增加、上下文跨越多个目录甚至需要从整个项目视角理解风险时，审查效率和稳定性都会明显下降。

大语言模型的出现，为代码审查自动化提供了新的实现路径。相比传统静态规则工具，LLM 不仅能够理解自然语言指令，也能对程序结构、错误模式和潜在风险进行一定程度的语义推理。因此，一个实用的代码审查助手，不应只是“把文件内容发给模型”这么简单，而应在输入组织、上下文控制、批次拆分、结果整合、错误回显和交互体验等方面进行完整设计。

本文围绕一个轻量级 AI 代码审查项目展开。该项目以 Python 为实现语言，采用 Gradio 构建交互界面，使用兼容 OpenAI 风格的接口作为模型接入方式，支持单文

件、多文件以及整个项目目录的审查流程。系统既可以直接读取本地项目目录，也支持上传 ZIP 压缩包进行项目级分析。在项目规模较大时，系统会自动拆分多个批次分别审查，再生成最终汇总报告。

和许多 Demo 性质的 AI 工具不同，本项目强调对可落地的审查过程控制。它不仅生成结果，还尽量让用户知道系统究竟审查了哪些文件、向模型发送了什么提示词、何时发生了异常，以及结果为什么会被拆分汇总。换言之，本项目的目标不是替代工程师判断，而是降低代码审查的进入门槛，提高首轮筛查效率，为开发、测试和评审提供一个可解释、可扩展的辅助系统。

1.1 问题背景

代码审查类工具大体可以分为两类。第一类是基于规则和模式匹配的静态分析工具，这类工具在语法错误、空指针、未使用变量、部分安全规则等方面效果稳定，但对于跨文件语义理解、业务合理性判断和测试建议生成往往能力有限。第二类是基于大模型的审查系统，这类系统具有较强的自然语言理解和综合分析能力，但也面临上下文长度、成本控制、结果一致性与数据安全等问题。

如果不对输入内容做管理，直接把大量项目文件拼接发送给模型，会迅速遇到三个问题。第一，输入体积不可控，超过上下文窗口后必须截断，而截断位置一旦不合理，就可能丢失关键语义。第二，文件数量过多时，模型容易给出泛化描述，而不是定位到具体文件的问题。第三，用户很难判断结果是否可信，因为他不知道模型看到了哪些内容、遗漏了哪些内容。

因此，一个真正能用于日常工作的代码审查助手，需要具备明确的输入选择策略、合理的上下文分片机制以及可追踪的输出方式。

1.2 设计目标

结合上述问题，本项目的设计目标可以概括为以下四点：

1. 提供统一的审查入口，支持单文件、多文件和项目目录三种输入模式。

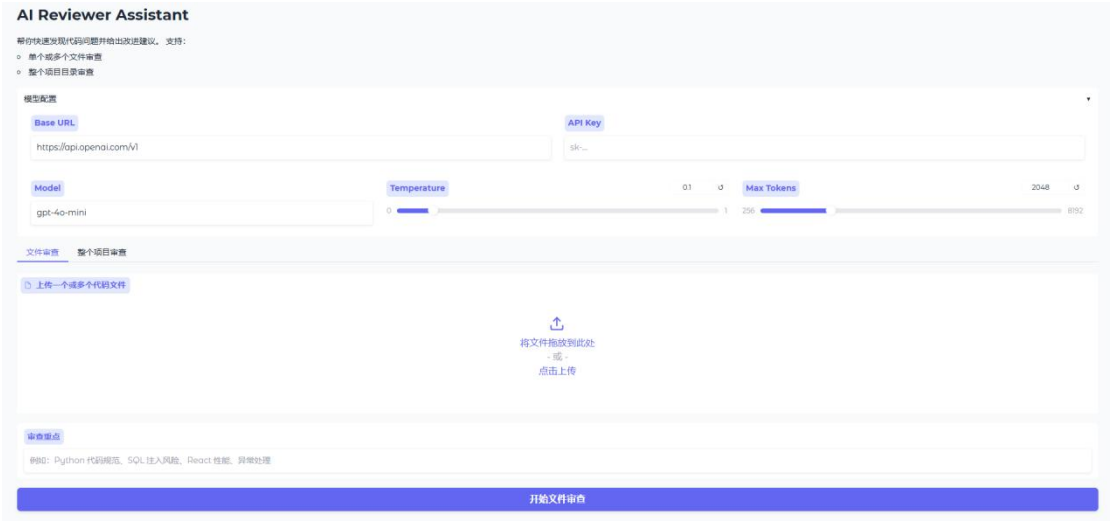
2. 在项目规模较大时自动完成内容裁剪、批次拆分和结果汇总，避免一次性向模型

灌入过长上下文。

3. 兼容 OpenAI 风格接口，便于接入不同厂商或本地模型，降低模型替换成本。

4. 提供面向使用者的可解释信息，包括涉及文件列表、提示词预览和错误回显，以提升审查结果的可理解性。

2. 系统概述



从功能定位看，本项目并不是一个通用代码托管平台插件，也不是一个完整的 CI 审查流水线，而是一个轻量、直接、可本地运行的 AI 审查助手。它的主要使用场景包括：测试开发对某个变更文件做快速风险评估，开发者在提交前对多个核心文件做自检，以及对一个中小型项目进行首轮整体审查。

系统当前提供两类审查模式。第一类是“文件审查”，支持上传一个或多个代码文件。该模式适合针对具体变更进行集中分析，尤其适用于补丁检查、功能点回归前的重点排查等场景。第二类是“整个项目审查”，支持输入本地目录路径或上传 ZIP 包，并允许用户配置包含模式、排除目录、最多读取文件数和每文件最大字符数。这一模式更接近架构性和工程性分析。

从交互方式看，系统采用 Gradio 提供 Web 页面，用户可以直接在界面中输入模型地址、密钥、模型名以及审查重点。审查结果以 Markdown 形式展示，便于阅读结构化结论；同时，系统还会展示本次真正发送给模型的提示词，以及参与审查的文件列表。对于需要调试提示工程或核对输入边界的用户，这一设计非常重要。

3. 具体使用

3.1 环境准备

在首次使用前，需要完成 Python 虚拟环境的创建与依赖安装。项目依赖项已在 `requirements.txt` 中定义，主要包括 `gradio` 和 `openai` 两个核心包。

Windows PowerShell:

```
powershell
python -m venv .venv
.\.venv\Scripts\Activate.ps1
pip install -r requirements.txt
```

Windows CMD：

```
bat
python -m venv .venv
.\.venv\Scripts\activate.bat
pip install -r requirements.txt
```

