微信扫一扫关注我们

投稿邮箱：editor@51testing.com

AI 自动化测试工具之 Midscene.js

◆ 作者：Carl_奕然

1、引言

一转眼，已经快一年没有在我们 51Testing 网更文了，有很多测试小伙伴私信我，问：咋这么长时间没在 51Testing 更新文章了，这干啥去了。

嗯，确实值得”回味”一下，这一年在做什么事情，做了哪些事情，成了哪些事情。

不仅团队在布局大模型、智能体领域，还需要对当前业界 AI 测试工具的探索和二次开发，以便更适应企业的业务。 这一忙就是小一年。

这时候你是不是就好奇地问道：那这一次，你会给我们聊哪些最新的测试工具呢，哪些测试方法，以及如何在场景中应用呢？

嗯，没错的，这次会分系列文章来聊，除了去年的《大模型工程服务测试》系列专栏之外，这次依然会聊 AI：即 AI 测试工具， 智能体测试系列专栏等等。

所以，是不是很期待哦， 我也是蛮期待的。

好了，咱闲言少叙，书说简短，开启今天的 AI 工具探索之：Midscene.js 。

2、Midscene.js 简介

2.1 什么是 Midscene.js

提到 Midscene.js，是不是很多小伙伴很陌生，它就仅仅是一个自动化工具么？ 它的神奇之处是什么？

别着急，跟着我的思路，咱逐层揭开 Midscene.js 的面纱。

Midscene.js 是由字节跳动 Web Infra 团队开源的一款基于 AI 的 UI 自动化工具，支持通过自然语言描述实现对网页、移动端、自动化测试和脚本编写等操作。

Midscene.js 兼容 JavaScript SDK、YAML 脚本和 Chrome 插件，可深度集成 Playwright、Puppeteer 等主流框架，实现智能、高效、低门槛的自动化体验。

2.2 Midscene.js 与其他工具差异

说到 Midscene.js，就不得不提另一个工具 Browser Use。这里，我通过技术栈与生态、核心功能与 API 设计、多模态与 AI 能力、使用门槛等来分别对比这两个工具的差异。

技术栈与生态

维度	Browser Use	Midscene.js
主要编程语言	Python（>=3.11）	JavaScript/TypeScript
底层浏览器自动化	Playwright（官方推荐）	Playwright / Puppeteer
AI 模型支持	支持 OpenAI、Claude、DeepSeek、Ollama 等所有 LangChain	默认 GPT-4o，支持 Qwen2.5-VL、UI-TARS、私有化部署等
依赖环境	Python 环境 + Playwright 浏览器驱动	Node.js + npm

通过上述的对比，可以更直观的看到：

Browser Use 属于 Python 生态，AI 模型适配性极广，适合后端/算法团队；

Midscene.js 属于 JS/TS 生态，深度集成前端工程链，适合前端测试和自动化项目。

核心功能与 API 设计

维度	Browser Use	Midscene.js
动作	Agent.run(task="…")，自动规划并执行	aiAction/ai（自然语言动作）、aiTap/aiInput（即时操作）
数据提取	支持自定义动作，需手动解析页面数据	aiQuery 可直接指定 JSON 结构返回
断言	无专用断言 API，需自行解析结果	aiAssert 自然语言断言
多标签管理	原生支持多标签页管理与并行任务	无原生多标签管理，需 Playwright/Puppeteer 代码扩展
可视化报告	依赖 Gradio UI 或自行扩展	自动生成可视化报告，含动画回放、AI 决策详情

同样，Midscene.js 的 aiQuery/aiAssert 等 API 设计更适合测试和自动化，Browser Use 适合灵活的任务规划和数据采集。

多模态与 AI 能力

维度	Browser Use	Midscene.js
视觉理解	use_vision 参数控制，支持截图+DOM 融合	多模态大模型，默认开启视觉理解，支持 UI-TARS/Qwen2.5-VL
模型私有化	支持 Ollama、本地模型 29	支持私有化部署，适配国产模型，数据安全可控
动态适配	自我纠正、自动重试、DOM 变化适应	AI 智能定位、Deep Think 模式、支持动态 UI 变化

两者均支持视觉理解和模型私有化，Midscene.js 在多模态 UI 理解和动态适配上更突出，Browser Use 在模型生态和自定义能力上更强。

应用场景

维度	Browser Use	Midscene.js
自动化测试	需自行封装测试流程，适合数据采集、爬虫、智能代理	UI 自动化测试设计，支持断言、数据提取、CI/CD
数据抓取	适合复杂网页、动态内容抓取，支持自定义动作	aiQuery 轻松提取 JSON，适合商品列表、表格抓取
RPA/智能代理	适合复杂任务规划、多标签并行、自定义扩展	适合前端流程、表单填写、页面交互自动化

在应用场景上， Browser Use 更适合爬虫、智能代理等复杂场景，Midscene.js 更适合前端测试、轻量自动化和 CI/CD 集成。

使用门槛

维度	Browser Use	Midscene.js
安装	Pip install browser-use + playwright install	npm install midscene.js
零代码/插件	无官方插件，可本地/云端运行，WebUI 可选	Chrome 插件，一键体验，无需代码环境
配置复杂度	需配置 LLM API Key，环境变量管理	支持 Chrome 插件粘贴 Key，YAML 配置等轻量方式

在使用门槛， Midscene.js 的 Chrome 插件和 YAML 脚本大幅降低上手门槛，Browser Use 需 Python 环境和 LangChain 知识，但功能扩展更自由。

所以，我相信除大部分测试团队在选择工具侧时候， 除了考察工具的应用能力，可拓展性，更重要的是看这个工具的使用门槛和上手速度。

综上所述，我相信，大家对 Browser Use 和 Midscen.js 这两款工具都有了一些初步的了解。

那是不是有同学会问了， 既然要聊 Midscen.js，不会仅仅让我们了解这两款工具的差异点吧，还有其他的让我们了解的么？

那是当然，关于 Midscen.js 的安装、模型配置等，我们继续往下聊。

3、Midscene.js 安装

3.1 手动下载安装

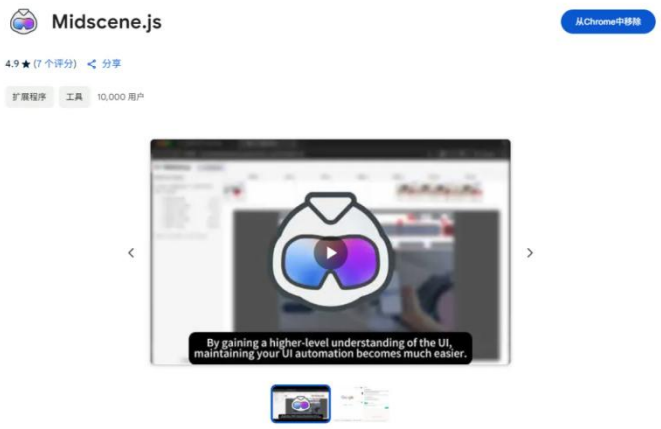
1、在 Github 下载 release 压缩包，路径：

https://github.com/web-infra-dev/midscene/releases

2、本地解压后，添加到 chrome 或 edage 浏览器的拓展里面。

3.2 在 chrome 浏览器下载插件安装

相对于通过 Github 下载安装包，这个方法更便捷， 直接在 chrome 应用商店搜索“Midscene.js”，找到后直接安装即可。



3.3 命令行安装

3.3.1 Npm 安装

如果要通过 sdk 安装，可以直接执行下面这个命令：

```
npm install midscene.js
```

同样，没有什么难度， 这里就不过多赘述。

3.3.2 源码安装

这种方式安装是最常见的，并且可以本地部署，也可以对源码进行二次优化，具体执行如下：

```
git clone https://github.com/web-infra-dev/midscene.git
cd midscene
```

4、Midscene.js 模型配置

上述方法选择其中一个安装完成后，我们就需要配置模型了，到现在，Midscene.js 已经可以适配视觉语言模型和 LLM 模型 ，这对我们来说非常友好，这或许也是为啥迅速在测试领域“受宠”的原因吧。

4.1 视觉语言模型配置

关于 Midscene.js 对视觉语言模型适配有：

千问 VL 、豆包系列视觉语言模型、Gemini-2.5-Pro 、UI-TARS

这里，我主要介绍豆包系列视觉语言模型和 千问模型的配置方法。

豆包系列视觉语言模型，配置如下：

```
OPENAI_BASE_URL="https://ark.cn-beijing.volces.com/api/v3"? #模型 url
OPENAI_API_KEY="...." #自己的 API 秘钥，
MIDSCENE_MODEL_NAME="ep-..." # 推理模型接入点 ID 或模型名称
MIDSCENE_USE_DOUBAO_VISION=1
```

使用 Qwen3-VL 模型的配置

```
OPENAI_BASE_URL="https://dashscope.aliyuncs.com/compatible-mode/v1"
OPENAI_API_KEY="....."
MIDSCENE_MODEL_NAME="qwen3-vl-plus"
MIDSCENE_USE_QWEN3_VL=1 # 注意，这个参数与 MIDSCENE_USE_QWEN_VL 不能同时使用
```

其他的模型配置跟千问模型和豆包视觉模型类似， 就不在这里一一展示了。

关于通用配置说明，如下：

名称	描述
OPENAI_API_KEY	必选项，没有 API_KEY 则模型服务调用与服务
OPENAI_BASE_URL	可选，API 的接入 URL。常用于切换到其他模型服务
MIDSCENE_MODEL_NAME	可选，指定一个不同的模型名称 (默认是 gpt-4o)， 常用于切换到其他模型服务

4.2 LLM 配置

关于 LLM， 我不会做过多介绍， 因为 LLM 会在 Midscene.js v1.05 之后版本不再支持。这里，就了解一下即可。

```
OPENAI_API_KEY="***"
OPENAI_BASE_URL="***"
MIDSCENE_MODEL_NAME="gpt-4o "
```

到这里，关于 Midscene.js 的基础知识以及前期准备以及完成， 接下来， 会带着我们实战了。分别为 WebUI ,Android、iOS、API 等。

5、Midscene.js 实例

如上所说，关于 Midscene.js 的自动化几乎适配的当前所有的维度，如：Web UI 自动化、Android 自动化、iOS 自动化、API 自动化等。

5.1 Web 自动化

5.1.1 集成到 Playwright

关于 Playwright 我相信大家都了解，尤其做 Web UI 自动化的同学， 对 Playwright 再熟悉不过了，尝于对网络应用程序进行端到端测试和网页抓取。