macOS / Linux：

```
bash
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

3.2 启动方式

项目提供两种启动入口：

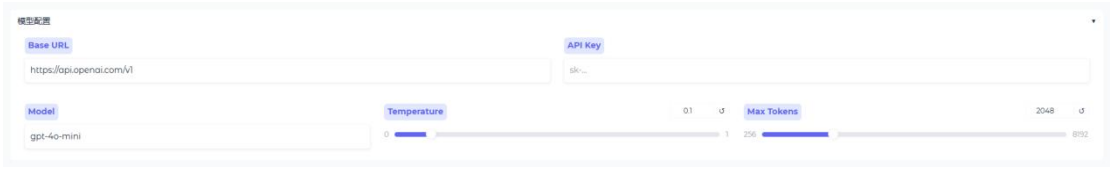
```
bash
 方式一：通过根目录入口启动
python app.py
 方式二：通过包模块入口启动
python -m ai_reviewer
```

启动后，系统会在本地启动 Gradio Web 服务，默认监听 `http://127.0.0.1:7860`。用户可在浏览器中打开该地址进行交互。

3.3 模型配置

在界面顶部的"模型配置"区域，需要填写以下参数：

参数	说明	默认值
Base URL	模型服务地址，兼容 OpenAI 风格接口	`https://api.openai.com/v1`
API Key	模型访问密钥	无（需用户填写）
Model	模型名称	`gpt-4o-mini`
Temperature	输出随机性，值越低越稳定	`0.1`
Max Tokens	最大输出长度	`2048`



用户可根据实际需求切换模型服务。例如，若使用本地部署的模型代理服务，只需将 `Base URL` 指向代理地址即可。

3.4 文件审查模式

文件审查模式适用于对单个或多个代码文件进行集中分析。操作步骤如下：

1） 切换到"文件审查"标签页

2） 点击上传区域，选择一个或多个代码文件（支持 `.py`、`.js`、`.ts`、`.java`、`.go` 等常见格式）

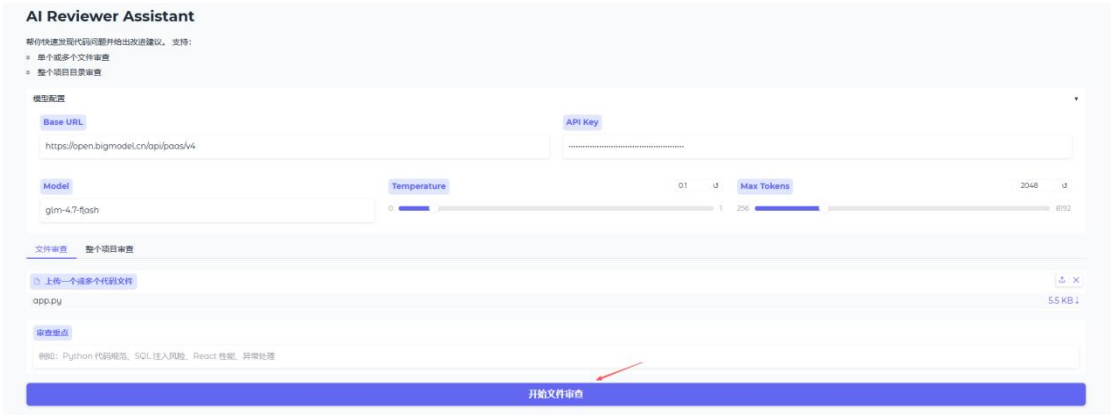
3） 在"审查重点"文本框中填写关注方向，例如：

- `Python 代码规范、异常处理`

- `SQL 注入风险、权限校验`

- `React 性能优化、Hooks 使用规范`

4） 确认模型配置无误后，点击"开始文件审查"按钮

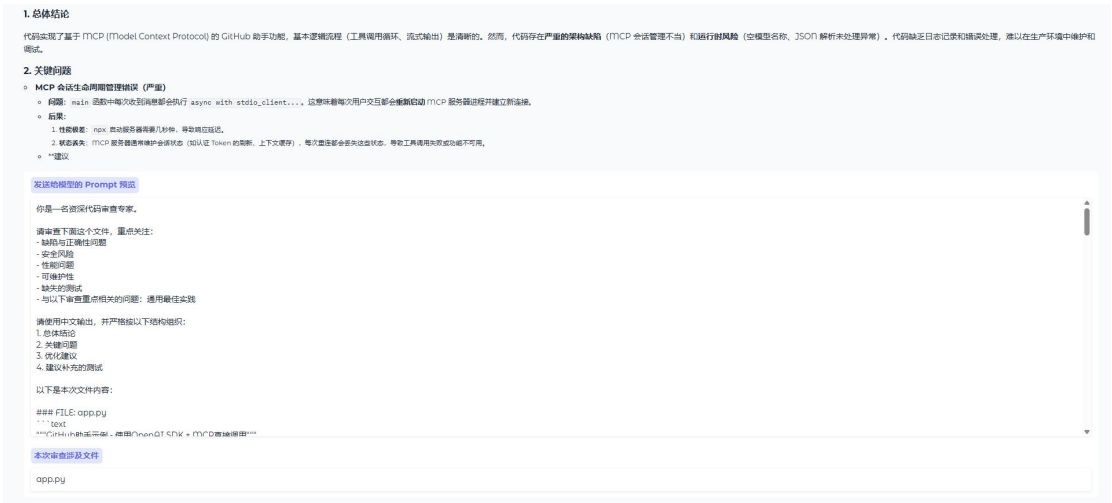


审查完成后，界面将展示三部分内容：

- 审查结果：以 Markdown 格式呈现的结构化审查报告，包含总体结论、关键问题、优化建议等

- Prompt 预览：本次实际发送给模型的完整提示词，便于调试和验证

- 涉及文件：本次审查纳入的文件列表



3.5 项目审查模式

项目审查模式适用于对整个项目目录进行全局分析。系统支持两种输入方式：

方式一：本地目录路径

直接填写项目所在目录的绝对路径，例如：

- Windows：`E:\my_project`

- macOS/Linux：`/home/user/my_project`

方式二：上传 ZIP 压缩包

将项目打包为 `.zip` 文件后上传。系统会自动解压并分析。

项目审查模式下，用户可配置以下参数：

参数	说明	默认值
包含文件模式	需纳入审查的文件类型，逗号分隔	`*.py, *.js, *.ts, *.tsx, *.jsx, *.java, *.go, *.rs, *.cpp, *.c, *.cs, *.md, *.json, *.yaml, *.yml, *.toml, *.sh`
排除目录	需跳过的目录，逗号分隔	`.git, .idea, .vscode, __pycache__, node_modules, dist, build, .next, .venv, venv, coverage`
最多读取文件数	项目审查时读取文件的上限	`40`
每个文件最多读取字符数	单文件内容截断阈值	`12000`

操作步骤：

1. 切换到"整个项目审查"标签页

2. 填写项目目录路径或上传 ZIP 包

3. 根据需要调整包含模式、排除目录、文件数限制等参数

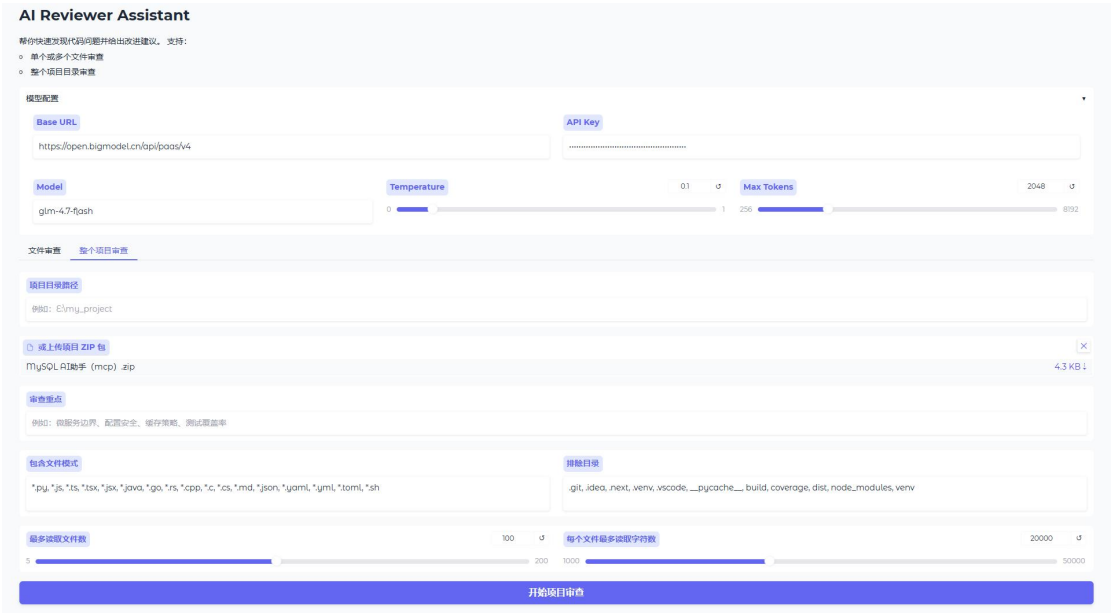
4. 填写审查重点，例如：

   -`微服务边界划分、服务间通信`

-`配置安全、敏感信息泄露`

-`测试覆盖率、关键路径测试`

5. 点击"开始项目审查"按钮



3.6 批次审查机制

当项目文件内容总量较大时，系统会自动启动批次审查机制：

1. 系统将文件片段按字符量拆分为多个批次（默认每批次约 32000 字符）

2. 对每个批次分别调用模型进行审查

3. 收集所有批次的审查结果

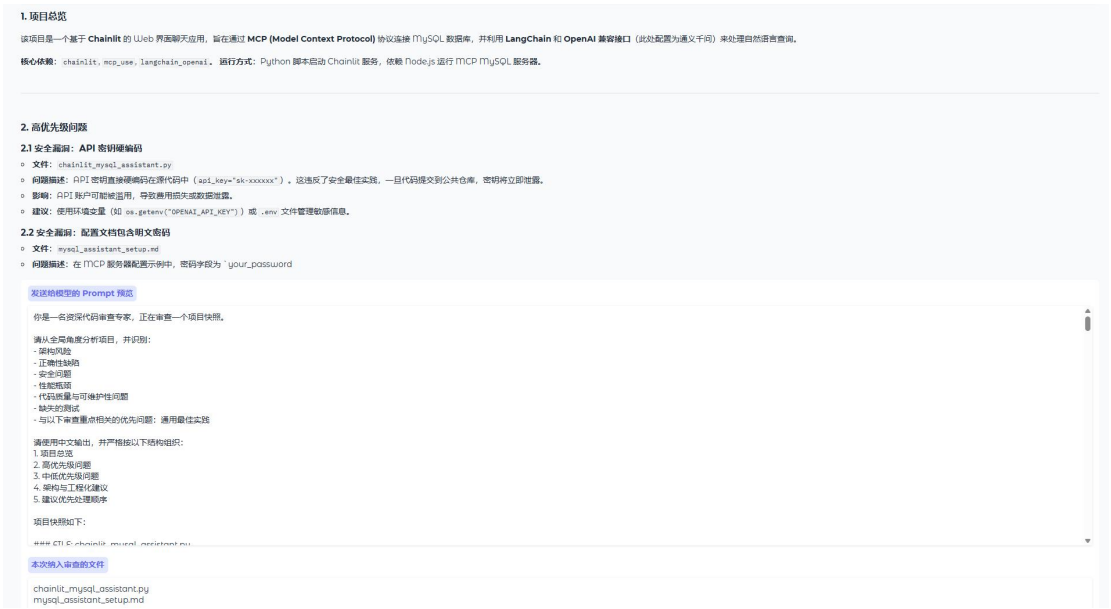
4. 调用综合提示词，将各批次结果合并为最终报告

在 Prompt 预览区域，用户可以看到：

- 批次拆分信息

- 首个批次的提示词内容

- 最终汇总提示词内容



这种机制确保了大规模项目审查的稳定性，避免因上下文过长导致的截断或质量问题。

3.7 结果解读

审查结果采用统一的中文结构化输出格式：

文件审查结果结构：

1） 总体结论

2） 关键问题

3） 优化建议

4） 建议补充的测试

项目审查结果结构：

1） 项目总览

2） 高优先级问题

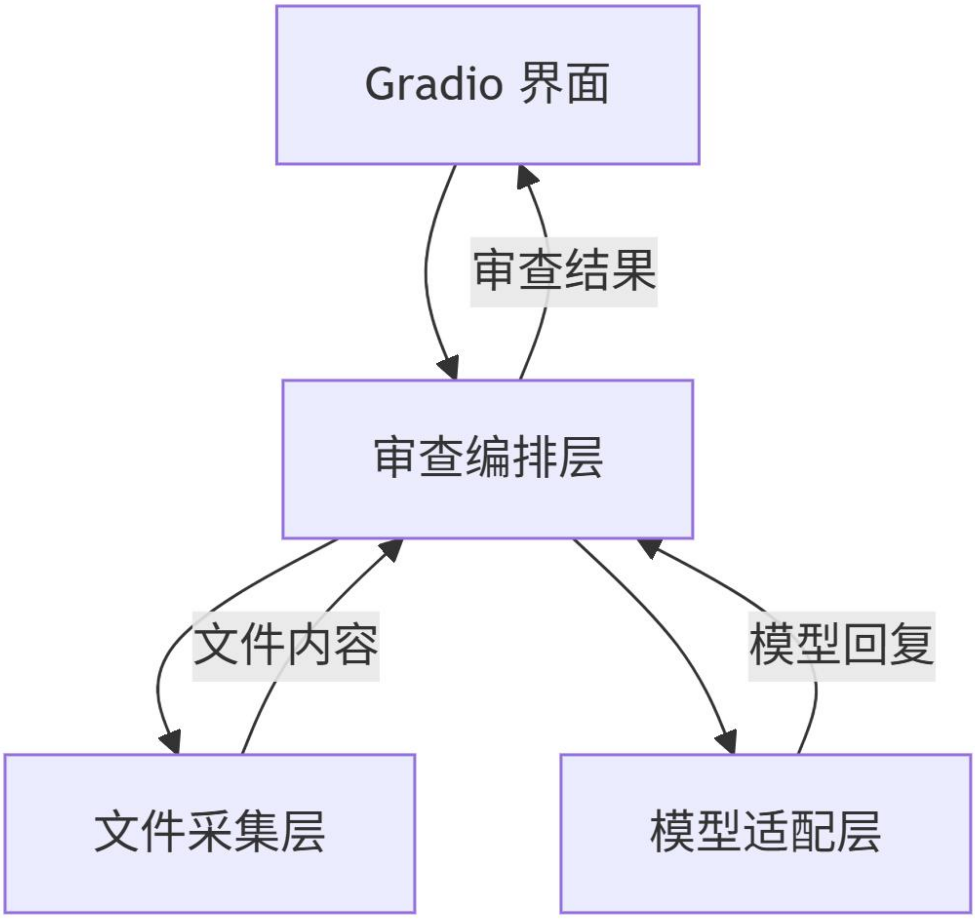
3） 中低优先级问题

4） 架构与工程化建议

5） 建议优先处理顺序

用户可根据报告中的优先级排序，有针对性地进行问题修复和代码改进。

4. 技术架构



本项目整体采用“交互层 + 审查编排层 + 文件采集与模型适配层”的分层设计。虽然项目体量不大，但内部职责边界相对清晰，这使得系统在保持轻量的同时，具备较好的后续扩展基础。

4.1 交互层

交互层位于 `ai_reviewer/ui.py` 中，负责页面渲染、参数收集、按钮事件绑定和异常提示。系统使用 Gradio `Blocks` 组织界面，分为模型配置、文件审查和项目审查三个主要区域。文件审查区域支持多文件上传，项目审查区域支持目录路径和 ZIP 包两

种输入方式，并开放包含模式、排除目录、最大文件数、最大字符数等参数。

交互层本身不承担审查逻辑，只负责把用户输入传递给后端函数，并把返回的三个核心结果展示出来：审查总结、提示词预览、涉及文件列表。这种设计减少了 UI 与业务逻辑的耦合度，使得后续替换前端框架时，不必重写核心审查流程。

4.2 审查编排层

编排层位于 `ai_reviewer/reviewer.py`，负责组织文件内容、构造提示词、调用模型、处理分批审查以及生成最终结果。该层是项目的核心，也是整个系统从“文件读取器”升级为“审查助手”的关键所在。

编排层中定义了两个核心数据结构：`ReviewResult` 和 `FileSnippet`。前者封装最终摘要、提示词预览和参与审查的文件集合；后者封装单个文件的相对路径与文本内容。这样的抽象使系统不再直接围绕“原始文件路径”处理问题，而是围绕“可送审的文件片段”处理问题，为后续引入缓存、增量审查甚至 Git diff 审查提供了良好接口。

4.3 文件采集层

文件采集层负责解决“哪些文件应该被送入模型”这一问题。项目通过 `collect_uploaded_file_snippets` 处理文件上传场景，通过 `collect_project_files` 处理目录遍历场景。目录审查时，系统会先根据包含模式和排除目录进行筛选，再对二进制文件进行识别和跳过，最后按照最大文件数与最大字符数对数据规模进行控制。

这一层的价值在于，它把模型成本控制前置到了文件采集阶段。也就是说，系统不是在模型调用失败后再考虑“输入太大”，而是在进入模型前就主动做边界治理。

4.4 模型适配层

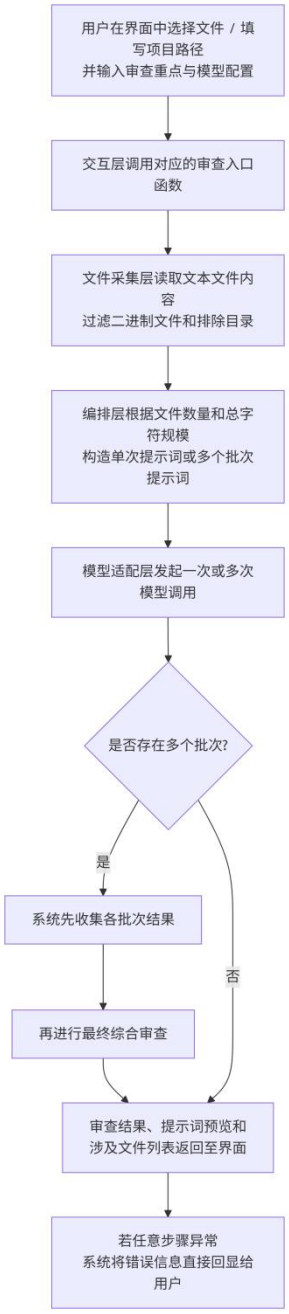
模型适配层通过 `OpenAI` 客户端对兼容 OpenAI 风格的接口进行调用。系统使用 `base_url`、`api_key` 和 `model` 三个输入参数完成模型路由配置，因此既可以对接官方接口，也可以对接代理服务或其他兼容实现。

值得注意的是，系统没有假设模型返回内容一定是纯字符串，而是通过

`_extract_message_text` 与 `_extract_response_text` 做了兼容解析。这样做的原因在于，不同兼容接口对 `message.content` 的组织形式并不完全一致，有些返回字符串，有些返回内容数组，甚至有些会把主要内容放在扩展字段中。通过统一提取逻辑，系统降低了兼容风险。

4.5 数据流分析

系统完整的数据流可以概括如下：



1. 用户在页面中选择文件或填写项目路径，并输入审查重点与模型配置。
2. 交互层调用对应的审查入口函数。
3. 文件采集层读取文本文件内容，过滤二进制文件和排除目录。
4. 编排层根据文件数量和总字符规模构造单次提示词或多个批次提示词。
5. 模型适配层发起一次或多次模型调用。
6. 若存在多个批次，则系统先收集批次结果，再进行最终综合审查。
7. 审查结果、提示词预览和涉及文件列表返回至界面。
8. 若任意步骤异常，系统将错误信息直接回显给用户。

这种流程兼顾了交互体验与稳定性。尤其是在项目级审查中，系统没有把“超长上下文”问题留给模型兜底，而是通过预先分批和后续综合，实现了更稳妥的推理组织方式。

5. 核心实现

5.1 单文件与多文件统一审查

在许多工具中，单文件和多文件会被当成两套不同流程处理，导致逻辑重复，维护成本上升。本项目采用统一抽象：无论上传一个文件还是多个文件，都会先转化为 `FileSnippet` 列表，再进入后续处理流程。这样一来，单文件只是多文件场景的特例，而不是另一条完全独立的实现路径。

统一处理还有一个直接好处，即多文件审查天然支持批量上下文组织。系统能够在同一轮审查中同时分析多个关联文件，例如某个控制器、服务层和配置文件的组合。这比孤立地审查单文件更贴近真实工程问题，也更容易让模型识别跨文件接口不匹配、配置缺失或职责边界混乱等问题。

5.2 项目目录与 ZIP 包审查

项目审查入口支持两种数据源：本地目录路径和 ZIP 压缩包。对于本地目录，系统直接遍历文件树；对于 ZIP 包，系统会先解压到临时目录，再执行同样的筛选与读

取逻辑。解压结束后，临时目录会在 `finally` 代码块中清理，避免中间文件残留。

这一设计的意义在于，它将“输入来源”与“审查流程”解耦。后续如果要接入 Git 仓库快照、CI 产物目录或对象存储下载包，也只需新增新的输入解析方式，而不需要大改核心审查逻辑。

5.3 批次拆分与汇总策略

批次拆分是本项目最重要的工程策略之一。系统通过 `_build_project_batches` 根据累计字符量把多个文件片段拆分为若干批次，每一批次都由多个 `FILE` 块组成。若总内容较小，则直接进行一次审查；若内容过大，则先对每个批次分别审查，再调用综合提示词，把所有批次结果合并为最终报告。

相比简单截断，这种分层审查策略有两个优势。第一，它尽量保留了每个文件片段的完整性，减少“一个文件被拦腰切断”带来的语义损失。第二，它把模型的推理任务拆成了“局部分析”和“全局综合”两个阶段，更符合人类审查复杂项目时的思考方式。

当然，这一策略也存在局限。例如，当两个高耦合文件恰好被分到不同批次时，模型在第一阶段可能无法直接发现跨批次问题。但从稳定性与通用性权衡看，这种设计已经显著优于一次性粗暴拼接。

5.4 提示词工程设计

系统中存在四类提示词模板：文件审查提示词、项目审查提示词、项目批次审查提示词和项目结果综合提示词。它们共享同一个核心思想，即明确限定模型角色、审查维度和输出结构。

例如，系统会要求模型重点关注缺陷与正确性、安全风险、性能问题、可维护性和缺失测试，并在输出中按“总体结论”“关键问题”“优化建议”等结构组织内容。对项目级审查，则进一步强调高优先级问题、架构建议和优先处理顺序。这样的结构化约束可以显著降低结果发散程度，提升报告的一致性和可读性。

值得一提的是，系统允许用户输入“审查重点”。这一字段不会替代默认审查维

度，而是作为额外优先级注入提示词。这样既保证了审查基础面不会缺失，又允许用户针对性能、异常处理、权限控制、测试覆盖率等特定方向做重点放大。

5.5 OpenAI 兼容接口调用

系统采用兼容 OpenAI 风格的调用方式，而不是把模型厂商写死在代码中。这个决定的工程价值非常明显：一方面，用户可以根据成本、速度或合规要求切换模型服务；另一方面，项目在部署环境中的适配门槛更低，只要目标服务遵守相近的接口约定即可。

在返回值处理上，系统并没有假设 `response.choices[0].message.content` 一定可直接使用，而是增加了兼容层来提取字符串、数组文本乃至某些扩展字段内容。对于模型输出为空的情况，系统也会显式抛出异常，而不是默默返回空字符串。这样做的好处是，一旦兼容接口行为异常，使用者能够第一时间看到错误原因，而不是误以为“项目没有审查结果”。

5.6 错误处理与可解释输出

如果一个 AI 工具只在成功时展示结果，而在失败时什么都不说，那么它在实际使用中往往难以建立信任。本项目在交互层专门提供了统一错误格式，将异常以 Markdown 形式返回到结果区。同时，`launch` 阶段开启了 `show_error`，便于开发和调试。

此外，系统会展示 prompt 预览与涉及文件列表。这一点非常关键。它意味着用户可以判断：本次审查究竟包含了哪些文件，某些文件为什么没进来，模型是在什么约束下生成结论的。对于一个面向工程使用的审查助手来说，这种“输入可追溯”能力比单纯追求结果华丽更重要。

6. 工程化设计

6.1 包结构重组

项目当前采用 `ai_reviewer` 包承载实际业务代码，根目录保留 `app.py` 和

`reviewer.py` 作为兼容入口。这种结构看似简单，实际上体现了较好的工程习惯：真实实现聚合到包内，便于后续继续拆分模块；外部入口保持兼容，避免已有启动方式和旧引用失效。

这一设计为未来扩展测试代码、命令行入口、配置模块甚至安装打包提供了基础。对于一个处于演进阶段的小项目而言，尽早完成“脚本式结构”向“包式结构”的迁移，是很有价值的工程动作。

6.2 文件过滤与边界治理

在目录扫描阶段，系统默认排除了 `.git`、`.vscode`、`node_modules`、`dist`、`build`、`.venv`、`coverage` 以及 `__pycache__` 等目录。这一策略有三方面价值：减少无关内容进入模型、降低审查成本、避免噪声文件干扰结论。

同时，系统还会识别二进制文件并跳过。这一点虽不复杂，却十分必要。因为在真实项目中，图片、编译产物、缓存文件和部分非文本资源都可能被遍历到，如果不做过滤，不仅浪费 token，还可能导致异常或无意义输出。

6.3 参数开放与默认值设计

系统开放了多个可调参数，包括包含模式、排除目录、最大读取文件数、每文件最大字符数、温度和最大输出 token 数。它们共同构成了“审查边界控制面板”。

从默认值选择看，项目采用了相对保守的策略。最大项目文件数和每文件最大字符数都有默认上限，温度默认较低，意味着系统更倾向于输出稳定、收敛的审查结论，而不是富于创造性的发散建议。对于代码审查场景而言，这种取舍是合理的。

7. 质量与风险分析

从质量保障角度看，本项目本身并不声称能够替代人工审查。它更适合做“第一轮自动分析器”或“审查辅助器”。其优势在于速度、覆盖面和结构化输出，尤其适合帮助测试开发或开发者快速识别高风险区域。

但系统同样存在若干风险。首先，模型审查本质上依赖输入上下文，如果关键文件

未被纳入，结果自然会失真。其次，模型输出具有概率性，即使提示词相同，也不保证结论完全一致。再次，若用户将包含敏感逻辑的源码发送至外部模型服务，则还需额外考虑数据泄露与合规要求。

针对这些风险，当前项目已经提供了若干缓解措施，如文件过滤、结果回显、提示词预览、输入规模限制等。但从生产使用角度看，后续仍应增加更严格的敏感目录排除、脱敏策略、本地模型支持和审查日志能力。

8. 场景化验证

本项目当前更适合通过功能场景而非学术基准来评估。基于现有实现，可以设计以下四类验证场景：

场景 A：单文件快速审查。 用户上传一个核心业务文件，重点关注异常处理与边界校验，系统应返回结构清晰的中文建议，并列出该文件作为唯一输入。

场景 B：多文件联动审查。 用户同时上传控制器、服务层和配置文件，系统应在同一轮中分析它们之间的潜在协同问题，而不是只给出孤立的单文件意见。

场景 C：项目级工程审查。 用户指定本地项目目录，并通过包含模式限制为 `.py, .md, .yaml`，系统应排除无关目录，按文件上限采样并生成项目级总览。

场景 D：大项目批次汇总。 当文件内容总量超出单次输入合理范围时，系统应自动拆分批次，分别审查后再综合，最终向用户展示完整汇总与首批提示词预览。

这些场景覆盖了系统最关键的四项能力：输入组织、多文件协同、项目级筛选和超长上下文治理。与追求绝对数值指标相比，这种面向真实使用链路的验证方式，更能反映该类工具的工程价值。

9. 局限性与未来工作

尽管本项目已经具备较完整的可用形态，但仍存在明显的提升空间。

第一，当前系统以“项目快照”方式审查代码，尚未直接支持 Git diff 级增量审查。因此在真实团队流程中，它更适合作为独立辅助工具，而非直接嵌入代码评审平台。

第二，系统目前不具备长期上下文记忆与领域知识注入能力。如果项目包含大量行业术语、隐式约束或内部框架约定，模型可能难以在缺乏额外背景的情况下给出足够精准的判断。

第三，当前项目尚未引入自动化测试或基准数据集来量化评估审查质量，这意味着系统更偏向“工程实用型工具”，而不是“研究型评测系统”。

第四，在安全层面，系统虽然做了文件过滤和输入控制，但尚未提供敏感文件黑名单、脱敏转换、私有模型接入建议模板等更强保护措施。

面向未来，可以从以下方向继续演进：

1） 引入 Git diff 审查模式，只分析变更文件及其依赖上下文。

2） 增加结构化报告导出能力，如 Markdown、JSON 或 HTML。

3） 引入规则引擎与 LLM 联合审查机制，让规则工具负责确定性问题，模型负责语义性问题。

4） 支持本地模型或企业代理部署，以满足更严格的合规需求。

5） 增加结果评分、风险标签和修复建议模板，提升报告可执行性。

10. 结论

本文围绕一个面向工程实践的 AI 代码审查项目，分析了其设计目标、系统结构与关键实现。该项目并不追求庞杂能力，而是聚焦于一个现实问题：如何让代码审查从“全靠人读、全靠人想”转向“由系统先做一轮高覆盖率筛查，再由工程师做最终判断”。

通过统一的文件片段抽象、项目目录采集、批次拆分与汇总、兼容 OpenAI 风格接口的模型调用，以及提示词预览和错误回显等机制，系统构建了一条相对完整的审查链路。它既能够处理单文件快速审查，也能够对中小规模项目做首轮结构化分析。

从工程视角看，这类工具的价值在于是否能够稳定地帮助团队更早发现问题、更快形成审查结论，并降低进入深度审查之前的认知成本。就这一目标而言，本项目已经具备较清晰的落地形态。