所以 Midscene.js 集成到 Playwright 也是必然的，常见方式有：

①直接用脚本方式集成和调用 Midscene Agent

② 在 Playwright 的测试用例中集成 Midscene

两者的区别有哪些：

维度	Playwright	Midscene Agent
集成方式	通过自定义 Fixture 嵌入 Playwright 测试结构	在代码中手动创建和管理 Agent 实例
语法风格	更贴近 Playwright 原生写法	更体现 Midscene 的独立 SDK 特性
使用便捷性	开箱即用，无需手动管理	需自行初始化和销毁
控制灵活性	遵循 Playwright 测试生命周期	可在测试外或非标准环境下使用
推荐场景	绝大多数基于 Playwright 的 E2E 测试项目	需要精细控制或与其他自动化框架混合编程的场景

你看，这两者的区别，是不是一目了然，如果你说能不能详细的讲一讲这两者的区别， 那接着往下看。

5.1.1.1 直接集成 Midscene Agent

实现步骤通常如下：

第一步：配置模型，详细参考第四节 Midscene.js 模型配置，这里就不做过多赘述。

第二步：安装依赖，方式有 npm、yarn、pnpm、bun 等方式。

• 这里就以 npm 为例，执行命令如下：

```
npm install @midscene/web playwright @playwright/test tsx --save-dev
```

第三步：脚本实例

```
import { chromium } from 'playwright';
import { PlaywrightAgent } from '@midscene/web/playwright';
import 'dotenv/config'; // 从 .env 文件读取环境变量

const sleep = (ms) => new Promise((r) => setTimeout(r, ms));

(async () => {
  const browser = await chromium.launch({
    headless: true, // 'true' 表示我们看不到浏览器窗口
    args: ['--no-sandbox', '--disable-setuid-sandbox'],
  });
  const page = await browser.newPage();
  await page.setViewportSize({ width: 1280, height: 768 });
  await page.goto('https://www.ebay.com');
  await sleep(5000);

  // 初始化 Midscene
  const agent = new PlaywrightAgent(page);

  // 输入关键词，执行搜索操作
  await agent.aiAction('type "MatPad" in search box, hit Enter');

  // 等待加载
  await agent.aiWaitFor('there is at least one headphone item on page');
  // 或者可以使用普通的 sleep: await sleep(1000);

  // 理解页面内容，查找商品
  const items = await agent.aiQuery(
    '{itemTitle: string, price: Number}[], find item in list and corresponding price',
  );
  console.log(MatPad in stock', items);
```

```
  const isMoreThan1000 = await agent.aiBoolean(
    'Is the price of the MatPad more than 1000?',
  );
  console.log('isMoreThan1000', isMoreThan1000);

  const price = await agent.aiNumber(
    'What is the price of the first headphone?',
  );
  console.log('price', price);

  const name = await agent.aiString(
    'What is the name of the first headphone?',
  );
  console.log('name', name);

  const location = await agent.aiLocate(
    'What is the location of the first headphone?',
  );
  console.log('location', location);

  // 使用 AI 断言
  await agent.aiAssert('There is a category filter on the left');

  // 点击第一个商品
  await agent.aiTap('the first item in the list');

  await browser.close();
})();
```

为了更直观的让看到这段代码，我对 API 进行说明：

API	作用	示例
aiAction	自动规划并执行自然语言描述的一系列操作	await agent.aiAction('type " MatPad " in search box, hit Enter');
aiTap	点击指定元素	await agent.aiTap('the first item in the list');
aiWaitFor	等待某条件成立	await agent.aiWaitFor('there is at least one headphone item on page');
aiQuery	提取结构化数据（JSON）	await agent.aiQuery('{itemTitle: string, price: Number}[], find item in list and corresponding price');
aiBoolean	提取布尔值	await agent.aiBoolean('Is the price > 1000?');
aiNumber	提取数字	await agent.aiNumber('What is the price of the first headphone?');
aiString	提取字符串	await agent.aiString('What is the name of the first headphone?');
aiAssert	断言	await agent.aiAssert('There is a category filter on the left');
aiLocate	定位元素坐标	await agent.aiLocate('Where is the first headphone?');

第四步：执行脚本

```
npx tsx demo.ts
```

敲黑板：

上述说了， Midscene.js 可以有拓展的能力， 关于 Midscene.js 的拓展可以通过 customActions 和 defineAction 扩展自定义交互动作，具体代码如下：

```
import { defineAction, z, getMidsceneLocationSchema } from '@midscene/core';
const ContinuousClick = defineAction({
  name: 'continuousClick',
  description: 'Click the same target repeatedly',
```

```
    paramSchema: z.object({
      locate: getMidsceneLocationSchema(),
      count: z.number().int().positive().describe('How many times to click'),
    }),
    async call(param) {
      const { locate, count } = param;
      console.log('click target center', locate.center);
      console.log('click count', count);
      // 实现自定义点击逻辑
    },
});

const agent = new PlaywrightAgent(page, {
  customActions: [ContinuousClick],
});

await agent.aiAction('点击绿色按钮三次');
```

5.1.1.2 在 Playwright 中集成 Midscene

实现步骤通常如下：

第一步：环境配置：

在项目根目录创建 .env 文件，配置 AI 模型参数。

第二步：创建 Fixture：

在 Playwright 测试项目中，创建一个自定义的 Fixture 文件，使用 createMidsceneFixture 来扩展原始的 test。

第三步：脚本示例

1)创建环境

```
// fixture.ts
import { test as base } from '@playwright/test';
import { createMidsceneFixture } from '@midscene/playwright';
```

```
// 通过 createMidsceneFixture 创建包含 Midscene 功能的 Fixture
export const test = createMidsceneFixture(base);
export { expect } from '@playwright/test';
```

2）执行查询操作：

```
// test.spec.ts
import { test, expect } from './fixture'; // 导入自定义 fixture
test('AI 查询测试', async ({ page, ai, aiQuery }) => { // ai 和 aiQuery 被注入到测试函数中
  await page.goto(' https://re.jd.com/ ');
  // 使用自然语言进行交互与断言
  await ai('在搜索框输入 " MatPad" 并敲回车');
  const items = await aiQuery('{itemTitle: string, price: Number}[], 提取商品列表的标题和价格');
  expect(items.length).toBeGreaterThan(0);
});
```

5.1.2 桥接模式（Bridge Mode）

如果不喜欢集成在 Playwright 这种方式，还有一个更简便的，就是直接基于 Chrome 的桥接模式，不需要太复杂的编码能力，完全是 WebUI 操作。

操作步骤非常简单：

第一步：环境准备

```
# 设置你的 AI 服务密钥
export OPENAI_API_KEY="你的密钥"

# 安装必要依赖
npm install @midscene/web tsx --save-dev
```

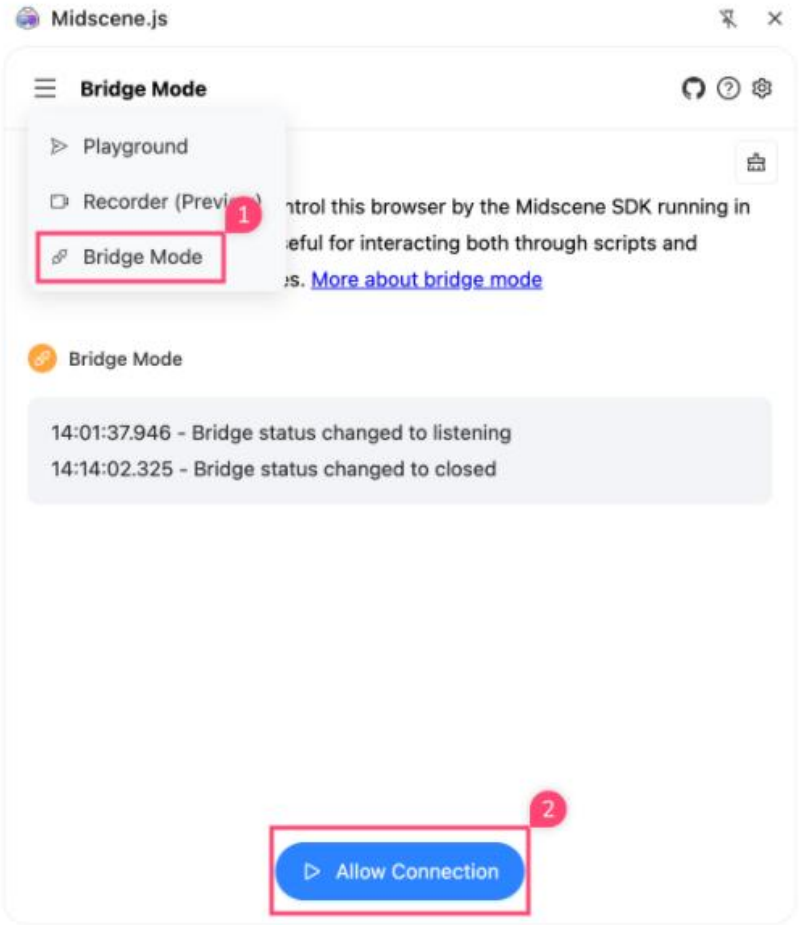
如果不会配置模型 API_KEY 和安装依赖，可以参照第三、第四章节，有详细说明，这里就不赘述。

第二步：启用插件

1）在 Chrome 应用商店搜索并安装 Midscene 插件

2）切换到 “Bridge Mode” 标签

3）点击 “Allow connection” 按钮



第三步：脚本示例

```
import { AgentOverChromeBridge } from "@midscene/web/bridge-mode";
// 创建一个桥接代理实例
const agent = new AgentOverChromeBridge();
(async () => {
  try {
    // 连接到新标签页并打开目标网站
    await agent.connectNewTabWithUrl("https://re.jd.com/ ");
    //控制浏览器
```

```
    await agent.ai('在搜索框中输入"AI 自动化"，然后按回车');
    // 等待一下，让页面加载完成
    await new Promise(resolve => setTimeout(resolve, 3000));

    // 验证搜索结果是否出现
    await agent.aiAssert("页面上显示了搜索结果");
    console.log(" 自动化任务完成！ ");
  } catch (error) {
    console.error("任务失败:", error.message);
  } finally {
    // 记得断开连接，释放资源
    await agent.destroy();
  }
})();
```

6、总结

看到这里，相信你已经对 Midscene.js 有了全新的认识。这款工具展现出的潜力，是否让你对智能自动化的未来充满了期待？

别着急，今天的内容只是打开这扇大门的第一道，在接下来的第二讲、第三讲中，我将带你深入更广阔的应用场景。

最后，按照流程，我们再回顾一下今天主要内容：

- 什么是 Midscene.js
- Midscene.js 与 Browser UI 的差异点
- Midscene.js 的安装方式
- Midscene.js 的模型配置方式
- Midscene.js 在 WebUI 的应用案例

最后，再唠叨一句：掌握工具最好的方式，就是实践。”

DeepSeek、豆包、ChatGPT 如何重塑测试用例设计

◆作者：崔岢

在当今快速发展的软件行业，软件产品的迭代速度极快，功能复杂度也在不断提升。为了应对这一挑战，测试团队必须寻找高效的方法来确保软件的质量。近年来，人工智能（AI）技术的兴起为测试领域带来了新的机遇。本文将探讨如何利用 DeepSeek、豆包等 AI 工具编写高质量的测试用例，以提升测试工作的专业性和效率。