随着后续在增量审查、知识注入、结构化报告和安全治理上的持续增强，这一类轻量级 AI 审查助手有望从单机工具逐步演化为研发流程中的常规质量组件，为测试开发、研发自检和技术评审提供更持续的支撑。

附：项目仓库

项目源码（Gitee）：https://gitee.com/blues_c/ai-code-reviewer

拓展学习

[2] 解锁 AI 测试试听课，获取大厂同款 AI 工具包

咨询：微信 atstudy-js 备注：51

测试人如何培养自己的风险意识

◆作者：九哥

《汉书·霍光传》记载了一则“曲突徙薪”的寓言：一位客人拜访主人，见主人家烟囱笔直、柴草堆在灶旁，便提到“改成曲烟囱、移走柴草，否则恐发火灾”，主人却置若罔闻。

不久后，主人家果然失火，邻里合力将火扑灭。主人摆酒谢恩，烧伤者居上座，救火者按功排位，唯独忘了那位提预警的客人。旁人点醒：“若早听客人之言，何来火灾与宴客之费？如今救火者受赏，防患者无名，岂不合理倒置？”主人这才幡然醒悟，派人去请客人。

这则寓言恰恰点透了风险识别的核心——测试人的价值从不是事后补救，而是事前防患于未然。对测试人而言，对待测试的态度决定了职业底线，那么识别风险的能力，就决定了职业高度。

【小伊的“线上惊魂”】

上周一下班，刚入职一年的测试新人小伊，抱着电脑蹲在公司楼下的咖啡店，红着眼圈找我吐槽。她刚经历了职业生涯第一次线上惊魂，内心满是忐忑。

事情的起因很简单：她负责的积分兑换小程序，上线前仅覆盖了兑换比例、库存扣减等核心流程，却遗漏了“用户重复提交兑换请求”的关键测试点——只验证了前端防重放控制，未校验后端接口幂等性，导致用户利用脚本重复提交，部分积分被重复扣除。上线两小时，就出问题了。虽及时完成数据回滚与用户补偿，但小伊仍被领导约谈了。

“我从来没想过这些‘小异常’会引发线上问题。”小伊搅动着咖啡，语气满是懊悔。我没有急着讲大道理，而是给她讲了 2 个因“忽视风险”酿成的行业重大事故——

比起她的小失误，这些案例更能说明：测试人的风险敏感度，直接决定着产品的生死线。

第一个是 2019 年拼夕夕“100 元无门槛券”Bug 事件，典型的电商高并发场景下规则校验缺失灾难。当时平台上线优惠券活动，因为高并发场景下“生成-领取-使用”全链路风控测试和规则穿透与压测不充分。活动上线后，系统出现异常，用户可无限领取 100 元无门槛券且支持叠加使用，有人以 0.01 元抢购家电，有人批量下单转卖，漏洞持续数小时才修复。传言平台直接损失超千万元，不仅紧急下架活动、封禁异常订单，还引发大规模用户投诉与舆论嘲讽，品牌口碑与财务双双承压。

这个案例提醒我们：测试绝不能只满足于“正常流程通了”，必须开展全链路规则穿透测试、高并发压测，同时预设异常场景熔断机制，把风险掐灭在上线前。

第二个是 2024 年 CS 公司全球 Windows 蓝屏事件，因为全量兼容性、边界测试及回滚机制验证等关键测试不到位。当 CS 公司为 Falcon 安全软件推送快速响应内容（RRC）更新后，全球约 850 万台 Windows 设备出现越界内存读取问题，集体蓝屏死机，波及航空、医疗、911 急救等关键领域，引发全球 IT 系统大规模停机。最终公司股价暴跌 32%，市值蒸发 250 亿美元，还面临股东集体诉讼。这个案例给所有测试人敲响警钟：无论配置更新、组件迭代还是紧急修复，都绝不能降低测试标准。

小伊听完后沉默良久，缓缓说道：“原来我不是运气差，而是根本没有风险思维——总想着‘没问题就好’，却从没想过‘可能会出什么问题’。”而这，正是多数测试人从“合格”迈向“优秀”的最大鸿沟。

【做一个具有风险意识的测试人】

风险意识并非天赋，而是可通过刻意练习培养的核心能力。结合多年测试与实战经验，可以从下面五个方面搭建完整的风险防控体系。

1. 前置介入：从需求端锚定风险源头

优秀的测试人从不是“等开发完再测”，而是主动前置到需求阶段，提前识别风险。拿到需求文档后，别急着梳理测试用例，先问自己三个核心问题：功能的核心价值是什么？用户可能存在哪些“非正常使用”场景？一旦出现问题，影响范围与严重程度如何？

然后对照这些问题答案，梳理关键测试点。以电商优惠券功能为例，核心价值是引流转化，非正常场景包括批量领券、重复使用、超范围核销等，风险影响覆盖财务损失与用户信任，想清楚这些，就能针对性设计测试场景；积分兑换功能的核心价值是引流转化，潜在异常场景包括重复提交、参数篡改、库存不足，对应的关键测试点就需要涵盖后端接口幂等性（通过 token 或请求唯一标识实现）、前端防重放与后端二次校验的双重保障、库存扣减的原子性以及异常返回时的友好提示。

同时，要敢于在需求评审中提出质疑。若开发提出“接口幂等性暂时不做，后续迭代补充”，务必明确反对：接口上线后就可能面临并发请求，风险不会等“后续”再爆发。及时向上汇报，并做好留痕。

2. 建立清单：让风险识别有章可循

经验的价值，在于可复用、可规避重复踩坑，建议搭建“业务+场景”双维度风险清单并定期更新。每次测试结束后，将遇到的风险点、线上问题整理成“风险清单”，按业务类型（电商、工具、安全等）分类，标注风险等级、触发条件与应对方案。比如电商支付模块中，支付结果回调重复是高风险点，多由网络延迟或第三方重试触发，必测测试点包括回调请求唯一标识校验、支付状态幂等更新、回调超时重试机制，应对方案可采用数据库订单状态加锁与重试次数限制（不超过 3 次）的组合；配置更新模块的高风险点是兼容性冲突，在多版本设备或系统环境下易触发，需覆盖全版本兼容性测试（涵盖主流系统与机型）、边界值校验（如配置为空或超范围）、灰度更新触发条件验证，同时提前预留回滚配置，经小流量验证无异常后再全量推送。

清单需定期迭代更新，将行业重大事故纳入其中，拆解问题根源与规避方法。长此以往，就能形成风险识别“条件反射”——看到对应功能，立刻联想到潜在风险点，大幅提升测试覆盖率。

3. 穿透测试：不做表面功夫，深挖隐性风险

很多测试人陷入“功能点全覆盖”的误区，却忽略了“场景穿透”的重要性。真正的高风险点，往往隐藏在功能交叉处、异常边界与高并发场景中。以登录功能为例，不能只满足于验证“正确账号密码登录”，还要延伸测试密码错误 3 次后的锁定机制、登

录态过期后操作的兼容性、高并发登录（如 1000+请求/秒）的稳定性，以及篡改登录态参数能否绕过验证；测试配置更新功能时，除了验证“更新成功”的正常场景，还要重点测试更新失败后的回滚是否生效、部分设备更新失败是否影响整体系统、配置参数超范围是否能触发熔断机制。

4. 兜底保障：建立“测试-监控-回滚”闭环

再周全的测试也无法完全规避线上风险，因此必须建立完善的兜底机制。首先，上线前推行灰度发布，先向小部分用户开放，观察无异常后再全量上线，缩小风险影响范围；其次，协同运维搭建多级监控体系，不仅监控功能可用性，还需覆盖资源使用率、接口响应时间、异常请求量等指标，实现“早发现、早预警”；最后，务必提前验证回滚机制，确保出现问题时能快速回滚至稳定版本，避免风险扩大。

5. 保持敬畏：把风险意识刻进职业习惯

测试工作最忌“麻痹大意”——觉得“功能简单不用细测”“之前没问题这次也安全”。但线上问题从不讲情面，一个不起眼的配置错误、一行未校验的代码，都可能引发灾难性后果。

养成两个小习惯：一是上线前做 10 分钟风险复盘，梳理测试遗漏点与潜在风险，确认无隐患后再放行；二是定期复盘线上问题，分析“测试时为何未发现”，优化测试流程与用例设计。唯有敬畏线上、敬畏风险，才能摆脱惯性思维，始终保持清醒的测试判断力。

三、结语

测试人的核心价值，从不是火灾爆发后奋力救火，而是提前发现“直烟囱”与“近旁柴草”，从根源上阻止灾难发生。这份工作或许不被聚光灯照耀，甚至会被质疑“小题大做”，但正是这份未雨绸缪，守护着产品的稳定运行，也支撑着我们的职业尊严。

对测试人而言，风险意识从不是额外负担，而是立足行业的核心竞争力。它能让我们在复杂业务逻辑中保持清醒，在紧张上线节奏中坚守底线，在突发线上问题中从容应

对。从今天起，做一个“爱操心”的测试人：多问一句“可能出什么问题”，多测一个异常场景，多留一个兜底方案。

毕竟，测试的终极目标从不是“找不出 Bug”，而是让产品更安全地服务用户。这份安全，始于每一次对风险的警惕，成于每一次对细节的坚守。

拓展学习

[3] 【全国软件测试学习交流群】学习交流

咨询：微信 atstudy-js 备注：51

用 Coze 做测试：从“点工”到“AI 训练师”的实战之路

◆ 作者：半夏

一个测试工程师的 24 小时

- 09:00 收到 30 页 PRD，下午要出用例
- 15:00 写了 60 条用例，发现登录规则改了，重来
- 22:00 回归测试挂了一个老功能，没用例，手工跑
- 00:30 还在写测试报告

痛点本质：需求理解靠人、用例设计靠手、缺陷分析靠猜。在传统的测试流程中，测试工程师需要反复阅读冗长的产品需求文档，逐字逐句理解后凭个人经验手动编写用例。这个过程不仅耗时，还容易遗漏边界情况或异常场景，尤其是当需求变更时，测试用例的同步更新又是一轮繁重的手工劳动。

Coze（扣子）不是又一个 Chatbot，而是一个可编排、懂业务、能执行的 AI 智能体平台。工作流本质上是一个有向无环图（DAG），由多个节点按照特定逻辑连接而成，支持条件分支、并行处理和错误恢复，更贴合真实业务场景。本文不讲概念，直接上硬货——四阶 AI 测试工作流，附代码和配置。

一、Coze 凭什么能干活？

能力	对测试的价值
可视化工作流	拖拽编排逻辑，不用写代码
知识库	上传 PRD/接口文档，AI“学会”你的业务
插件系统	调数据库、发 HTTP、写 Jira，自动提单
自愈定位	元素变了自动重试，脚本维护量↓70%

Coze 集成了 GPT-4、DeepSeek 等多种大模型能力，其核心优势在于通过可视化拖拽节点的方式编排逻辑，无需编码即可构建复杂流程。同时，知识库集成功能允许上传项目文档、接口规范、测试标准等，让 AI 真正具备“领域知识”。

二、实战案例：一个完整的工作流设计

为了让你更直观地理解工作流的设计过程，这里以“用户登录功能”为实战案例，完整拆解从零搭建到上线运行的全流程。

第一步：明确输入与输出

假设我们收到这样一份需求文档：

登录功能需求：

1.用户可通过手机号和密码登录

2.手机号为 11 位数字，以 1 开头

3.密码长度为 8-16 位，需包含大写字母、小写字母和数字

4.登录成功：跳转至首页，显示“欢迎，[用户名]”

5.登录失败（手机号或密码错误）：提示“手机号或密码错误”

6.登录失败（手机号或密码为空）：提示“请输入手机号和密码”

7.连续失败 5 次后，账号锁定 30 分钟

期望输出：一份标准的测试用例列表，包含用例 ID、测试模块、用例标题、前置条件、操作步骤、预期结果、优先级等字段。

第二步：设计工作流框架

根据上述需求，设计如下工作流：

整个工作流框架由四个核心节点构成：

1.开始节点：定义输入参数，如用户上传的 PRD 文档或需求文本

2.文档解析节点：读取并解析上传的文档内容，将其转化为纯文本

3.LLM 处理节点：这是工作流的“大脑”，根据解析出的需求文本生成结构化测试用例

4.结束节点：将生成的测试用例以指定格式输出

第三步：创建工作流

实际操作步骤如下：

1.登录 Coze 平台（https://www.coze.cn），在个人空间中选择“工作流” → “创建工作流”

2.填写工作流名称和描述，点击“确认”进入编辑画布

3.插入开始节点，定义输入变量（如将变量名命名为 requirement_text，类型设置为文本或 Word 文档）

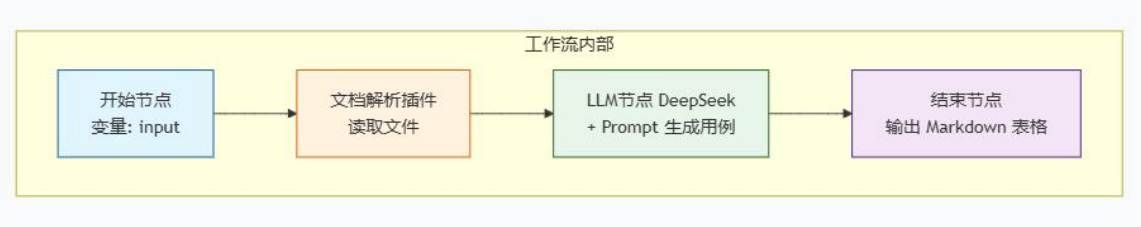
4.添加插件节点 → 选择“文件读取”插件，将开始节点与读取节点连接

5.添加大模型节点，选择合适的模型（如 DeepSeek-R1），配置提示词

6.将各节点首尾相连，形成完整的数据流向

7.点击“试运行”验证工作流，测试成功后发布

下图为工作流的详细结构示意图：



工作流的最大优势在于支持条件分支、并行处理和错误恢复。例如，在“登录功能”案例中，可以在 LLM 节点后添加一个“判断节点”，当需求中涉及“记住密码”“多因素认证”等扩展功能时，自动分支到对应的用例生成逻辑，实现更精准的覆盖。

三、四阶 AI 测试工作流（含可复用 Prompt）

第一阶段：需求解析 → 自动输出风险点和验收标准

工作流节点：知识库检索 → LLM 分析 → 结构化输出

在开始需求解析之前，需要先创建知识库。知识库是让 AI“学会”业务的核心。

知识库创建步骤：

1.点击“个人空间” → “知识库” → “创建知识库”

2.填写知识库名称（如“XX 项目需求库”）和描述

3.点击“新增单元”，上传以下文档：

o 产品需求文档（PRD）——PDF 或 Word 格式

o 接口文档（Swagger JSON）

o 测试标准文档（优先级定义、通过准则等）

o 历史用例库（Excel/CSV，作为 AI 的范例）

4.选择“自动采集”或“手动上传”，等待系统完成文档切片

5.Coze 会自动将文档分割成内容片段进行存储，并通过向量搜索检索最相关的内容

知识库结构说明：

• 知识库：一整套领域知识，是 Bot 加载的最小单位

• 单元：知识库的一部分，可上传的最小内容单位（一个 PDF 文件或一个网页）

• 分段：一个单元切分成多个分段，是模型查询的最小单位。分段内容的完整度和准确性直接影响 AI 回答的质量

工作流 Prompt 模板（可直接复制）：

你是测试架构师。分析以下需求，输出：

一、核心测试要点（3-5 个）

二、模糊点与风险点（表格：模糊描述 | 潜在风险 | 澄清问题）

三、验收标准（Given/When/Then 格式）

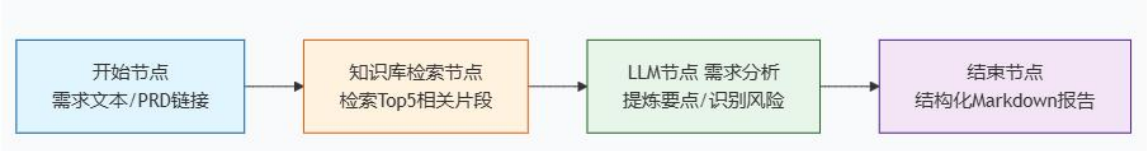
四、建议测试数据

不确定的信息不要编造，放入“模糊点”。

输入：PRD 片段或文档链接

输出：一份可直接评审的结构化报告，需求理解时间从 2 小时压缩到 10 分钟。

下图展示了第一阶段工作流的节点构成和数据流向：



实战输出示例（登录需求）：

一、核心测试要点

1.正常登录流程

2.密码错误锁定机制

3.输入校验（手机号格式、密码复杂度）

4.连续失败计数与锁定恢复

二、模糊点与风险点

模糊描述	潜在风险	澄清问题
“连续失败 5 次”是否包含空密码尝试？锁定 30 分钟后是否自动解锁？	计数规则不明确设计可能不同	空密码是否计入 5 次？是否需要手动解锁？

三、验收标准

- 场景 1：正确手机号+正确密码 → 跳转首页
- 场景 2：连续 5 次错误密码 → 账号锁定 30 分钟

四、建议测试数据

- 正确手机号：13812345678
- 正确密码：Abc12345
- 错误密码：Abc1234（7 位）、abc12345（全小写）等

第二阶段：用例生成 → 20+条用例/30 秒

工作流连接：将第一阶段的输出作为本工作流的输入。

Prompt 模板：

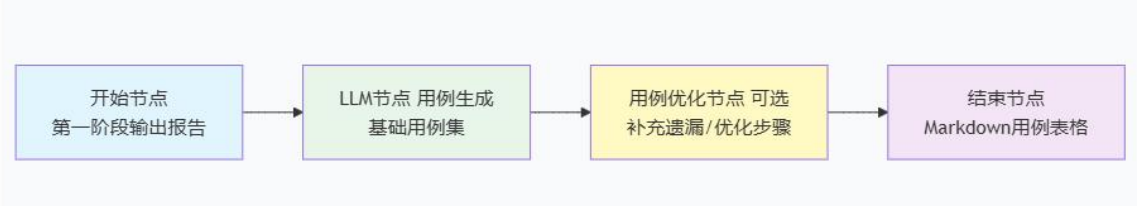
基于以下需求分析，生成测试用例。要求：

- 覆盖：正常、异常、边界、安全
- 技术：等价类、边界值、判定表
- 格式：Markdown 表格（编号|标题|前置条件|步骤|预期结果|优先级）

编号规则：TC-模块-类型-序号（FUNC/BOUND/EXCP/SEC）

优先级：P0 阻塞/P1 高/P2 中/P3 低

工作流图示：



实战效果（登录功能为例）：

编号	标题	步骤	预期结果	优先级
TC-LOGIN-FUNC-001	正常登录	手机+正确密码	跳转首页	P0
TC-LOGIN-BOUND-001	手机号边界-11 位	输入 11 位数字	成功	P1
TC-LOGIN-BOUND-002	密码边界-8 位	输入 8 位密码	成功	P1
TC-LOGIN-BOUND-003	密码边界-16 位	输入 16 位密码	成功	P1
TC-LOGIN-EXCP-001	手机号为空	手机号留空	提示“请输入手机号”	P1
TC-LOGIN-EXCP-002	密码为空	密码留空	提示“请输入密码”	P1
TC-LOGIN-EXCP-003	手机号格式错误-10 位	输入 10 位数字	提示格式错误	P1
TC-LOGIN-EXCP-004	手机号格式错误-含字母	输入含字母的手机号	提示格式错误	P2
TC-LOGIN-EXCP-005	连续 5 次错误	连输 5 次错误密码	锁定 30 分钟	P0
TC-LOGIN-EXCP-006	锁定期间输入正确密码	锁定后输入正确密码	仍提示锁定	P0
TC-LOGIN-SEC-001	SQL 注入	密码框输入 ' OR '1'='1	登录失败，无堆栈泄漏	P1
TC-LOGIN-SEC-002	XSS 攻击	用户名输入 <script> 标签	脚本不执行，正确转义	P2

产出：20+条用例，耗时 30 秒，人工审核 5 分钟。效率提升 8 倍。

第三阶段：执行辅助 → 数据生成 + 环境巡检

测试数据自动生成（独立工作流）：

工作流设计:

- 开始节点: 输入数据规格描述
- LLM 节点: 生成测试数据
- 代码节点: 格式校验（正则表达式）
- 结束节点: 输出 JSON 数组

输入示例: "10 个合法手机号 + 5 个非法手机号"

LLM 生成后校验: 正则 ^1[3-9]\d{9}$

输出: ["13812345678", "13987654321", ...]

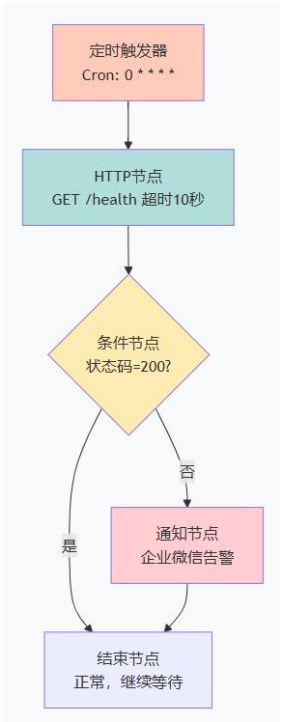
环境巡检 Bot（定时触发）:

工作流设计:

- 触发器: 定时触发（每小时）
- HTTP 节点: 调用服务健康检查接口 /health
- 代码节点: 解析响应码
- 条件节点: 非 200 则触发告警
- 通知节点: 企业微信发送告警

告警示例: "【告警】订单服务 /health 返回 503，请及时处理"

工作流图示:



效果：7×24 小时无人值守，MTTR 从 2 小时降到 5 分钟。

第四阶段：缺陷分析 → 自动提单 + 根因推断

自动提单工作流（输入失败日志 → 输出 Jira 工单）：

工作流设计：

1.触发节点：手动触发（粘贴失败日志）或 Webhook 触发（自动化测试失败时自动调用）

2.知识库检索节点：从历史缺陷库中检索类似问题

3.LLM 节点：分析失败信息，生成结构化缺陷报告

4.HTTP 请求节点：调用 Jira/TAPD 的 API 自动创建缺陷单

5.通知节点：发送企业微信消息

LLM Prompt：

根据以下失败信息，输出 JSON 格式缺陷报告：