一、传统测试用例设计的四大瓶颈

AI 辅助传统测试,高度依赖个人经验、耗时且重复性高、覆盖度难以保障、维护成本巨大。

高度依赖个人经验：测试用例质量与测试工程师经验强相关，团队水平参差不齐，新人培养周期长。

耗时且重复性高：复杂功能测试设计占用 30%-40%测试时间，大量重复性工作消耗工程师精力。

覆盖度难以保障：人工难以穷尽所有场景，边界条件、异常场景经常遗漏。

维护成本巨大：需求变更导致测试用例需要频繁更新，版本管理复杂。

对比维度	传统测试设计	AI 辅助测试
设计速度	慢（2-3 天/模块）	快（1-2 小时/模块）
场景覆盖度	依赖个人能力，易遗漏	系统化覆盖，减少遗漏
边界测试	经验驱动，不全面	自动生成边界值用例
一致性	因人而异，标准不一	统一标准，格式规范
维护成本	高，需人工更新	低，可快速重新生成
创新能力	有限，受限于经验	强，能发现非预期场景

表 1 传统测试设计与 AI 辅助测试的主要差异

二、基于 AI 工具的测试用例编写实践

2.1 需求分析与工具选择

根据项目特点和团队技能水平，选择合适的 AI 工具至关重要。选型考量因素包括：项目技术栈兼容性、团队学习成本曲线、工具集成能力（CI/CD）、成本与 ROI 分析、表 2 是不同测试类型和测试场景推荐的工具。

测试类型	适用场景	推荐工具	优势
功能测试	Web/移动应用测试	DeepSeek, Testim	视觉识别，自愈性定位
接口测试	API 测试验证	豆包, Postbot	智能生成测试数据，异常场景覆盖
性能测试	负载压力测试	LoadNinja	智能并发控制，实时分析
安全测试	漏洞扫描	Contrast	智能漏洞识别，风险评估

表 2 测试场景下各 AI 工具优势

2.2 测试用例生成

2.2.1 使用 DeepSeek 生成测试用例

假设我们需要测试一个电商平台的商品查询功能，原始需求为商品查询功能支持按类别、价格、销量等多维度筛选，同时支持关键词搜索和排序功能，对应的 sql 如图 1 所示。

```sql
-- 商品查询SQL（简化版）
SELECT * FROM products
WHERE category = #{category}
AND price < #{price}
AND status = 'ON_SALE'
ORDER BY #{sort_field} #{sort_order};
```

图 1 商品查询 sql

AI 分析过程：解析 SQL 结构，识别所有查询条件和参数、分析数据库中的实际数据分布，识别边界值、提取历史缺陷数据，重点关注易出错场景、结合业务规则生成测试场景矩阵，最终生成测试用例如图 2 所示。

DeepSeek生成的测试用例如下：

```
# 商品查询功能测试用例

## 正常流测试用例
1. TC001 - 查询电子产品类别且价格低于1000元的商品
   - 输入：category="电子产品", price=1000
   - 预期：返回所有符合条件的商品列表

2. TC002 - 查询家居用品类别且价格在500-1000元之间的商品
   - 输入：category="家居用品", price=1000
   - 预期：返回价格<1000的家居用品

## 边界值测试用例
3. TC003 - 查询价格恰好为1000元的电子产品
   - 输入：category="电子产品", price=1000
   - 预期：不返回价格为1000的商品（因条件是<）

4. TC004 - 查询不存在的商品类别
   - 输入：category="不存在的类别", price=1000
   - 预期：返回空列表

## 异常流测试用例
5. TC005 - 价格参数为负数
   - 输入：category="电子产品", price=-100
   - 预期：系统正确处理，返回空列表或错误信息

6. TC006 - 类别参数为空值
   - 输入：category=null, price=1000
   - 预期：系统返回所有价格<1000的商品
```

图 2 DeepSeek 生成测试用例

2.2.2 使用豆包生成接口测试用例

豆包适用于接口测试，现在有个用户管理系统接口测试案例，测试准出：

接口定义：

方法：POST

端点：/api/users

参数：username, email, password, role

成功响应：201 Created，包含用户 ID 和创建时间

豆包生成的测试用例：

```
{
  "test_cases": [
    {
      "name": "用户创建成功流程",
      "method": "POST",
      "url": "/api/users",
      "headers": {
        "Content-Type": "application/json",
        "Authorization": "Bearer ${token}"
      },
      "body": {
        "username": "testuser_${timestamp}",
        "email": "test${timestamp}@example.com",
        "password": "Password123!",
        "role": "user"
      },
      "expected": {
        "status_code": 201,
        "schema": {
          "type": "object",
          "required": ["id", "create_time"],
          "properties": {
            "id": {"type": "string", "pattern": "^[a-f0-9]{24}$"},
            "create_time": {"type": "string", "format": "date-time"}
          }
        },
        "response_time": {"max": 1000}
      }
    },
    {
      "name": "邮箱格式验证",
      "method": "POST",
      "url": "/api/users",
      "body": {
        "username": "testuser",
        "email": "invalid-email",
        "password": "Password123!",
        "role": "user"
      },
      "expected": {
        "status_code": 400,
        "error_code": "INVALID_EMAIL"
      }
    }
  ]
}
```

图 3 豆包生成接口测试用例

2.3 建立多维度验证机制

AI 生成的测试用例必须经过严格验证才能投入使用。图 4 为 AI 测试用例的生成评审

流程， 测试用例质量评估指标为：

需求覆盖率（≥95%）：确保所有需求都有对应测试用例

代码覆盖率（≥80%）：语句覆盖、分支覆盖、路径覆盖

缺陷检出率：衡量测试用例的有效性

测试执行通过率：评估测试用例的稳定性

图 4 为 AI 测试用例的生成评审流程

2.4 测试用例优化策略

基于 Google 的测试专家 James Whittaker 的经验，我们建立了以下优化机制，优化方向包括去冗余、补遗漏、提效率、增稳定，优化的示例如图 5 所示：

去冗余：使用代码相似性分析识别重复测试用例

补遗漏：基于代码变更和缺陷分析补充测试场景

提效率：优化测试数据准备和执行顺序

增稳定：增强用例的稳定性和容错性

优化实践示例：

```python
# 优化前：硬编码测试数据
def test_user_creation():
    data = {
        "username": "testuser001",
        "email": "test001@example.com",
        "password": "Password123!"
    }
    # 测试逻辑...

# 优化后：使用工厂模式生成测试数据
def test_user_creation(user_factory):
    user_data = user_factory.build(role="admin")
    response = create_user(user_data)
    assert response.status_code == 201

# 进一步优化：参数化测试
@pytest.mark.parametrize("user_role,expected_status", [
    ("admin", 201),
    ("user", 201),
    ("invalid_role", 400)
])
def test_user_roles(user_factory, user_role, expected_status):
    user_data = user_factory.build(role=user_role)
    response = create_user(user_data)
    assert response.status_code == expected_status
```

图 5 测试用例优化示例

2.5 测试自动化框架集成

将 AI 生成的测试用例无缝集成到自动化测试框架中，集成架构如图 6 所示，由需求到测试用例生成再到缺陷管理系统形成自动化的测试架构， Jenkins Pipeline 集成示例如图 7 所示。

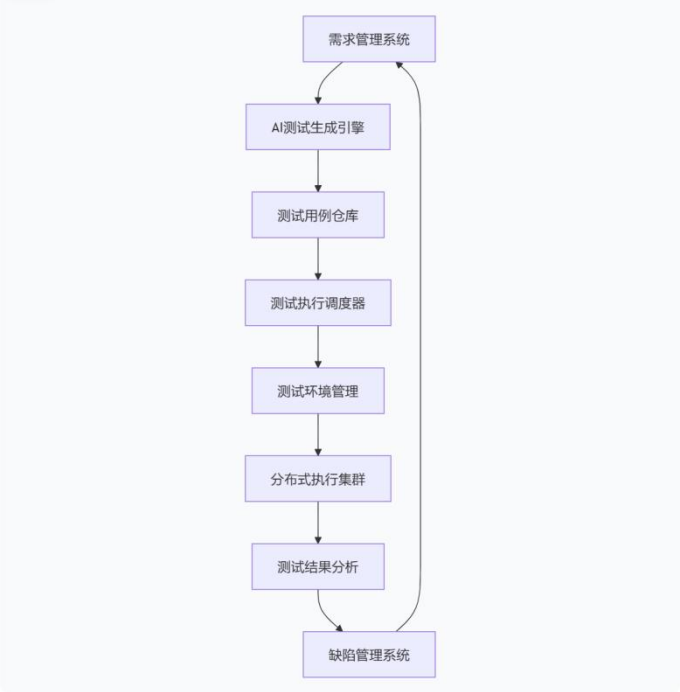


图 6 自动化测试集成框架

实践示例：Jenkins Pipeline集成

```
pipeline {
    agent any
    environment {
        TEST_ENV = 'staging'
        AI_GENERATOR = 'deepseek'
    }
    stages {
        stage('Generate Test Cases') {
            steps {
                script {
                    // 基于需求变更自动生成测试用例
                    sh """
                    python ai_test_generator.py \
                    --requirement $JOB_BASE_NAME \
                    --output generated_tests \
                    --tool $AI_GENERATOR
                    """
                }
            }
        }
        stage('Execute Tests') {
            parallel {
                stage('API Tests') {
                    steps {
                        sh 'pytest generated_tests/api/ --junitxml=api-results.xml'
                    }
                }
                stage('UI Tests') {
                    steps {
                        sh 'pytest generated_tests/ui/ --junitxml=ui-results.xml'
                    }
                }
            }
        }
        stage('Analysis & Report') {
            steps {
                junit '*-results.xml'
                script {
                    def coverage = getCoverage()
                    currentBuild.description = "覆盖率: ${coverage}% | 通过率: ${currentBuild.result}"

                    // 自动生成测试报告
                    generateTestReport()

                    // 根据结果自动创建缺陷
                    if (coverage < 80) {
                        createJiraIssue('测试覆盖率不足', '需要补充测试用例')
                    }
                }
            }
        }
    }
    post {
        always {
            // 清理和通知
            cleanWs()
            script {
                notifyTestResults()
            }
        }
    }
}
```

图 7Jenkins Pipeline 集成示例

3. 结语

本文探讨了 AI 工具如何颠覆传统测试用例编写模式，为核心工作流带来三重价值：显著提升编写效率、提高覆盖率与缺陷检出能力、赋能持续测试流水线。成功的 AI 测试实践需要人的智慧与工具的能力完美结合，需要流程的优化与技术的创新协同发展。随