```
{
  "title": "【模块】简述",
  "severity": "致命/严重/一般",
  "steps": "1...2...",
  "actual": "...",
  "expected": "...",
  "root_cause_guess": "基于日志的初步分析"
}
失败场景：{{scenario}}
日志：{{logs}}
```

根因分析增强（结合知识库）：

- 输入：NullPointerException at OrderService.calculatePrice:127

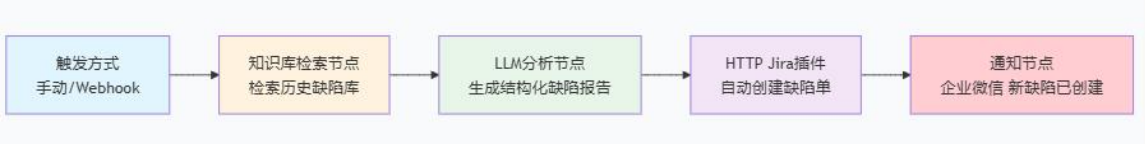
• 知识库检索：第 127 行调用了 couponService.getUserCoupons()

• 历史故障匹配：2025-01-15 类似问题，缓存返回 null

• 输出："根因假设：getUserCoupons 返回 null 而非空列表；修复建议：返回 Collections.emptyList()"

开发拿到的不是一句"报错了"，而是带排查路径和修复建议的报告。

工作流图示：



完整四阶工作流总览

四、落地配置（可直接抄）

4.1 自愈定位配置（coze-config.yaml）

```
element_strategy:
  primary: "data-test-id"
  fallback: ["id", "text", "xpath"]
self_healing:
  enabled: true
  max_attempts: 3
  healing_strategies: ["re_locate_by_text", "scroll_and_retry"]
```

效果：按钮 ID 从 login-btn 改成 submit-login，脚本不报错——自动通过文字"登录"重新定位。

4.2 Docker 私有化部署

字节跳动于 2025 年 7 月 26 日正式开源 Coze，包含 Coze Studio 可视化开发工具和 Coze Loop 运维管理系统两大核心组件，仅需 2 核 CPU+4GB 内存即可本地运行。

```
# 1. 克隆代码
git clone https://github.com/coze-dev/coze-studio.git
cd coze-studio/docker
# 2. 配置环境变量
cp .env.example .env
# 3. 配置模型（以 DeepSeek 为例）
# 编辑 backend/conf/model/deepseek-r1.yaml，填写 base_url 和 api_key
# 4. 一键启动
docker compose --profile '*' up -d
# 5. 访问控制台
# 浏览器打开 http://localhost:8888
```

避坑指南：

• 端口冲突：若提示端口被占用，使用 netstat -ano | findstr :3306 查找占用进程后结束，或修改 docker-compose.yml 中的端口号

• MySQL 启动失败：报错 MYSQL_USER cannot be "root" 时，删除系统环境变量中的 MYSQL_USER 和 MYSQL_PASSWORD

• Elasticsearch 启动失败：报错 exit 127 时，用 VSCode 打开 docker/volumes/elasticsearch/setup_es.sh，将右下角 CRLF 切换为 LF 后保存

4.3 CI/CD 集成（GitLab CI）

```
coze_test:
  image: coze/runner:latest
  script:
    - coze run --project web-e2e --env staging
  artifacts:
    paths:
      - reports/
```

每次代码提交自动触发测试套件，报告自动归档。

五、模型选型 & 避坑指南

任务	推荐模型	原因
用例生成	DeepSeek-R1	逻辑严谨，中文强
日志分析	GPT-4	推理深度好
简单问答	豆包 Pro	速度快、成本低

三个必须注意：

1.知识库质量决定一切——上传前清洗 PDF 页眉页脚，表格转 Markdown

2.Prompt 要强制格式——加一句“禁止输出非表格内容”

3.不确定就输出【需澄清】 ——杜绝幻觉

六、测试工程师的新姿势

阶段	AI 负责	你负责
需求评审	生成草案（节省 60%时间）	评审确认
用例设计	生成 80%基础用例	补充复杂场景
执行监控	7×24 巡检+自动提单	异常决策
报告输出	多模态报告+AI 总结	最终审核

核心转变：从“写脚本”到“训模型”，从“点工”到“AI 训练师”。

写在最后

Coze 不能完全替代测试工程师，但它可以把你的时间从重复劳动中解放出来，让你专注于策略、风险、深度分析。未来的测试工程师，不是被 AI 取代，而是被会用 AI 的同行取代。下一步：打开 Coze 控制台，创建一个 Bot，把上面的 Prompt 贴进去，上传一份 PRD——10 分钟后，你会回来感谢我。

如何判断发现的缺陷是普通问题还是特定的配置问题

◆作者：Bigder

引言：为什么区分“普通感冒”和“水土不服”至关重要？

想象一下，你是一个医生，有两个病人都发烧咳嗽。

• 病人 A：无论在北京、上海还是广州，只要天气一变冷就感冒。这是普通问题，就像软件的通用缺陷——无论在什么环境下，问题都会出现，根源在病人自身免疫力（软件代码逻辑）。

• 病人 B：只有去西藏这种高海拔地区才会头晕恶心、呼吸困难。一回到平原地区，症状就消失了。这是特定配置问题，就像软件的配置相关缺陷——问题只在特定的“环境”（高海拔/特定配置）下出现，根源在于环境与个体的交互。

作为医生，如果你误把病人 B 的高原反应当成普通感冒来治，不仅无效，还可能耽误病情。同样，在测试中，误判缺陷类型会导致：

• 开发人员浪费时间：在正确的环境下无法复现问题，像个无头苍蝇。

• 测试人员信誉受损：频繁提交无法复现的缺陷，会降低开发团队对你的信任。

• 项目风险增加：可能忽略了一个只在客户特定环境下才会触发的严重隐患。

接下来，我们就学习如何成为软件世界的“诊断专家”。

第一章：认识两位“病人”——普通缺陷 vs. 配置缺陷

我们先来给这两位“病人”画个像。

1. 普通缺陷（共性问题）

通俗理解：软件的“先天疾病”。无论放在什么环境下（就像无论吃什么食物），只要执行到有问题的代码，就一定会出错。

核心特征：稳定可复现。在不同的机器、不同的操作系统、不同的浏览器上，只要操作步骤一致，问题 100%会出现。

比喻：就像一台无论加什么牌子的汽油（配置），发动机都会在转速达到 5000 转时异响的汽车。问题出在发动机本身（代码）。

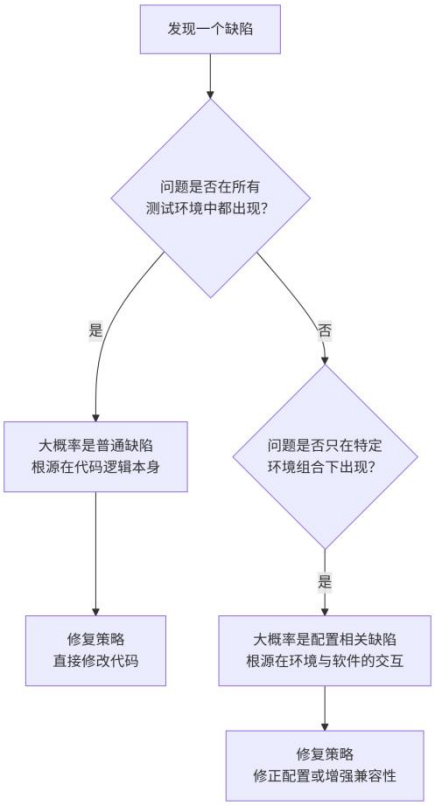
2. 配置相关缺陷（特定问题）

通俗理解：软件的“水土不服”。只有在特定的环境组合下（就像吃了特定的食物，如海鲜），才会出现症状。

核心特征：环境依赖性极强。问题的出现与特定的配置参数直接相关。

比喻：就像一台车，加了 92 号汽油（配置 A）一切正常，但加了 95 号汽油（配置 B）就会爆震。问题出在汽油和发动机的匹配上（环境与软件的交互）。

为了更直观地理解它们的区别，我们可以看下面这张图：



第二章：诊断“六脉神剑”——如何一步步判断缺陷类型

光有定义不够，我们需要一套可操作的诊断方法。这套方法就像六脉神剑，一招一式帮你锁定问题根源。

第一剑：环境隔离法（核心剑法）

这是最核心、最有效的一招。思路很简单：控制变量。

操作步骤：

1.记录现场：在发现缺陷的环境（我们称之为“问题环境”）中，详细记录所有配置信息：操作系统版本、浏览器类型和版本、屏幕分辨率、安装的软件、网络设置等。

2.搭建“无菌”环境：找一个全新的、干净的环境（比如一台刚装好系统的虚拟机），只安装最基本的软件运行环境。

3.复现测试：在干净环境中，严格按照问题环境中的操作步骤尝试复现缺陷。

• 如果复现：恭喜，你大概率发现了一个普通缺陷。因为连最干净的环境都出问题，说明是代码的“原罪”。

• 如果没复现：嫌疑指向了配置问题。进入下一步。

4.“投毒”测试：在干净环境的基础上，一步一步地将其“改造”成问题环境。比如，先更换浏览器、再调整分辨率、然后安装某个特定软件……每做一项改变，就测试一次。

5.锁定元凶：当进行到某一项改变后，缺陷突然出现了！那么，这项改变就是触发缺陷的“元凶”。比如，你发现只要把浏览器从 Chrome 100 升级到 Chrome 101，问题就出现，那么这就是一个针对 Chrome 101 的浏览器兼容性缺陷。

图文示意：

[问题环境] --(问题出现)--> [记录配置] --> [干净环境] --(问题不出现)--> [逐步添加配置] --(当添加配置 X 时问题出现)--> [找到元凶：配置 X]

第二剑：环境矩阵验证法

当第一剑无法快速执行时（比如搭建环境耗时），可以用这招辅助。

操作步骤：在现有的多个测试环境（比如环境 A、B、C、D）中，都用同样的步骤去测试。

结果分析：

o 所有环境都失败：强烈暗示是普通缺陷。

o 只有环境 A 失败，其他都成功：强烈暗示是配置缺陷，且与环境 A 的独有特性有关。

第三剑：配置回滚法

如果怀疑是某项配置的改动引入了问题，这招非常有效。

操作步骤：在问题环境中，将最近修改过的配置项逐一回滚到之前的版本，每回滚一个，测试一次。

结果分析：当回滚某个特定配置后，问题消失，那么这个配置就是罪魁祸首。

第四剑：日志与错误信息分析法

这是“听病人自述”，获取内部信息。

操作步骤：仔细查看软件运行时打印的日志（Log）和错误提示。配置问题的错误信息常常包含关键线索。

结果分析：

o 普通缺陷的错误信息可能比较“通用”，如 NullPointerException（空指针异常）。

o 配置缺陷的错误信息往往会直接指向某个环境因素，例如：

• Cannot find the specified module.（找不到指定模块） -> 可能缺少某个动态链接库（DLL 文件）。

• Failed to connect to database on host 192.168.1.100.（连接数据库失败） -> 数据

库地址或网络配置错误。

• This resource is not available in your region.（此资源在您所在区域不可用） -> 地区配置问题。

第五剑：网络与权限排查法

针对特定类型的配置问题。

网络问题：超时、连接被拒绝等，往往和防火墙、代理服务器、DNS 设置有关。

权限问题：“访问被拒绝”、“需要管理员权限”等，明显是运行软件的用户账户权限不足。

第六剑：与开发人员会诊

如果以上五剑都用完了，还是无法确定，不要自己硬扛。拿着你收集到的所有“诊断报告”（复现步骤、环境信息、日志截图）去找开发人员。清晰的信息能极大帮助他们快速定位问题。

第三章：实战演练——看图说话

让我们通过两个具体的例子来运用上面的“剑法”。

案例一：网页按钮点击无效

缺陷描述：在网站注册页面，点击“获取验证码”按钮无任何反应。

诊断过程：

1.环境矩阵验证：

• 测试环境 A（Win10 + Chrome 110）：点击无效。

• 测试环境 B（Win11 + Chrome 110）：点击无效。

• 测试环境 C（macOS + Safari 16）：点击无效。

• 测试环境 D（Win10 + Firefox 108）：点击有效！

2.初步判断：不是所有环境都失败，排除了普通缺陷的可能。问题可能与浏览器有关。

3.环境隔离法（精确定位）：

- 在环境 D（Firefox 正常）的基础上，只将浏览器换成 Chrome 110，问题出现。
- 再换回 Firefox，问题消失。
- 诊断结论：这是一个针对 Chrome 浏览器的兼容性缺陷（配置问题）。

4.提交缺陷报告：在缺陷管理系统中，清晰标注【浏览器相关】，并说明复现环境为 Chrome 浏览器，同时附上 Firefox 正常的截图作为对比。

案例二：软件启动时崩溃

缺陷描述：一款图片处理软件，在双击打开时立即闪退。

诊断过程：

1.查看错误信息：软件闪退前弹出一个错误框，显示 “The program can‘t start because VCRUNTIME140.dll is missing.”（程序无法启动，因为找不到 VCRUNTIME140.dll 文件）。

2.分析信息：错误信息直接指出，缺少一个名为 VCRUNTIME140.dll 的动态链接库文件。这是 Visual C++运行库的一部分。

3.环境验证：在其他几台电脑上测试，有的能打开，有的不能。不能打开的电脑都提示缺少类似的运行库文件。

4.诊断结论：这是一个典型的环境配置缺陷。根源在于目标电脑上没有安装软件所需的 Visual C++运行库。

5.修复方案：开发人员不需要修改代码，而是应该在软件安装包中捆绑这个运行库，或者在安装指南中明确告知用户需要提前安装。

第四章：如何清晰地报告两种缺陷

准确的诊断需要清晰的“病历”，以便“医生”（开发人员）对症下药。

报告普通缺陷的要点：

标题：清晰描述问题现象，如 【通用缺陷】用户管理页面，删除用户后列表未实时刷新。

内容：

o 复现步骤：详细的、可重复的操作步骤。

o 预期结果：应该发生什么。

o 实际结果：实际发生了什么。

o 环境信息：虽然问题是通用的，但仍需注明测试环境，但可加一句“此问题在其他多个环境中已验证存在”。

报告配置缺陷的要点：

标题：强烈建议在标题中注明配置相关性，如 【MacOS 特定】在 macOS Ventura 13.2 系统下，软件全屏时菜单栏无法自动隐藏。

内容：

o 复现步骤：同上。

o 预期/实际结果：同上。

o 关键：环境信息：必须极其详细地列出问题环境的所有配置。

问题环境：macOS Ventura 13.2, Safari 16.3, 分辨率 2560x1600。

对照环境：务必提供正常环境的信息！？如：在 Windows 11 + Chrome 110 环境下，此功能正常，并附上正常环境下的截图。有对比，才有说服力。

日志/错误信息：完整截图或复制错误日志。

总结

判断一个缺陷是普通问题还是配置问题，是一个从现象到本质的侦探过程。其核心心法可以概括为以下几点：

1.建立“环境”思维：永远不要孤立地看一个缺陷，要把它和它所在的环境绑定在一起思考。

2.信奉“控制变量”法则：通过搭建干净环境和逐步“污染”的方法，是锁定问题根源的最可靠手段。

3.善用“对比”武器：提供一个正常的环境作为对照，是证明问题为配置相关性的最强证据。

4.细节决定成败：错误信息、日志文件、版本号这些细节，往往是破案的关键线索。

成为一名优秀的测试工程师，不仅仅是找到 Bug，更是要精准地描述 Bug，为开发团队提供最清晰的“导航”。

拓展学习

[4] 【Python 自动化测试学习交流群】学习交流

咨询：微信 atstudy-js 备注：51

从零搭建企业级自动化测试流水线：Jenkins + Docker + Playwright 实战

◆ 作者：TestCraft

引言

自动化测试当然需要自动的运行，Jenkins 作为一款经典的持续集成工具仍被广泛应用于企业的自动化测试部署和执行。作为测试工程师掌握 Jenkins 的使用，能够有效提升测试效率从而扩大自动化测试的影响力！

本篇文章主打一个实战干货，直接带大家通过一个 demo project 学习使用 Jenkins 运行 Playwright 测试并生成报告的完整链路实现。

最终效果：我们将实现什么？

- **一键执行，全程自动化**

在 Jenkins 点击“构建”，流水线将自动拉取最新代码，在隔离的容器中准备好所有环境，并运行 Playwright 测试。

- **Allure 报告集成**

生成 Allure 测试报告并且可以通过 Jenkins 直接查看

- **邮件即时通知**

运行测试结束后，自动发送包含构建状态、测试信息和 Allure 报告链接的邮件。

准备工作

1.Docker：

本文将会使用 Docker 的方式安装 Jenkins，并且 playwright 测试也会在容器中执行

2.Playwright 测试项目：

• 准备一个可正常运行的 Python Playwright + Pytest 项目。

• 如果需要一个示例项目，可以从我的 GitHub 仓库获取：[https://github.com/bridgeshi85/playwright-pytest-allure-framework]

3.邮件发送服务：

准备一个 SMTP 邮箱账号（推荐 163 邮箱）

部署 Jenkins 服务

1. 准备 Jenkins 镜像

首先创建一个 Dockerfile，代码如下：