着 MIT 研究人员提出的"AI-Human Collaboration"模式的发展，我们相信未来测试领域将出现更加紧密的人机协作模式。

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：AI 测试

命令行参数的 3 种实战用法

◆ 作者：Tynam

大家是否还记得《黑客帝国》里那些经典名场面？主角们端坐屏幕前，漆黑的背景上滚动着绿色数据流，指尖在键盘上飞速敲击，一行行命令行代码如咒语般倾泻而出 —— 无需复杂界面，无需多余操作，仅仅凭借一串精准指令，就能穿透系统壁垒、掌控数字世界，甚至达到 “颠覆一切” 的震撼效果。那种纯粹用代码对话机器的炫酷感，那种指令下达后即刻生效的掌控力，既充满了科技的神秘张力，也让无数人对命令行工具生出莫名的向往。而在软件自动化测试领域，命令行同样藏着类似的 “魔力”，一行参数就能灵活切换测试环境、精准筛选测试用例，让自动化脚本摆脱硬编码的束缚，变得高效又灵活。

就像我们日常高频使用的单元测试框架 pytest，正是命令行参数应用的典型例子。只需在终端输入 pytest --help，屏幕上便会清晰罗列所有支持的命令行参数，从用例筛选、运行模式到报告生成、日志级别，每一项都有明确说明。我们直接输入 pytest 就可执行全量用例，或者拼接一定的参数（如 pytest -m smoke --timeout 60）按照需求精准控制测试范围、设置运行阈值，让自动化测试按照预期高效执行。

```
ydjdeMacBook-Air:~ ydj$ pytest --help
usage: pytest [options] [file_or_dir] [file_or_dir] [...]

positional arguments:
  file_or_dir

general:
  -k EXPRESSION         Only run tests which match the given substring expression. An expression is a Python
                        evaluatable expression where all names are substring-matched against test names and their
                        parent classes. Example: -k 'test_method or test_other' matches all test functions and classes
                        whose name contains 'test_method' or 'test_other', while -k 'not test_method' matches those
                        that don't contain 'test_method' in their names. -k 'not test_method and not test_other' will
                        eliminate the matches. Additionally keywords are matched to classes and functions containing
                        extra names in their 'extra_keyword_matches' set, as well as functions which have names
                        assigned directly to them. The matching is case-insensitive.
  -m MARKEXPR           Only run tests matching given mark expression. For example: -m 'mark1 and not mark2'.
  --markers             show markers (builtin, plugin and per-project ones).
  -x, --exitfirst       Exit instantly on first error or failed test
  --fixtures, --funcargs
                        Show available fixtures, sorted by plugin appearance (fixtures with leading '_' are only shown
                        with '-v')
  --fixtures-per-test   Show fixtures per test
  --pdb                 Start the interactive Python debugger on errors or KeyboardInterrupt
  --pdbcls=modulename:classname
                        Specify a custom interactive Python debugger for use with --pdb.For example:
                        --pdbcls=IPython.terminal.debugger:TerminalPdb
  --trace               Immediately break when running each test
  --capture=method      Per-test capturing method: one of fd|sys|no|tee-sys
  -s                    Shortcut for --capture=no
  --runxfail            Report the results of xfail tests as if they were not marked
  --lf, --last-failed   Rerun only the tests that failed at the last run (or all if none failed)
  --ff, --failed-first  Run all tests, but run the last failures first. This may re-order tests and thus lead to
```

仅凭几句简单命令，就能让自动化测试精准贴合需求执行，是不是很有魅力？其实，命令行运行程序远没有《黑客帝国》里渲染的那般神秘莫测，它本质上就是一种直接与程序对话的简洁方式，无需复杂交互，用参数传递指令，就能让工具精准响应我们的需求。

本文我们就来聊聊如何制作命令行参数，实现自定义命令。本文将分别使用 sys.argv、argparse 和 pytest_addoption 三种方法进行演示。

一、sys.argv

sys.argv 是一个简单直接的命令行参数获取方式，适合快速处理少量、逻辑简单的参数场景（如测试开关、简单参数传递等）。它本质是一个列表，存储了命令行输入的所有参数，无需额外安装库，直接通过标准库 sys 调用即可。

既然是列表，我们就可以通过列表的方式使用它。我们先通过列表打印 sys.argv 是如何获取参数的。

先一个简单的脚本，内容如下：

```
# test_sys_argv.py
import sys
for i in range(len(sys.argv)):
    print(f"第 {i} 个参数是：{sys.argv[i]}")
```

在命令行中输入运行 test_sys_argv.py 脚本的命令，然后添加一些参数。脚本运行后会将添加的所有参数都打印出来。

我们分别输入 python test_sys_argv.py 和 python test_sys_argv.py env=abc is_auto=True auth=tynam 运行脚本，输出结果如下。

```
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python test_sys_argv.py
第 0 个参数是: test_sys_argv.py
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python test_sys_argv.py env=abc is_auto=True auth=tynam
第 0 个参数是: test_sys_argv.py
第 1 个参数是: env=abc
第 2 个参数是: is_auto=True
第 3 个参数是: auth=tynam
(enve) ydjdeMacBook-Air:cmdline_params ydj$
```

从结果中可以看到，获取到的第一个参数被调用的脚本文件名，然后依次是输入的参数。因此我们可以这样理解，sys.argv 获取到的就是用户在命令行中输入的从第一个被调用的脚本文件名开始，以空格分割元素的一个列表。

了解了 sys.argv 的内容，我们就很容易制作一些参数，根据我们所需执行脚本。

我们在自动化测试经常会遇到环境切换的问题，dev、stage 等环境。下面我们添加一个命令行参数用来控制自动化脚本在哪个环境上执行。代码如下：

```
# env_switching_with_argv.py
import sys
# 定义基本的信息
env_info = {
    "dev": {
        "host": "dev.db.example.com",
        "user": "dev_user",
        "password": "dev_password",
    },
    "stage": {
        "host": "stage.db.example.com",
        "user": "stage_user",
        "password": "stage_password",
    }
}
# 获取命令行参数
def get_env():
    env = "dev"  # 默认测试环境
    # 使用 sys.argv 解析命令行参数
    for i in range(1, len(sys.argv)):
        arg = sys.argv[i]
        if arg.startswith("--env="):
            env = arg.split("=")[1]
            if env not in ["dev", "stage"]:
                raise ValueError(f"错误：无效的环境参数 {env}，可选值：dev/stage")
```

```
        return env
    def test_case():
        print(f"测试开始执行...")
        env = env_info.get(get_env())
        print(f"当前环境地址是：{env.get('host')}，使用的用户名和密码是：
{env.get('user')}/{env.get('password')}")
        print(f"测试执行完成")
    if __name__ == '__main__':
        test_case()
```

首先定义了一个字典 env_info，存储 dev 和 stage 不同环境的配置信息，环境信息中分别配置 host、user 和 password；然后添加了一个命令行参数解析函数 get_env()，用以获取当前用户想要执行的环境，默认值为 dev。当代码执行后，程序便会查找用户是否输入了以 --env= 开头的参数，如何输入了则提取环境值，然后对环境值进行合法性校验，若输入的环境不在 ["dev", "stage"] 中，则抛出异常提示错误；最后模拟了一条测试用例，在测试用例中调用 get_env() 获取命令行指定的环境，然后从 env_info 中加载该环境的配置。

下面我们从三个场景验证添加的参数，如果不添加--env 参数，则会使用默认的 dev 环境；如果添加了--env 参数，则会使用指定的环境；如果添加的--env 参数值不合法，则抛出异常。

在命令行中依次执行命令 python env_switching_with_argv.py、python env_switching_with_argv.py --env=stage 和 python env_switching_with_argv.py --env=test。结果如下所示：

```
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python env_switching_with_argv.py
测试开始执行...
当前环境地址是：dev.db.example.com，使用的用户名和密码是：dev_user/dev_password
测试执行完成
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python env_switching_with_argv.py --env=stage
测试开始执行...
当前环境地址是：stage.db.example.com，使用的用户名和密码是：stage_user/stage_password
测试执行完成
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python env_switching_with_argv.py --env=test
测试开始执行...
Traceback (most recent call last):
  File "/Users/ydj/Desktop/tynam/pythonProject/cmdline_params/env_switching_with_argv.py", line 41, in <module>
    test_case()
  File "/Users/ydj/Desktop/tynam/pythonProject/cmdline_params/env_switching_with_argv.py", line 35, in test_case
    env = env_info.get(get_env())
  File "/Users/ydj/Desktop/tynam/pythonProject/cmdline_params/env_switching_with_argv.py", line 29, in get_env
    raise ValueError(f"错误：无效的环境参数 {env}，可选值：dev/stage")
ValueError: 错误：无效的环境参数 test，可选值：dev/stage
(enve) ydjdeMacBook-Air:cmdline_params ydj$
```

从结果可以看到，与预期一致。

通过 sys.argv 解析命令行参数 --env，实现了测试环境的动态切换，避免了修改代码中环境配置的麻烦。这种方式适合简单场景下的多环境测试，尤其在本地调试或快速验证不同环境时非常便捷。

二、argparse

argparse 是 Python 标准库中用于解析命令行参数的模块，专为处理命令行输入设计，支持参数定义、类型校验、自动生成帮助信息等功能。它可以自动生成帮助和使用信息，并且能够处理命令行参数的解析和错误检查。

argparse 的使用分为两步，第一步是使用 argparse.ArgumentParser() 创建一个解析器对象，用来负责管理和解析命令行参数。第二步是使用 add_argument() 方法向解析器中添加参数，这也是这块的重点内容，例如：

```
parser.add_argument(
    "--env", "-e",  # 长选项 --env，短选项 -e（可简化输入）
    type=str,
    choices=["dev", "stage", "prod"],  # 只允许这三个值
    default="dev",
    help="测试环境（默认：dev，可选：dev/stage/prod）"
)
```

而添加参数的核心操作又在参数的定义上。我们先来看看 add_argument() 方法中常用的一些参数：

- name or flags：参数名，例如 '-env' 或 '--e'。
- action：布尔值开关，输入参数时值为 True，否则 False。
- default：默认值，当参数未在命令行中出现并且也不存在于命名空间对象时给的值。
- type：命令行参数类型，例如 int、str。
- choices：参数可选值。如?["dev", "stage"]。

- required：是否必须参数。
- help：参数描述。

了解了方法中的参数说明，接下来使用就会变得异常容易。

下面我们还是通过“添加一个命令行参数用来控制自动化脚本在哪个环境上执行”的示例来说明 argparse 的使用。示例代码如下：

```
# env_switching_with_argparse.py
import argparse
# 定义基本的信息
env_info = {
    "dev": {
        "host": "dev.db.example.com",
        "user": "dev_user",
        "password": "dev_password",
    },
    "stage": {
        "host": "stage.db.example.com",
        "user": "stage_user",
        "password": "stage_password",
    }
}
# 获取命令行参数
def get_env():
    # 创建解析器
    parser = argparse.ArgumentParser(description="多环境切换")
    # 添加测试环境参数 --env 或 -e
    parser.add_argument(
        "--env", "-e",
        type=str,
        choices=["dev", "stage"],  # 只允许这两个值
        default="dev",
        help="指定测试环境（默认：dev，可选：dev/stage）"
```

```
        )
        # 解析参数：将命令行输入转换为可调用的对象
        args = parser.parse_args()
        return args.env
    def test_case():
        print(f"测试开始执行...")
        env = env_info.get(get_env())
        print(f"当前环境地址是：{env.get('host')}，使用的用户名和密码是：
{env.get('user')}/{env.get('password')}")
        print(f"测试执行完成")
    if __name__ == '__main__':
        test_case()
```

与 sys.argv 中的示例代码相比，我们只是修改了 get_env()函数，其他内容都没有做改动。

完成后我们来进行验证。我们在命令行工具中使用 --help 参数就能看到添加的参数 --env 以及描述内容，如下图所示。

```
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python env_switching_with_argparse.py --help
测试开始执行...
usage: env_switching_with_argparse.py [-h] [--env {dev,stage}]

多环境切换

options:
  -h, --help            show this help message and exit
  --env {dev,stage}, -e {dev,stage}
                        指定测试环境（默认：dev，可选：dev/stage）
(enve) ydjdeMacBook-Air:cmdline_params ydj$
```

下面我们从三个场景验证添加的参数，如果不添加--env 参数，则会使用默认的 dev 环境；如果添加了--env 参数，则会使用指定的环境；如果添加的--env 参数值不合法，则抛出异常。

在命令行中依次执行命令 python env_switching_with_argparse.py、python env_switching_with_argparse.py --env stage 和 python env_switching_with_argparse.py --env test。结果如下所示：

```
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python env_switching_with_argparse.py
测试开始执行...
当前环境地址是：dev.db.example.com，使用的用户名和密码是：dev_user/dev_password
测试执行完成
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python env_switching_with_argparse.py --env stage
测试开始执行...
当前环境地址是：stage.db.example.com，使用的用户名和密码是：stage_user/stage_password
测试执行完成
(enve) ydjdeMacBook-Air:cmdline_params ydj$ python env_switching_with_argparse.py --env test
测试开始执行...
usage: env_switching_with_argparse.py [-h] [--env {dev,stage}]
env_switching_with_argparse.py: error: argument --env/-e: invalid choice: 'test' (choose from 'dev', 'stage')
```

从结果可以看到，与预期一致。

我们可以看到，在使用上，argparse 的实例比 sys.argv 实例更具魅力。argparse 主要优点有：

- 自动校验：通过 choices、type 等参数自动检查输入合法性，无需手动写校验逻辑。
- 帮助信息：--help 自动生成参数说明，方便他人使用。
- 默认值处理：直接通过?default?定义默认值，无需手动判断 “是否传入参数”。

三、pytest_addoption

如果自动化测试中使用的 pytest 框架，要实现命令行参数，那么我会强烈建议你使用 pytest_addoption。因为他是直接嵌套在 pytest 单元测试框架中，参数解析逻辑与 pytest 测试框架无缝衔接，可以集成使用，无需再做过多的处理。

在 pytest 框架中，pytest_addoption 是一个核心钩子函数（hook），用于注册自定义命令行参数，让我们能在运行 pytest 测试时通过命令行传递个性化参数（如测试环境、账号信息、运行模式等），并在测试用例或 fixture 中灵活使用。

pytest_addoption 的使用，主要由下面三个核心步骤组成。

- 注册参数：在项目 conftest.py 文件中定义 pytest_addoption 函数，使用 parser.addoption() 注册参数。conftest.py 是 pytest 自动识别的钩子文件，无需导入即可生效。
- 封装参数：定义 fixture 函数，通过 request.config.getoption() 获取参数值。
- 使用参数：在测试函数中直接引用 fixture，获取命令行参数值并应用于测试逻辑。

下面我们进行实践演示，通过命令行参数化实现测试的基础 URL 地址、用户名和密码。依照上面的三个核心步骤我们来实践：

步骤 1：在 conftest.py 中注册参数

在项目 conftest.py 文件中定义 pytest_addoption 函数，然后使用 parser.addoption() 注册基础 URL 地址、用户名和密码三个参数。添加的代码如下：

```
# conftest.py
import pytest
def pytest_addoption(parser):
    # 注册 --base_url 参数
    parser.addoption(
        "--base_url",
        action="store",
        type=str,
        default="dev.example.com",
        help="指定测试环境的基础 URL"

    # 注册 --user 参数
    parser.addoption(
        "--user",
        action="store",
        type=str,
        default="admin",
        help="指定测试账户"

    # 注册 --pwd 参数
    parser.addoption(
        "--pwd",
        action="store",
        type=str,
        default="123456",
        help="指定测试账户密码"
```

pytest_addoption(parser) 是固定写法，parser.addoption 里面的参数写法与前面 argparse 中参数写法相同。我们依次添加了--base_url、--user 和--pwd 三个参数。

步骤 2：封装参数

我们在 conftest 中定义 fixture 函数，并通过 request.config.getoption() 获取--base_url、--user 和--pwd 参数值。代码如下：

```
# conftest.py
@pytest.fixture(scope="session")
def get_base_url(request):
    """获取测试环境基础 URL"""
    return request.config.getoption("--base_url")
@pytest.fixture(scope="session")
def get_user_pwd(request):
    """获取用户名和密码"""
     return request.config.getoption("--user"), request.config.getoption("--pwd")
```

将 fixture 设置为 session 级，无论什么时候都可以使用。定义了一个获取基础 URL 的函数 get_base_url 和获取用户名和密码的 get_user_pwd 函数。由于用户名和密码一般都是成对使用的，因此我们只用一个函数 get_user_pwd 返回其值。

步骤 3：使用参数

接下来我们在测试函数中直接引用 fixture，获取命令行参数值并应用于测试逻辑。代码如下：

```
# test_case_with_pytest_addoption.py
def test_case(get_base_url, get_user_pwd):
    # 根据参数参数执行对应的测试逻辑
    print(f"测试开始执行...")
    user, pwd = get_user_pwd
    print(f"当前环境地址是：{get_base_url}，使用的用户名和密码是：{user}/{pwd}")
    print(f"测试执行完成")
```

上面是一个简单的测试用例，测试函数 test_case 中直接传入 get_base_url 和 get_user_pwd 函数（fixture）名即可执行对应的 fixture 并获取其返回值。

至此从注册到测试用例应用，参数的创建和使用就完成了，接下来就是使用。

首先在命令行工具中输入 pytest --help，查看帮助信息，查看我们注册的参数。截图如下：

```
Custom options:
  --lsof               Run FD checks if lsof is available
  --runpytest={inprocess,subprocess}
                       Run pytest sub runs in tests using an 'inprocess' or 'subprocess' (python -m main) method
  --base_url=BASE_URL  指定测试环境的基础 URL
  --user=USER          指定测试账户
  --pwd=PWD            指定测试账户密码
```

在自定义参数选项（Custom options）下可以看到我们刚才注册的三个参数--base_url、--user 和--pwd 及其说明。说明参数注册成功。

下面我们从两个场景验证添加的参数，如果不添加--base_url、--user 和--pwd 三个参数，则会使用默认的 dev.example.com、admin、123456 三个值；如果添加了--base_url、--user 和--pwd 三个参数，则会使用指定的值。

在命令行中依次执行命令 pytest test_case_with_pytest_addoption.py -s、pytest test_case_with_pytest_addoption.py -s --base_url dev.com --user admin --pwd admin123。为了将测试用例中的 print 内容输出，我们在执行时添加了 -s 参数。结果如下所示：

```
(enve) ydjdeMacBook-Air:cmdline_params ydj$ pytest test_case_with_pytest_addoption.py -s
============================= test session starts ==============================
platform darwin -- Python 3.10.7, pytest-8.3.4, pluggy-1.5.0
rootdir: /Users/ydj/Desktop/tynam/pythonProject
plugins: anyio-4.9.0, html-4.1.1, metadata-3.1.1, langsmith-0.4.42
collected 1 item

test_case_with_pytest_addoption.py 测试开始执行...
当前环境地址是：dev.example.com，使用的用户名和密码是：admin/123456
测试执行完成
.

============================== 1 passed in 0.03s ===============================

(enve) ydjdeMacBook-Air:cmdline_params ydj$ pytest test_case_with_pytest_addoption.py -s --base_url dev.com --user admin --pwd admin123
============================= test session starts ==============================
platform darwin -- Python 3.10.7, pytest-8.3.4, pluggy-1.5.0
rootdir: /Users/ydj/Desktop/tynam/pythonProject
plugins: anyio-4.9.0, html-4.1.1, metadata-3.1.1, langsmith-0.4.42
collected 1 item

test_case_with_pytest_addoption.py 测试开始执行...
当前环境地址是：dev.com，使用的用户名和密码是：admin/admin123
测试执行完成
.

============================== 1 passed in 0.04s ===============================

(enve) ydjdeMacBook-Air:cmdline_params ydj$
```

结果与预期一致。

四、总结

本文围绕自动化测试中的命令行参数自定义需求，系统介绍了 sys.argv、argparse 和 pytest_addoption 三种实现方案，从基础用法到实战场景，覆盖了不同测试场景下的参数设计需求。三种方案各有侧重，适配不同的测试环境和技术选型：

sys.argv 以 “轻量无依赖” 为核心优势，无需复杂配置，几行代码即可实现简单参数解析，适合本地调试、临时脚本或仅需 1-2 个参数的场景。但需手动处理参数校验和格式判断，扩展性较弱，不适合复杂参数逻辑。

argparse 作为 Python 标准库的 “专业工具”，支持自动参数校验、帮助信息生成、多类型参数定义（如布尔开关、可选值限制），兼顾易用性和扩展性。无需依赖第三方框架，适合独立的自动化测试脚本，尤其适合需要向团队共享、参数较多的场景。

pytest_addoption 是 pytest 框架的 “原生方案”，与 pytest 生态深度集成，通过 conftest.py 注册参数、fixture 封装参数，无需额外处理框架兼容问题。适合 pytest 驱动的自动化测试项目，参数可在所有测试用例中复用，尤其适合多环境、多配置项的复杂测试场景。

从使用场景来看，若仅需快速实现简单参数控制，优先选择 sys.argv；若需构建规范、可复用的测试脚本，argparse 是更优选择；若项目已基于 pytest 框架，pytest_addoption 能最大程度发挥框架优势，让参数管理更高效。

读完本文，你还会觉得在命令行工具中输入一条条命令就能完成各种操作很神秘吗？

拓展学习

[2] 解锁【企业级云端项目实战】试听，获取全链路测试技术全景

咨询：微信 atstudy-js 备注：11

APP 专项测试 - 弱网测试指南

◆作者：测测小仙女儿

一、弱网测试核心概念

1.定义与范围

弱网测试是移动端健壮性测试的核心环节，需覆盖断网、网络故障等复杂场景。其数据定义因应用类型而异：

- 移动端通用标准：低于 2G 速率或 3G 网络均属弱网范畴
- 特殊场景：弱信号 WiFi 同样纳入测试范围
- 业务定制：需结合具体业务逻辑划分网络质量阈值