```
FROM jenkins/jenkins:lts

USER root

# -------- Install Docker CLI --------
RUN apt-get update && \
    apt-get install -y \
    ca-certificates \
    curl \
    gnupg \
    lsb-release && \
    install -m 0755 -d /etc/apt/keyrings && \
    curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /etc/apt/keyrings/docker.gpg && \
    chmod a+r /etc/apt/keyrings/docker.gpg && \
```

```
        echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/debian $(lsb_release -cs) stable" | tee /etc/apt/sources.list.d/docker.list > /dev/null && \
        apt-get update && \
        apt-get install -y docker-ce-cli docker-compose-plugin && \
        rm -rf /var/lib/apt/lists/*

# -------- Install Jenkins plugins (as root) --------
COPY plugins.txt /usr/share/jenkins/plugins.txt
RUN jenkins-plugin-cli --plugin-file /usr/share/jenkins/plugins.txt

EXPOSE 8080
```

重点说明：

Jenkins 需要通过 docker 命令启动容器，playwright 的测试会运行在宿主机的容器中(稍后会提到)，所以需要安装 docker cli 命令行工具

为了避免 Jenkins Pipeline 中执行 Docker 命令时出现 permission denied，使用了 root 用户。但是需注意，尽量避免在生产环境中这样操作

通过 jenkins-plugin-cli 预先安装了以下 plugin，具体文件在示例项目 jekins 目录下 plugins.txt 中， 内容如下：

```
workflow-aggregator:2.7
git:5.8.0
docker-workflow:1.28
credentials-binding:702.vfe613e537e88
ssh-agent:295.v9ca_a_1c7cc3a_a_
email-ext:2.102
allure-jenkins-plugin:2.33.0
```

准备完成后在 Dockerfile 和 plugins.txt 所在目录下执行以下命令来构建镜像：

```
docker build -t my-jenkins:latest .
```

2 启动 Jenkins

首先通过以下命令确认自己的 Docker Socket 使用的路径:

```
docker context ls
```

运行后可以看到以下内容:

```
NAME        DESCRIPTION        DOCKER ENDPOINT     ERROR
default              Current DOCKER_HOST based configuration     unix:///var/run/docker.sock
desktop-linux *     Docker Desktop
unix:///Users/xxxx/.docker/run/docker.sock
```

接着使用 docker 运行 Jenkins 容器

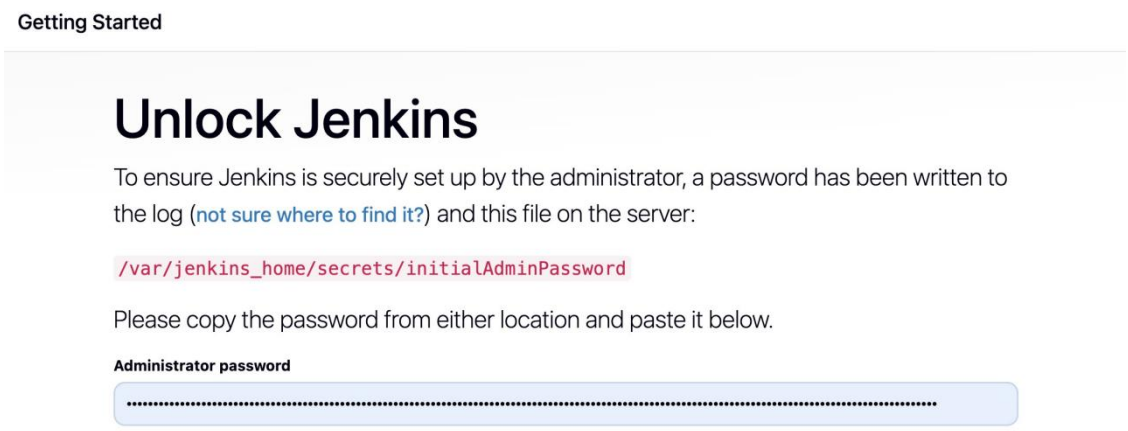
```
docker run -d \
  --name my-jenkins \
  -p 8080:8080 \
  -p 50000:50000 \
  -v jenkins_home:/var/jenkins_home \
  -v ~/.docker/run/docker.sock:/var/run/docker.sock \
  my-jenkins:latest
```

注意启动 Jenkins 容器时，需要将宿主机上 Docker Desktop 使用的 socket 路径（~/.docker/run/docker.sock）挂载到容器内的 /var/run/docker.sock。

它的作用是什么？ 默认情况下，容器内部无法直接运行底层的 Docker 命令。因此，我们将宿主机的 Docker Socket 目录映射到 Jenkins 容器中，使 Jenkins 能够调用宿主机的 Docker 启动测试容器。这种方式通常称为 Docker outside Docker (DooD) 模式。

3 初始化 Jenkins

Jenkins 容器成功运行后，打开浏览器访问 http://localhost:8080，将会看到 Jenkins 的“解锁”页面。



初始密码可以从容器日志中获取。因为之前启动的容器名为 my-jenkins，可以执行以下命令查看其启动日志：

```
docker logs my-jenkins
```

在日志输出中向下滚动，找到被 *** 包围的部分，其中的字符串就是密码。

```
[LF]>
[LF]> ************************************************************
[LF]> ************************************************************
[LF]> ************************************************************
[LF]>
[LF]> Jenkins initial setup is required. An admin user has been created and a password generated.
[LF]> Please use the following password to proceed to installation:
[LF]>
[LF]> d26b15f6f46f415ca77ee3f06faa1fbc
[LF]>
[LF]> This may also be found at: /var/jenkins_home/secrets/initialAdminPassword
[LF]>
[LF]> ************************************************************
[LF]> ************************************************************

[LF]> ************************************************************
```

将初始密码复制并粘贴到浏览器的输入框中，点击“继续”。随后按照引导安装推荐的插件，等待安装完成并设置好你的管理员账号后，就可以正式进入我们的 Jenkins 了！

创建 Pipeline Job 并运行测试

准备测试环境：启动被测应用与构建测试镜像

为了让测试顺利运行，本文准备了一个简单的 demo 前端项目（demo-frontend）作为被测系统。你可以通过仓库获取相关代码：[https://github.com/bridgeshi85/playwright-pytest-allure-framework] 接下来，我们将搭建测试环境并构建 Playwright 测试镜像。

构建并运行前端应用

为了让 Jenkins 中的测试容器能够访问到被测应用，我们首先需要创建一个 Docker 网络。

```
docker network create e2e-test
```

进入 demo-frontend?目录，构建应用镜像并启动容器。关键点在于使用?--network?参数将容器加入我们刚创建的 e2e-test 网络。

```
# 构建镜像
docker build -t demo-frontend:latest .
# 启动容器，并加入网络
docker run -d \
  --name demo-frontend \
  --network e2e-test \
  -p 3000:80 \
  demo-frontend:latest
```

构建测试代码运行镜像

在测试代码所在的目录（playwright-automation-test）下，打开终端执行以下命令。

请注意，镜像名称?playwright-test-agent:latest?必须与后续?Jenkinsfile?中指定的名称保持一致。

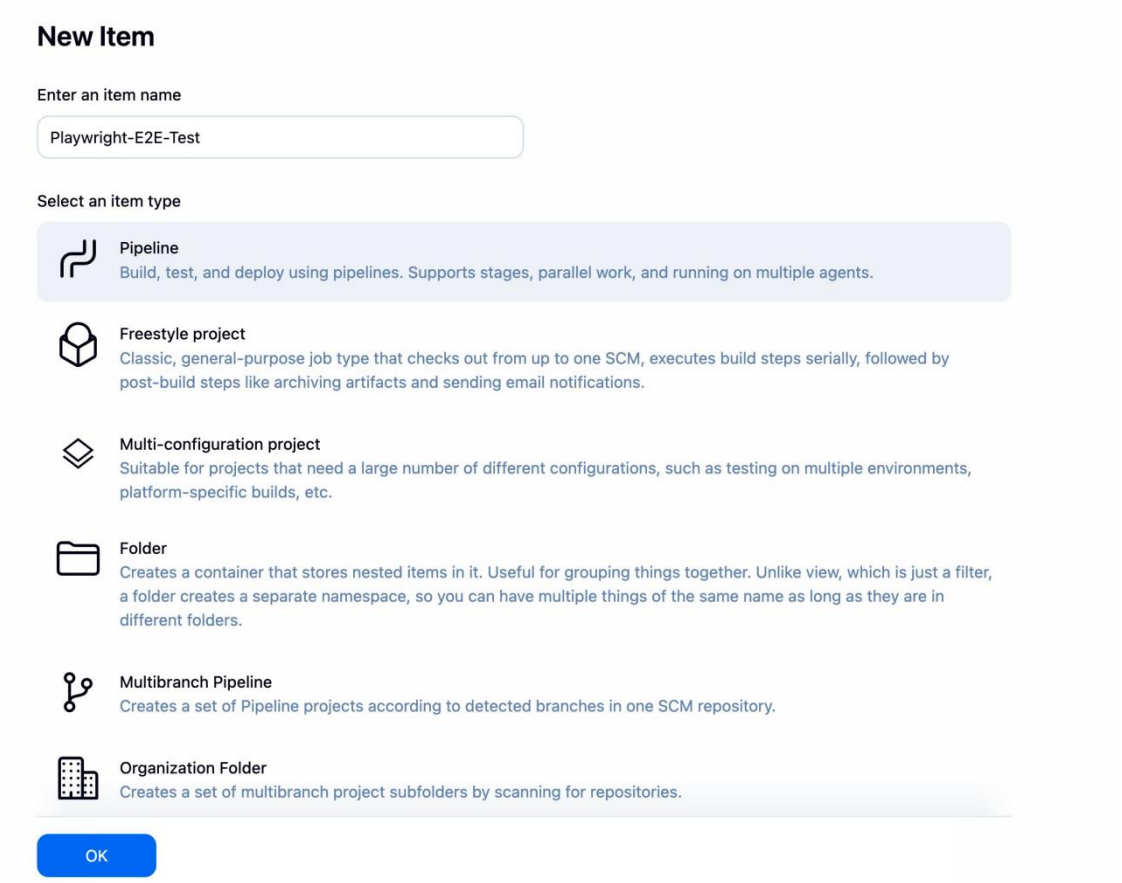
```
docker build -t playwright-test-agent:latest .
```

至此，测试所需的基础设施（测试应用和镜像）已准备完成。

第一步：在 Jenkins Pipeline 中运行测试

1.在 Jenkins 首页左侧点击 New Item (新建任务)。

2.输入任务名称（例如：Playwright-UI-E2E），选择 Pipeline (流水线)，点击 OK。



3.下拉至 Pipeline 区域，选择 Pipeline script 并将以下代码贴入