说人话就是：不仅仅是看“能不能用”，更是要看“用得顺不顺畅”、“会不会出奇怪的问题”。

2.测试必要性

国内移动设备普及催生多弱网场景：

A[车站/地铁]-->B[电梯/楼梯间]

C[高铁/隧道]-->D[地下车库]

用户在这些场景下的体验痛点：

- 核心功能中断导致用户流失
- 交互卡顿降低使用意愿
- 数据不同步引发信任危机

3.弱网涉及概念

带宽：单位时间内能够传输的数据量，一般单位 kbs，分为上行和下行带宽，上行指上传文件的速率，下行指下载文件的速率，通常下行速率大于上行速率。

延时：网络延时，可以简单的理解为客户端向服务器发送一个请求，这个请求发出到服务器立刻返回并达到客户端的时间，叫做网络延时，通常是 ms 单位。

——延时可能导致的问题，可以在没有返回的时候发出多个重复的请求，延时导致返回的数据包的顺序不一致。

丢包：数据包，数据在传输过程中是会被划分为多个数据包的形式进行传输的，客户端发出的包没有到达或者服务器返回的包没有到达，就是丢包

抖动：评估网络稳定性的。

二、测试实施框架

1.关键测试思路

常见弱网缺陷清单：

缺陷类型	典型表现	优化方向
资源加载	图片加载失败/模板混乱	加载逻辑优化
交互体验	响应超时无反馈	增加重试机制
系统稳定性	页面崩溃/UI 错乱	请求队列管理
数据一致性	Session 丢失	连接保持机制

2.强制质量标准

必须满足要求：

+核心场景断线自动重连

+网络抖动时保持状态一致

+杜绝卡死/崩溃等致命异常

严禁出现情况：

- 核心功能（登录/支付）阻断
- 重连强制退回登录页
- 无网络延迟提示机制

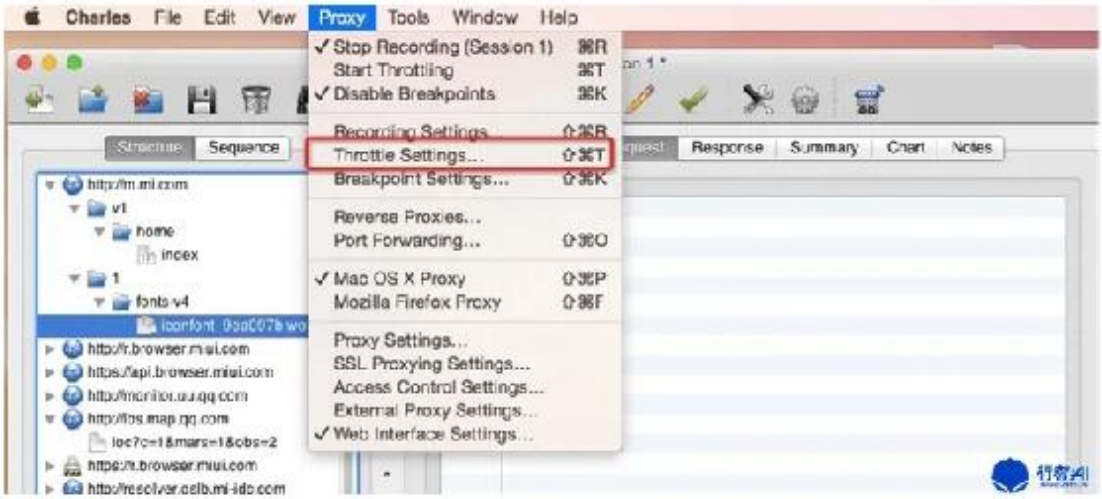
三、常用弱网测试工具

1.三大测试方案详解

方案一：PC 代理工具（Fiddler/Charles）

实现原理：

设备网络代理→PC 抓包工具→延时设置



进行常用的网速模拟设置

优劣分析：

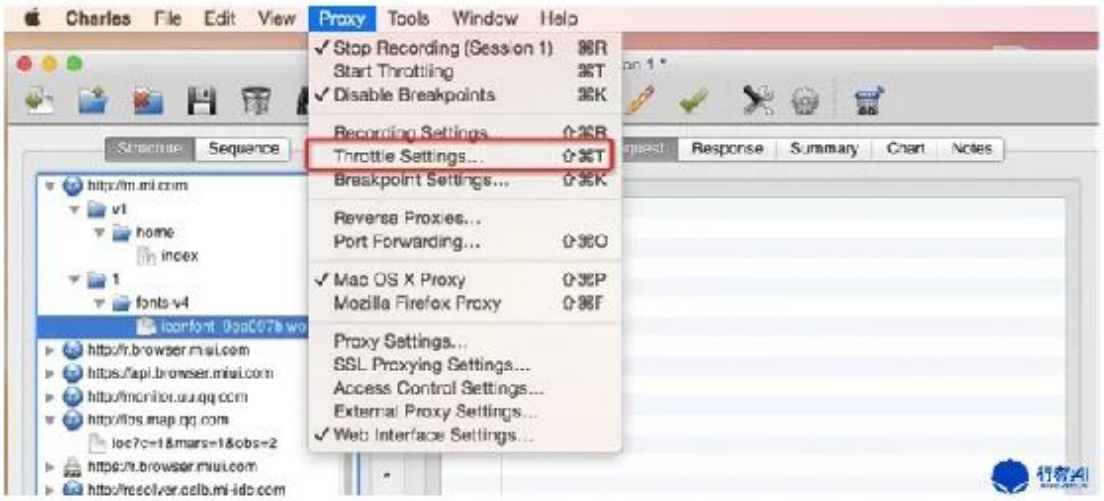
优点：跨平台支持/协议分析能力强/参数灵活配置

缺点：证书安装复杂/不支持丢包模拟

方案二：专用 WiFi 基站（ATC/Wetest-WiFi）

技术架构：

Linux 流量控制→Web 控制界面→多设备独立测试



进行常用的网速模拟设置

适用场景：

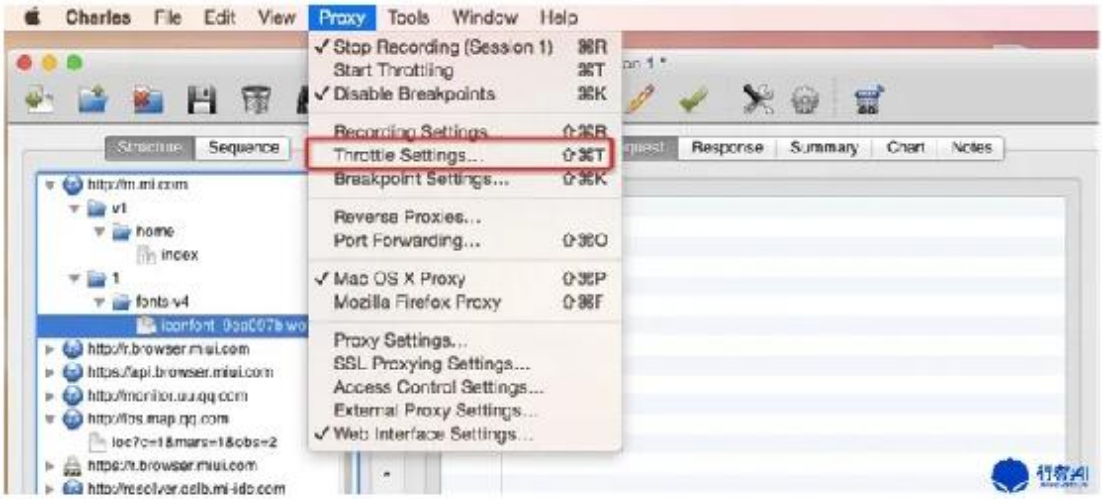
- 多设备并行测试需求
- 复杂网络参数组合验证
- 长期弱网回归测试

方案三：独立 APP 工具（推荐方案）

移动端直接部署：

以独立 app 的方式，为用户提供弱网络模拟服务。

Android：比如腾讯 wetest 服务平台推出的弱网测试工具 QNET，在 Android 设备上直接安装使用。



进行常用的网速模拟设置

iOS：系统一般自带弱网环境测试，可以通过设置各种网络环境，模拟弱网环境，如 3G，wifi，verybadNetwork 或者自定义网络环境进行测试。

在苹果手机的设置如下：设置>开发者选项>Status:on，选择想要测试的网络环境，便可以在该环境下测试 App。

2.工具选型决策矩阵

维度	PC 代理工具	WiFi 基站方案	独立 APP 方案
部署复杂度	★★☆	★☆☆	★★★
参数灵活性	★★★	★★★	★★☆
多设备支持	✘	✔	✘
操作连贯性	★★☆	★☆☆	★★★
移动友好度	★☆☆	★★☆	★★★

四、弱网测试主要关注点

1.功能性“还能正常工作吗？”

关注点：核心功能是否可用，是否会崩溃，数据提交是否成功。

举例说明：

电商下单：弱网下点击“提交订单”，需关注按钮是否防重、是否有加载提示、超时后是否有友好提示、最终是否因重试导致生成重复订单。

发送消息：发送图片或消息时，发送失败是否有明确提示（如红色感叹号），并提供重试选项，避免同一内容重复发送。

2.响应与反馈“用户知道发生了什么吗？”

关注点：界面是否有加载状态提示，操作失败是否有清晰提示。

举例说明：

视频加载：视频卡顿时是显示“加载中”的动画，还是完全卡死？失败后是静默失败还是提示“点击重试”？

用户登录：点击登录后按钮应变为“登录中...”并禁用。超时应提示“网络缓慢”，彻底失败应提示“失败，请检查网络后重试”。

3.健壮性与容错性“出现异常能自己处理好吗？”

关注点：是否妥善处理超时、断线、数据不完整等异常，重试机制是否合理。

举例说明：

大文件下载：应支持断点续传，网络恢复后能从断点继续下载，而不是从头开始。

数据同步：弱网下多设备编辑同一条数据，网络恢复后应有冲突解决机制（如提示用户选择版本），而不是静默覆盖。

4.性能与用户体验“用起来卡不卡？费不费流量？”

关注点：页面加载时间、操作完成时间、流量消耗、电量消耗。

举例说明：

新闻 App：在 2G 网络下应优先加载文字内容（秒看），图片延迟加载，以提升首屏

加载速度。

视频播放：应支持自适应码率，网速变差时自动从高清切换至标清，以保证流畅播放，而非一直缓冲。

5.一致性“数据会不会错乱？”

关注点：是否会出现数据重复、数据丢失、状态不一致。

举例说明：

点赞功能：弱网下点赞，客户端立即显示红心，但请求未到达服务器。此时取消点赞，若逻辑处理不当，会导致客户端与服务器状态不一致。

余额支付：连续点击支付，若服务端无幂等性校验，可能导致一笔交易被重复扣款。

总之，弱网测试远非“把网速调慢”那么简单。它是一个系统性的测试过程，需要从功能、反馈、容错、性能、数据一致性等多个维度出发，去验证产品在真实世界复杂网络环境下的表现。其最终目标是：即使在恶劣的网络条件下，也能为用户提供尽可能可靠和可预期的体验。

五、弱网测试常见误区

1.误区一：只测“慢”，不测“抖”和“断”

错误：仅模拟高延迟，忽略网络抖动和断线重连。

正解：真实弱网复杂多变，抖动（Jitter）和断线更容易引发深层 Bug（如闪退、花屏、同步死锁）。

2.误区二：只关注前端 App，忽略后端和服务

错误：认为弱网测试只是客户端的事。

正解：弱网问题需前后端联动分析。重点关注服务端的超时设置、重试策略、幂等性设计（防止重复支付、重复下单）。

3.误区三：忽视不同网络协议的差异

错误：用同一套网络参数测试所有功能。

正解：不同协议对弱网反应不同。TCP 抗延迟性强，UDP 怕丢包，HTTP 短连接怕高延迟。需区分场景测试。

4.误区四：不测试边界和极端情况

错误：测试参数过于“温柔”（如只测 5%丢包）。

正解：Bug 常藏在极端场景。必须测试 100%丢包（完全断网）、极高延迟（如 10 秒）、网络频繁剧烈波动下的表现。

5.误区五：忽略用户体验和性能数据

错误：仅凭主观感受（“有点卡”），没有量化数据。

正解：优化需数据支撑。应采集页面加载时间、操作完成时长、流量/电量消耗等指标进行对比分析。

6.误区六：认为自动化测试无法覆盖弱网场景

错误：完全依赖手动测试，效率低下。

正解：可通过工具 API 将网络模拟集成到自动化框架中，对核心流程进行常态化弱网回归测试。

六、弱网测试面试常被问到的题目

1.什么是弱网测试？它的主要目的是什么？

期望答案：弱网测试是模拟低带宽、高延迟、高丢包、抖动等恶劣网络环境，验证应用程序在此类条件下的表现。目的不仅是看功能是否可用，更是评估其稳定性、容错性、用户体验以及是否会发生数据错误。

2.列举几个关键的弱网测试参数，并解释它们的含义。

期望答案：

带宽（Bandwidth）：网络的数据传输速率（如：100kbps）。模拟 2G/3G 等慢速网络。

延迟（Latency）：数据包从发送到接收所需的时间（如：500ms）。模拟物理距离远或路由拥堵。

丢包率（PacketLoss）：传输过程中丢失数据包的百分比（如：10%）。模拟网络不稳定、信号差。

抖动（Jitter）：延迟的变化程度。模拟网络拥堵导致的不稳定延迟，对音视频实时通信影响极大。

3.弱网测试和功能测试、性能测试有什么区别和联系？

期望答案：

区别：功能测试主要在理想网络下验证功能正确性；性能测试（如压力、负载）主要在良好网络下关注系统极限能力；而弱网测试专注于网络异常这一特定维度对应用的影响。

联系：弱网测试是性能测试和稳定性测试的一个子集，同时也需要验证核心功能的正确性，是三者的一种交叉和补充。

4.你通常使用哪些工具进行弱网测试？请简述其用法。

期望答案：

Charles/Fiddler：通过`Proxy>ThrottleSettings`或`Rules>Performance`设置带宽、延迟、丢包率。

优点：简单易用，适合 HTTP/HTTPS 请求。

缺点：对 TCP/UDP 等协议支持弱。

FacebookATC(AugmentedTrafficControl)：功能更强大的网络模拟平台，可模拟预设网络 profile（如“恶劣的 3G”），并通过 Web 界面控制。优点：更专业，可同时对多台设备进行控制。

硬件网络损伤仪：专业硬件，提供最精确和稳定的控制。

优点：精度高，支持所有协议。

缺点：成本高。

开发者选项：AndroidStudio 的虚拟设备管理和 Xcode 的网络链接调节器。

5.请描述一次你进行的完整的弱网测试案例。

期望答案：采用 STAR 原则（情境，任务，行动，结果）回答。

情境(S)：在测试 XX 项目的视频会议功能时。

任务(T)：需要确保用户在网络不稳定的地铁里也能有基本可用的体验。

行动(A)：

1）确定测试场景：视频发起、接收、实时语音、会议中网络变化。

2）选择工具：使用 Charles 模拟高延迟(500ms)+高抖动(100ms)+5%丢包。

3）执行测试：

功能：会议能否成功建立？断线后能否自动重连？

体验：视频是否降级为音频？是否有“网络不佳”的提示？音频是否卡顿、碎音？

数据：是否因重试导致多次发送入会请求？

4）记录缺陷：发现了网络剧烈抖动时客户端闪退的问题。

结果(R)：提交了闪退 Bug，并验证了修复。给出了视频卡顿时间的量化数据，推动了优化排期。

6.弱网环境下，如何判断一个问题是前端问题还是后端问题？

期望答案：

前端问题迹象：UI 无响应、无加载提示、无错误提示、本地数据错乱、点击无效。

后端问题迹象：接口请求超时（但前端处理得当）、日志发现重复请求、数据库产生重复或脏数据、服务端逻辑超时。

排查方法：抓包（看请求是否发出、响应是否返回）、查看客户端和服务端日志（比对时间戳和请求 ID）。

7.在弱网测试中，你发现过一个最有价值的 Bug 是什么？为什么它有价值？

期望答案：考察实际经验和问题严重性判断。

示例：“在测试支付流程时，模拟支付请求高延迟后超时，客户端自动重试了 3 次。但由于服务端没有做幂等性校验，导致用户一笔交易被扣了 3 次款。这个 Bug 直接关系到资金安全，是 P0 级别的缺陷。”

价值：体现了对数据一致性和后端逻辑的深度测试，而非停留在表面 UI 问题。

8.针对一个即时通讯 App（如微信）的‘发消息’功能，你会设计哪些弱网测试用例？

期望答案：

消息发送过程中网络中断，检查恢复后是否自动重发。

消息发送超高延迟（如 10 秒），检查客户端是否显示“发送中”状态，能否取消发送。

连续快速发送多条消息，在弱网下检查消息顺序是否错乱。

发送图片/文件时高丢包，检查是部分失败还是全部失败，是否有进度条提示。

模拟网络切换（Wi-Fi->4G），检查消息发送是否会中断。

9.什么是幂等性？为什么它在弱网测试中至关重要？请举例说明。

期望答案：

定义：幂等性是指一次请求和多次请求对系统资源造成的副作用是相同的。

重要性：弱网下客户端极易因超时而重试请求。如果服务端接口不幂等，就会导致重复下单、重复支付、重复点赞等严重问题。

例子：“支付接口必须设计为幂等的。通常通过使用唯一的交易号（Token）来实现，服务器收到相同交易号的请求时，只会处理一次。”

拓展学习

[3] 解锁 APP 测试试听课，获取移动端测试实战包

咨询：微信 atstudy-js 备注：11

swagger 生成接口文档

◆ 作者：刚子

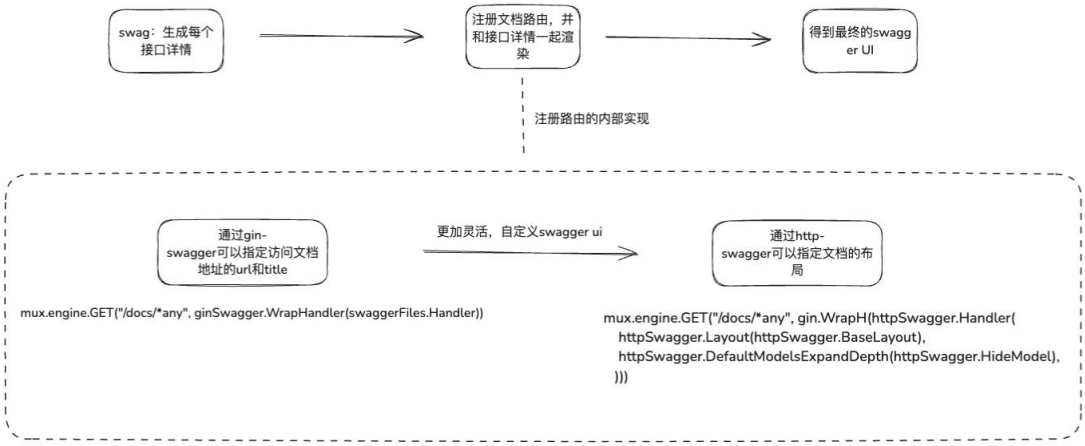
背景

本文适用于 go 语言基于 gin 框架为例，使用 gin-swagger 库以及 Swagger2.0 自动生成 RESTful API 文档，其他语言可参考官方文档。

目标

能够满足我们输出文档的需求（用于后续测试断言），又能随代码的变更自动更新，保持文档和代码的一致性。

原理



步骤

一般需要下面三个步骤：

1.程序入口 main 函数和 controller 层上以注释的方式添加

• main:项目相关介绍信息（描述、版本、path、安全认证等）

• @description.markdown，支持读取 markdown 文件渲染，默认读取 api.md 文件，如果需要支持 markdown，则创建 api.md，具体可参考文档。

```
//  @description 接口文档
// @version 1.0
//
// @description 接口文档
// @description.markdown
//
// @host example.com
// @BasePath /v1
func main(){
...
}
```

• controller：具体的接口函数（协议、router、入参和返回）

示例：支持安全认证：// @Security Bearer，和 main 函数的对应，不添加会失效，具体可参考文档。

注释中参数类型使用了 object, entity.ExampleParam： 格式为包名.struct 名称。

• 在具体的结构体 struct 上添加注释（//开头）,可以在生成文档中的 Model 标签中可以看到：

```
// @Summary 接口总结
// @Description 接口描述
// @Tags 接口分类
// @Accept json
// @Produce json
```

```
// @Param object body entity.ExampleParam true "请求参数"
// @Success 200 {object} entity.ExampleResponse "返回内容"
// @Failure 500 {object} entity.ErrorResponse
// @Router /example [post]
func (h *handler) Example() core.HandlerFunc {
...
}
// 请求参数
type ExampleParam struct {
  ...
}
```

2. 编写完注释后，使用命令生成文档数据（后续会集成到 makefile 工具链中）

执行下面命令后，会在 app 目录下新增 docs 文件夹。

```
// 安装 swag 工具
go install github.com/swaggo/swag/cmd/swag@latest
// 和 main 函数同级?录
cd src/app
// 会读取 src/app 目录下的 api.md 文件进生成 API 整体的介绍信息
swag init --md .
// 在 main 函数文件引入
import _ "app/docs"
./docs
├── docs.go
├── swagger.json
└── swagger.yaml
```