```
pipeline {
    agent any

    stages {
```

```
        stage('Checkout') {
            agent any
            steps {
                git branch: 'main',
                url: 'https://github.com/bridgeshi85/playwright-pytest-allure-framework'
            }
        }

        stage('Run UI Tests (Playwright Agent)') {
            agent {
                docker {
                    image 'playwright-test-agent:latest'
                    args '--network e2e-test'
                    reuseNode true
                }
            }
            steps {
                sh '''
                    set -euxo pipefail
                    echo "Running tests..."
                    cd playwright-automation-test
                    pytest tests/test_login.py --env=jenkins -s
                '''
            }
        }
    }
}
```

该 Pipeline 中定义了两个主要阶段：代码拉取和在执行测试。

阶段一：代码拉取 (stage('Checkout'))

从 GitHub 仓库拉取代码，指定分支为 main

阶段二：运行 UI 测试 (stage('Run UI Tests (Playwright Agent)'))

使用 docker agent， 指定镜像 playwright-test-agent:latest，并通过加入网络 e2e-test, 这样我们的测试容器就能找到 demo-frontend 应用。

在容器内执行以下命令 Shell 命令。

• set -euxo pipefail: 设置 Bash 选项，确保脚本严格执行、打印命令并在错误时退出。

• echo "Running tests...": 输出日志。

• cd playwright-automation-test: 切换到测试目录。

• pytest tests/test_login.py --env=jenkins -s: 通过 pytest 执行测试

备注：本示例仓库已设为公开访问，因此可以直接拉取。如果是私有仓库，请使用自己的 GitHub Token。

4. 运行测试

准备就绪后，点击左侧菜单栏中的立即构建（Build Now）运行测试，运行完成后可以在任务日志中看到如下内容：

```
$ docker top ed4258679c071e502f9f50e88311191e49a673cfa7286903ce3a30b1b6737389 -eo pid,comm
[Pipeline] {
[Pipeline] sh
+ set -euxo pipefail
+ echo Running tests...
Running tests...
+ cd playwright-automation-test
+ pytest tests/test_login.py --env=jenkins -s

tests/test_login.py::test_login_success
------------------------------- live log setup --------------------------------
08:30:55 [INFO] load config
08:30:55 [INFO] start test session: launching browser
08:30:55 [INFO] launch browser=chromium, headless=True, slowmo=0
08:30:55 [INFO] create result folder
08:30:55 [INFO] Test passed, no screenshot or page source need to be saved
------------------------------- live log call ---------------------------------
08:30:55 [INFO] Navigating to login page: http://demo-frontend
08:30:56 [INFO] Performing login action
08:30:56 [INFO] username entered: admin
08:30:56 [INFO] password entered
08:30:56 [INFO] login button clicked
08:30:56 [INFO] Checking welcome text
08:30:56 [INFO] Test passed, no screenshot or page source need to be saved
PASSED
------------------------------ live log teardown ------------------------------
08:30:56 [INFO] save video path to request.node
08:30:56 [INFO] close the page
08:30:56 [INFO] close the browser
08:30:56 [INFO] Test teardown, checking for video attachment
08:30:56 [INFO] Save video to Allure report
08:30:56 [INFO] Test passed, no screenshot or page source need to be saved
```

第二步：展示 Allure 报告

测试通过后会在工作区生成 Allure 报告文件，我们可以通过 Jenkins 的 Allure 插件直接在 Jenkins 中展示报告。

配置步骤

1.进入全局工具配置： 在 Jenkins 页面右上角点击系统配置，然后找到并点击 Tools (全局工具配置)

2.添加 Allure Commandline：

o 在配置页面中向下滚动，找到 Allure Commandline 区域，点击 Add Allure Commandline

o 输入别名，并勾选自动安装

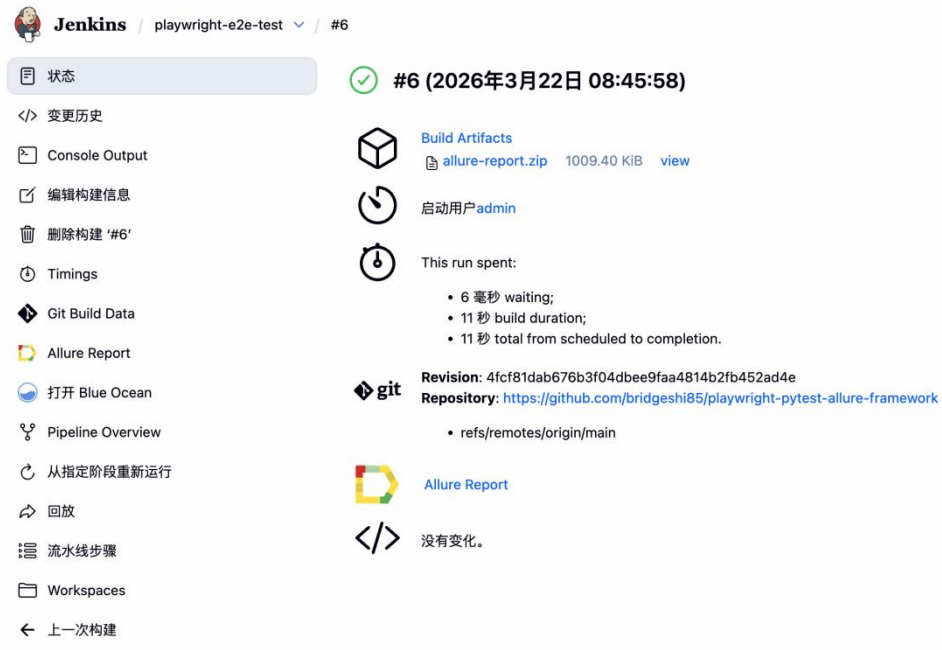
o 在下拉菜单中选择一个较新的版本，并选择默认从 Maven Central 下载即可。

3.在 Job 中配置 Allure

回到之前创建的 Pipeline Job，修改 Pipeline 脚本，在 stages 部分之后添加以下代码：

```
post {
    always {
        allure(
            includeProperties: false,
            jdk: '',
            resultPolicy: 'LEAVE_AS_IS',
            results: [[path: 'playwright-automation-test/reports/allure-results']]
        )
    }
}
```

再次 Build 之后，我们可以在构建页面中看到 Allure Report 的链接



第三步：发送邮件报告

最后一步是将测试结果和报告链接及时推送给团队邮箱，便于测试成果的共享与同步。Jenkins 提供了邮件插件 emailext（Extended E-mail Notification）来帮助我们实现。

Jenkins Email 插件配置步骤

首先，需要准备一个已开通 SMTP 的邮箱，这里以 163 邮箱为例（SMTP 设置方法不再赘述）。邮箱准备好后，回到 Jenkins 进行插件配置：

1.添加邮箱 Credentials：

• 进入 Manage Jenkins（系统管理） → Security → Credentials

• 点击 + Add Credentials，选择 Username with password

• Username 填写邮箱地址，Password 填写 SMTP 授权码（不是邮箱登录密码）

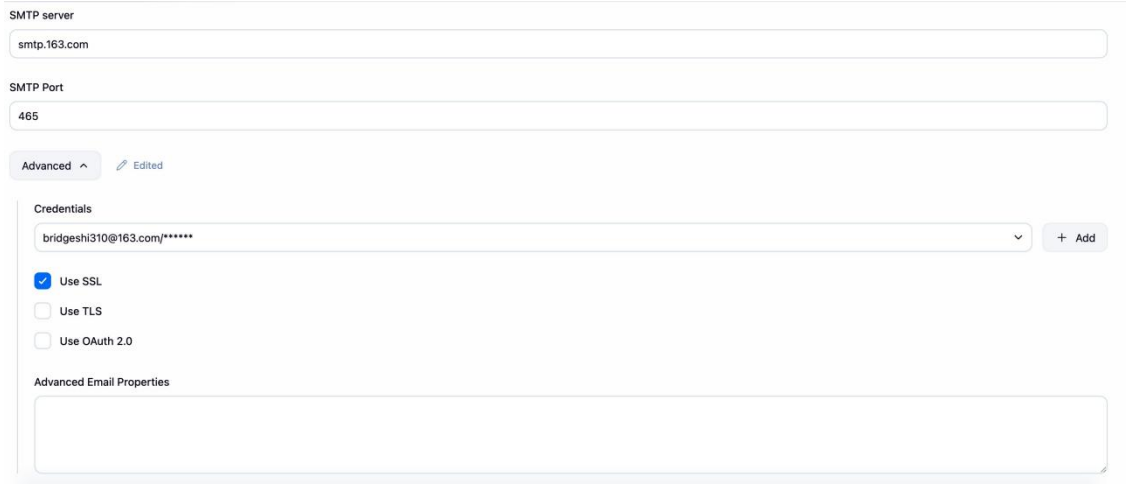
2.配置插件：

• 进入 Manage Jenkins（系统管理） → System

• 向下滚动找到 Extended E-mail Notification（注意：页面中还有一个普通的 “E-mail Notification”，实测两者任选其一即可，也可都填写）

3.填写参数：

- SMTP server： 输入 smtp.163.com
- SMTP port： 输入 465
- Use SSL： 勾选 ☑
- Credentials: 选择之前创建的 163 邮箱凭证
- Default user e-mail suffix： 输入 @163.com



4.配置 Jenkins 默认发送的邮箱地址：

- 进入 Manage Jenkins (系统管理) -> System
- System Admin e-mail address 中填写与第一步相同的 Username (由于 163 安全设置，SMTP 发送时的邮箱地址必须一致)

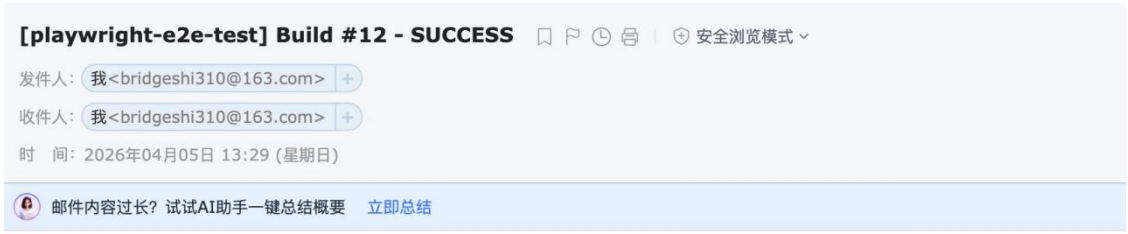
Pipeline 中添加邮件通知步骤

配置完成后，我们回到之前创建的 Pipeline，在 allure step 后添加以下步骤：

```
emailext(
    subject: "[${env.JOB_NAME}] Build #${env.BUILD_NUMBER} - ${currentBuild.currentResult}",
    to: "${params.EMAIL_RECIPIENTS}",
    mimeType: 'text/html',
    body: """
    <p><b>Build Status:</b> ${currentBuild.currentResult}</p>
```

```
        <p><b>Job:</b> ${env.JOB_NAME}</p>
        <p><b>Build Number:</b> ${env.BUILD_NUMBER}</p>
        <p><b>Build URL:</b>
        <a href="${env.BUILD_URL}">${env.BUILD_URL}</a>
        </p>
        <p><b>Allure Report:</b>
        <a href="${env.BUILD_URL}allure/">View Report</a>
        </p>
    """
    )
```

Build 完成后，Jenkins 就会自动将这封包含了测试结果的邮件，发送到我们在触发时填写的收件人邮箱里：



Build Status: SUCCESS

Job: playwright-e2e-test

Build Number: 12

Build URL: http://localhost:8080/job/playwright-e2e-test/12/

Allure Report: View Report

我们将拉取代码、Docker 容器隔离执行测试、生成 Allure 报告和发送邮件通知的所有步骤完美结合起来了。以下就是完整的 Jenkinsfile 代码：

```
pipeline {
    // 全局基础代理设定
    agent any
    // 参数化构建，允许每次运行前动态指定收件人邮箱
    parameters {
        string(
```

```
                name: 'EMAIL_RECIPIENTS',
                defaultValue: 'your_email@domain.com', // ?? 请将此处替换为你的默认收件邮箱
                description: 'Comma-separated list of email addresses to receive build notifications'
            )
        }
        stages {
            // 阶段 1：拉取最新测试代码
            stage('Checkout') {
                steps {
                    git branch: 'main',
                    url: 'https://github.com/bridgeshi85/playwright-pytest-allure-framework' //替换成自己的 git 仓库
                }
            }
            // 阶段 2：在隔离的 Docker 容器中运行测试
            stage('Run UI Tests (Playwright Agent)') {
                agent {
                    docker {
                        image 'playwright-test-agent:latest'
                        args '--network e2e-test' // 接入被测系统所在的专属网络
                        reuseNode true   // 复用工作空间，确保能读到上一阶段拉取的代码
                    }
                }
                steps {
                    sh '''
                        set -euxo pipefail
                        echo "Running tests..."
                        // 进入测试脚本所在目录并执行 Pytest
                        cd playwright-automation-test
                        pytest tests/test_login.py --env=jenkins -s
                    '''
                }
            }
```

```
        }
        // 无论成功或失败，雷打不动的收尾工作
        post {
            always {
                // 1. 收集测试数据并生成 Allure 报告
                allure(
                    includeProperties: false,
                    jdk: '',
                    resultPolicy: 'LEAVE_AS_IS',
                    results: [[path: 'playwright-automation-test/reports/allure-results']]
                )

                // 2. 发送包含了报告链接的 HTML 邮件
                emailext(
                    subject: "[${env.JOB_NAME}] Build #${env.BUILD_NUMBER} -
${currentBuild.currentResult}",
                    to: "${params.EMAIL_RECIPIENTS}",
                    mimeType: 'text/html',
                    body: """
                        <p><b>Build Status:</b> ${currentBuild.currentResult}</p>
                        <p><b>Job:</b> ${env.JOB_NAME}</p>
                        <p><b>Build Number:</b> ${env.BUILD_NUMBER}</p>
                        <p><b>Build URL:</b>
                            <a href="${env.BUILD_URL}">${env.BUILD_URL}</a>
                        </p>
                        <p><b>Allure Report:</b>
                            <a href="${env.BUILD_URL}allure/">View Report</a>
                        </p>
                    """
                )
            }
        }
    }
```

写在最后

在真实的工程实践中，写出优秀的代码只是第一步，让自动化测试在持续集成中不断运行，才是将其价值最大化的途径。 因此，掌握完整的 Jenkins 自动化链路，不仅能成倍提升团队的测试效能，更是测试工程师扩大技术影响力、推动质量保障体系升级的关键。希望这篇实战指南能帮助你顺利在团队中落地自动化基础设施，让测试代码真正产生价值！

拓展学习

[5] 领取【企业级全链路测试实战】项目包

咨询：微信 atstudy-js 备注：51

一张截图，差点让整个系统 “瘫痪”

◆作者：测测小仙女儿

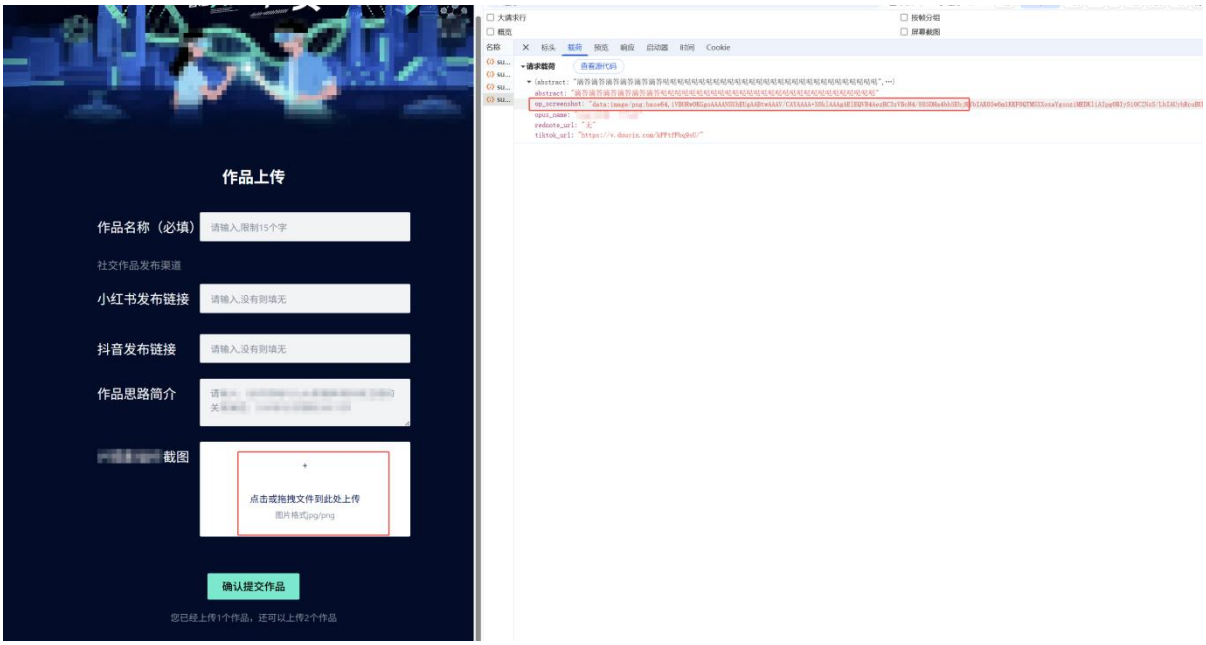
开场：你以为只是上传了一张图？

你有没有想过，一张小小的截图，竟然能：

- 让接口响应时间从 200ms 爆升到 15 秒以上
- 把 Postman 搞到直接崩溃
- 浏览器调试栏卡死、抓包工具提示 “无法加载响应数据”
- 运营后台彻底变 “慢动作播放”

这不是什么极端场景，而是我们团队最近一次运营活动上线后的真实写照。

而这一切的源头，就是这张图 ——



左边是用户上传截图的界面，右边是开发者工具中抓取的请求体，其中 op_screenshot 字段是一个长达 10 万 + 字符 的 Base64 编码字符串。

看到这里，你是不是也觉得有点熟悉？

“哦，不就是传了个图片嘛？”

但你知道吗？这短短一句话，几乎要了系统的命。

第一幕：一场 “无声” 的爆炸

问题爆发

某天下午，运营同学反馈：“我刚提交完作品，去后台查看列表，页面一直转圈…… 点进去都打不开。”

我们立刻登录后台，打开接口调试，结果：

- 接口返回状态码 200？
- 响应时间：18.7 秒！
- 响应体大小：3.2MB
- Postman 直接卡死，强制关闭才恢复

更离谱的是，用 Fiddler 抓包，发现响应数据根本加载不出来，提示：“无法解析响应内容”。

那一刻，我们意识到：这不是 bug，这是 “性能灾难”。

第二幕：真相浮出水面

我们迅速排查，最终锁定罪魁祸首 ——

错误设计流程（错误示范）

```
{
  "op_screenshot": "...（超长）"
}
```

- 用户上传图片 → 前端自动转成 Base64 → 发送给后端

• 后端不做任何处理 → 直接存入数据库（字符串字段）

• 列表接口查询所有记录 → 返回完整数据 → 包括每一个用户的 Base64 图片

这意味着：每条记录都带了一个几 MB 的 Base64 字符串！

假设列表有 100 条数据，每条含一张 1MB 的图片，那么：

总响应体积 ≈ 100 × 1.3MB ≈ 130MB

这已经不是 “慢” 了，这是在给服务器和客户端 “送 guan 材”。

第三幕：为什么没人提前发现？

日常测试中常见的盲区

测试环节	是否覆盖	问题所在
功能测试	✓ 是	只验证“能不能上传”，没测“多大能传”
接口测试	✗ 否	用小图测试，没模拟真实场景
性能测试	✗ 否	没考虑“列表页批量返回大字段”
安全测试	✗ 否	忽略了 Base64 超大字符串的潜在风险

这就是典型的 “功能正常，但体验致命” 陷阱。

我们在测试时，通常会上传一张小图（比如 100KB），然后确认接口返回成功，就认为 “没问题”。

殊不知，真实用户可能上传的是 4K 截图、屏幕录制截图、甚至整屏截屏！

第四幕：如何 “救活” 系统？

我们紧急启动修复方案，分三步走：

步骤一：前端优化 —— 不再内联 Base64

```
// × 错误做法
const base64 = await convertToBase64(file);
await api.submit({ op_screenshot: base64 });
// √ 正确做法
const formData = new FormData();
formData.append('file', file);
const res = await api.uploadImage(formData); // 返回图片 URL
await api.submit({ op_screenshot_url: res.data.url });
```

将图片上传到 OSS 或 CDN，只返回 URL，避免传输原始数据。

细节探讨：

FormData 对象：相比直接将文件转换为 Base64 字符串，使用 FormData 更加高效，它允许你以二进制形式发送文件，减少了不必要的编码开销。

异步上传：利用现代浏览器支持的异步上传机制，可以显著提升用户体验，减少等待时间。

URL 返回：通过返回图片的公网访问地址而不是 Base64 字符串，不仅减小了响应体大小，还使得图片能够被缓存，进一步提升了性能。

步骤二：后端改造 —— 分离存储，按需加载

```
// 修改前
{
  "op_screenshot": "base64...（超长）"
}
// 修改后
{
  "op_screenshot_url": "https://cdn.example.com/images/abc123.png",
  "op_screenshot_thumb": "https://cdn.example.com/thumb/abc123.jpg" // 缩略图
}
```

- 原始图存 CDN，仅用于下载或预览
- 列表页只返回缩略图 URL，减少响应体积
- 点击查看详情时，再加载原图

细节探讨：

缩略图生成：为了提升列表页的加载速度，我们可以在后端对上传的图片进行缩略图生成，并将缩略图单独存储。这样，在展示大量图片时，只需要加载较小的缩略图，极大地提高了页面加载效率。

CDN 集成：通过集成内容分发网络（CDN），我们可以确保图片资源的快速加载，无论用户位于世界的哪个角落。同时，CDN 还提供了强大的缓存机制，减少了服务器的压力。

步骤三：服务端压缩 + 格式优化

使用 sharp 自动处理上传图片：

```
const sharp = require('sharp');
async function processImage(file) {
  const thumbnail = await sharp(file.buffer)
    .resize(200, 200)
    .webp({ quality: 80 })
    .toBuffer();
  const original = await sharp(file.buffer)
    .webp({ quality: 70 })
    .toBuffer();
  return {
    thumb: thumbnail,
    original: original
  };
}
```

WebP 比 PNG 小 25%~30%，非常适合网络传输。

细节探讨：

WebP 格式的优势：相较于传统的 JPEG 和 PNG 格式，WebP 在保持图像质量的同时，能够显著减少文件大小。这对于提升网站加载速度和节省带宽具有重要意义。

动态调整质量：根据不同的应用场景，我们可以灵活地调整图片的质量设置。例如，在生成缩略图时，可以适当提高质量，而在生成原始图时，则可以降低质量以换取更小的文件大小。

第五幕：测试警醒点（测试小伙伴都认真看！）

这次事件让我们深刻反思：测试不能只看 “功能对不对”，更要关注 “数据量够不够”。

以下是针对 “上传图片类功能” 的 5 大测试警醒点：

警醒点 1：不要只测小图，必须测 “极限图”

测试用例中加入：

- 10MB 图片（模拟用户误传）
- 4K 屏幕截图（常见于 PC 端）
- 透明背景 PNG（容易导致 Base64 更大）

建议：设置上传文件大小限制（如 ≤5MB），并提示用户。

细节探讨：

文件大小限制：为了避免用户上传过大的文件导致系统性能下降，我们应该在前端和后端都设置合理的文件大小限制。此外，还需要向用户提供清晰的提示信息，告知其上传文件的最大尺寸限制。

格式检测：除了大小限制外，还应该检查上传文件的格式是否符合要求。对于不支

持的格式，应当及时给出错误提示，防止意外情况的发生。

警醒点 2：检查接口是否返回大字段

- 使用 Postman / Swagger / Charles 模拟大量数据查询
- 查看响应体大小，若超过 1MB，立即报警
- 特别注意：列表接口 ≠ 单个详情接口

建议：列表页只返回必要字段，图片用 URL 替代。

细节探讨：

响应体优化：在设计 API 时，应当遵循 “最小化原则”，即只返回客户端真正需要的数据。特别是在涉及大量数据的场景下，更要注意避免一次性返回过多无用信息。

懒加载机制：为了进一步提升用户体验，我们可以引入懒加载机制。也就是说，只有当用户滚动到特定位置时，才会触发相应的数据加载操作，从而避免了初始加载时的性能瓶颈。

警醒点 3：警惕 Base64 内联陷阱

所有涉及图片上传的功能，优先问一句：

“你是把图片转成 Base64 还是上传文件？”

若前端使用 Base64，请务必：

- 限制最大长度（如 ≤2MB）
- 提供压缩选项
- 直接改用文件上传 + CDN 回调

细节探讨：

Base64 的局限性：虽然 Base64 编码在某些场景下非常方便，但它并不是解决所有图片上传问题的最佳选择。特别是当涉及到大文件时，Base64 编码会导致数据量显著增加，进而影响传输效率。

替代方案：除了直接上传文件之外，还可以考虑使用其他更加高效的传输方式，例如流式传输。这种方式特别适合于处理超大文件，因为它允许我们将文件分块传输，减轻了服务器和客户端的负担。

警醒点 4：CDN 和缓存机制是否到位？

- 图片一旦上传，是否被 CDN 加速？
- 是否支持浏览器缓存？
- 是否允许懒加载？

建议：所有图片资源都走 CDN，避免服务器直传。

细节探讨：

CDN 的作用：内容分发网络（CDN）通过在全球范围内分布多个节点，使得用户可以从距离自己最近的服务器获取所需资源，从而大大缩短了加载时间。因此，在涉及图片等静态资源时，一定要充分利用 CDN 的优势。

缓存策略：除了 CDN 加速之外，我们还可以结合浏览器缓存机制来进一步提升性能。通过设置适当的缓存头部信息，可以让浏览器在一段时间内无需重新请求相同的资源，从而降低了服务器的压力。

警醒点 5：压力测试不能少

- 模拟 1000 用户同时提交作品
- 查看接口响应时间、内存占用、GC 频率
- 关注是否有 OOM（内存溢出）风险

建议：使用 JMeter 做压力测试，提前暴露隐患。

细节探讨：

压力测试的重要性：在实际项目中，往往会出现单个用户操作看似正常，但在高并发情况下却暴露出各种问题的情况。因此，进行充分的压力测试是非常必要的。通过模拟大量的并发请求，我们可以发现潜在的性能瓶颈，并采取相应的优化措施。

性能监控：除了定期进行压力测试之外，我们还应该建立完善的性能监控体系。通过实时监测服务器的各项指标（如 CPU 使用率、内存占用等），可以及时发现问题并做出调整，确保系统的稳定运行。

最后我想说的是：一张图，教会我们敬畏细节

这张看似普通的截图，像一颗 “定时炸弹”，藏在功能的背后，静静等待引爆。

它告诉我们：

功能正常 ≠ 系统稳定，用户体验 ≠ 代码正确。

作为测试人员，我们不仅要验证 “能不能用”，更要思考：

- 数据会不会太大？
- 接口会不会崩溃？
- 用户会不会卡死？

有时候，最致命的问题，往往来自最不起眼的地方。下次当你看到某个接口返回一个 “很长的字符串”，请停下来问一句：

“这真的是必要的吗？”

也许，那正是下一个 “血案” 的开始。

刚入职场一两年时，甚至有一个老员工拉着我告诉我，测试不用那么较真，随便测下就好了，然后没过多久，她就被开除了，再后来，她貌似没再踏入测试行业。作为一个在测试圈摸爬滚打十多年的老阿姨，跟你们掏心窝子说句实在话：做测试这活儿，真不是图省事就能混过去的。别怕细碎、别嫌麻烦，一个小小的图片上传，就能让整个系

统瘫成一滩泥——就像咱们这篇博文写的，那你想想那些复杂点的场景，如果随便测测，线上发生事故后果不堪设想。。。测个功能就"差不多了"，心想"反正没人知道"，测试用例随便糊弄两下就交差，这种心思真的不能有，只要开始，你就会越来越懒，越来越投机取巧，你的思维也会跟着固化，迟早得被淘汰。你想啊，姐妹，咱摸鱼也是一天，好好对待也是一天，多花两分钟仔细过一遍，多动脑子想一想，可能就避免了一场"血案"。咱们干的是别人的活儿，但混的是自己的日子。

如果你正在做类似功能，请转发给你的开发同事，一起避免 “一张图毁掉一个项目” 的悲剧。

拓展学习

[6] 【软件测试实战特训营】学习交流

咨询：微信 atstudy-js 备注：51

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