3. 引入 gin-swagger 渲染文档数据

• 在项目代码中注册路由

•把项目运行起来，打开浏览器访问 http://{your_host}:{your_port}/docs/index.html 就能看到 Swagger 2.0 Api 文档了。

Q&A

Q1：同一个 api，针对参数不同，均是正常响应，如何支持两种输出结构？

A：选取一个 struct 可以覆盖两种输出结构，在特定返回字段上添加注释，解释下该字段在什么情况下会返回，哪些情况下不会返回

Q2：swagger 注册的路径是什么，*any 代表什么意思？

A：可以自定义，go-gin 默认支持 /docs/*any 路由，*any 代表后面任意的路径都可以支持

Q3：swagger 文档中的【Try it out】按钮，请求的是哪个地址？

A：入口处 @host 注解指定的地址

实战

良好书写的 swagger 注释，可以有效的生成跟随代码变更的接口描述文档。我们在书写 Go 代码的 swagger 注释时，可以简单的将注释分为三大部分：

- 主函数注释
- API 接口注释
- 结构体注释

主函数注释

主函数注释的书写位置为 main 函数之上，为应用的描述信息。

主函数可以加的注释种类很多，具体可参考 https://github.com/swaggo/swag/blob/master/README_zh-CN.md#%E9%80%9A%E7%94%A8api%E4%BF%A1%E6%81%AF

一个例子

上面的主函数信息，会在生成的 swagger 文档的首页展示：

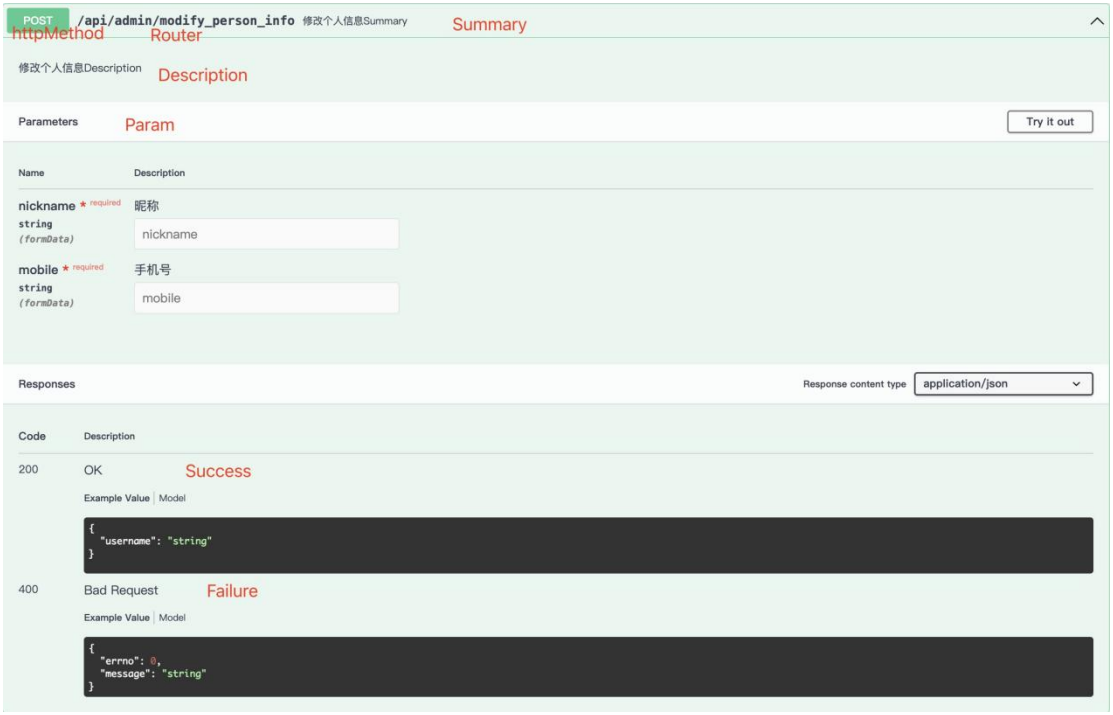
API 接口注释

api 接口注释为描述每个 API 接口的信息，下面是较常见的注释。

一个例子：

```
// ModifyPersonalInfo 修改个人信息
// @Summary 修改个人信息 Summary
// @Description 修改个人信息 Description
// @Tags API.admin, Admin, Modify
// @Accept multipart/form-data
// @Produce json
// @Param nickname formData string true "昵称"
// @Param mobile formData string true "手机号"
// @Success 200 {object} modifyPersonalInfoResponse
// @Failure 400 {object} code.Failure
// @Router /api/admin/modify_person_info [post]
```

生成后的 swagger 文档中接口信息与上述注释的对应：



tags 标签

tags 可以简单的理解为 API 接的分组，注意下列几点：

- tags 区分大小写
- 一个接口可以设置多个 tags，这样就会出现在多个分组中

下面是 swagger 文档中的 tags 分组展示举例：

router 里的 httpMethod

常见的 httpMethod 有很多，建议只使用 GET 和 POST。

MIME 类型

swag 接受所有格式正确的 MIME 类型，还接受某些 MIME 的别名。

返回

返回目前有成功（success）和失败（failure）两种情况，写法：

```
// @Success    200    {object}    entity.ImagesInPaintResponse    "返回内容"
// @Failure    400    "参数错误，可能是部分必传参数缺失，部分参数类型错误"
// @Failure    401    "鉴权失败"
// @Failure    429    "频次限制"
```

参数

参数解释是注释中非常重要的环节，这里单独来讲述。

参数的注释格式如下：

```
// @param 参数名称 参数类型 数据类型 是否必传 注释 属性 (可选)
```

数据类型

常用数据类型有：

- string
- integer
- number（浮点数）
- boolean
- 自定义结构体

需要注意的是，swagger 对于数据类型没有做强限制，我们也可以使用一些见名知意

的类型，例如 UUID，email，password 等。

属性

属性一般描述参数的限制，例如长度、大小等。

举例：

```
参数列表
// @Param enumstring    query    string        false    "string enums"          Enums(A, B, C)
// @Param enumint       query    int           false    "int enums"             Enums(1, 2, 3)
// @Param enumnumber    query    number        false    "int enums" Enums(1.1, 1.2, 1.3)
// @Param string        query    string        false    "string valid" minlength(5)    maxlength(10)
// @Param int           query    int           false    "int valid" minimum(1)      maximum(10)
参数为一个结构体
// @Param          object body          entity.ImagesInPaintParam          true      "请求参数"
```

结构体注释

先看一个例子：

```
// Sex
// @description 性别：0 男 1 女
type Sex int
const (
 Male Sex = iota
 Female
)
type detailData struct {
 RaceName string `json:"className"` // 比赛名称，必填
 Score int `json:"score" minimum:"1" maximum:"100"` // 得分，必填
 IsPass bool `json:"isPass"` // 是否通过
}
type detailResponse struct {
 Username string `json:"username" maxLength:"11" minLength:"3"` // 用户名
```

```
  Nickname string `json:"nickname"` // 昵称
  Mobile string `json:"mobile" format:"1[3|56789][0-9]"` // 手机号,
new
  Sex Sex
  Data detailData `json:"data"` // 比赛数据
}
```

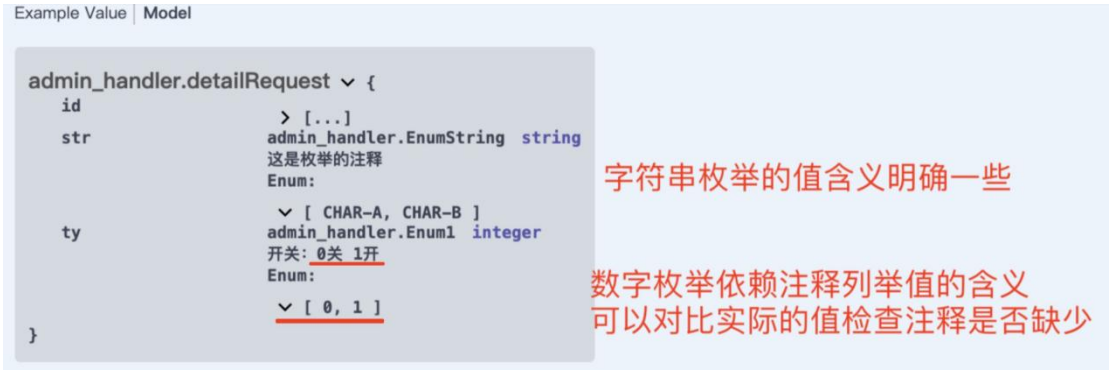
对应 swagger 的显示：

枚举

数字枚举

字符串枚举

枚举的值含义名明确：



拓展学习

[4] 领取【自动化测试全链路】资料包

咨询：微信 atstudy-js 备注：11

从零到工程化:用 GitHub Actions 高效集成 Playwright 测试

◆作者：TestCraft

在现代软件开发流程中，自动化测试的真正价值往往需要通过与 CI/CD 流水线的集成才能发挥。因为只有当测试成为每次代码提交的“守门员”，自动执行并快速反馈时，它才真正发挥了作用。

GitHub Actions 作为与代码仓库原生集成的 CI/CD 工具，凭借其简洁的 YAML 配置、丰富的生态市场和强大的并行处理能力，已经成为无数开发团队的首选。

本文章会带你完成如何通过 Github Action 去运行你的 Playwright 测试，并且完成并发测试和上传报告，最终打造一个高效的现代化 CI 测试体系。

实战演练：Playwright 与 GitHub Actions 集成

废话不多说，那我们就开始了。

这就带大家在 Github 中运行 Playwright，最终目标是实现：在每次发起 Pull Request (PR) 时自动运行测试，并将详细的测试报告发布到 GitHub Pages 以便查看。

Step1 初始化测试框架和测试脚本

在这篇文章中，使用 Node.js 版本的 Playwright 进行演示

首先，通过以下命令可以初始化 Nodejs 的 Playwright 测试框架:

```
npm install --save-dev @playwright/test # Playwright install
npm init playwright@latest
```

其中，npm init playwright@lates 命令运行后会引导我们搭建一个测试框架的脚手架。

```
$ npm init playwright@latest
Getting started with writing end-to-end tests with Playwright:
Initializing project in '.'
√ Do you want to use TypeScript or JavaScript? · TypeScript
√ Where to put your end-to-end tests? · tests
√ Add a GitHub Actions workflow? (y/N) · true
√ Install Playwright browsers (can be done manually via 'npx playwright install')? (Y/n) · true
```

Step2 让测试在 Actions 上跑起来

要在 GitHub Actions 中实现自动运行测试，我们需要完成以下几个步骤：

1.在 .github/workflows/ 目录下创建一个 playwright.yml 工作流文件。

2.使用 on 关键字定义触发时机（例如 push 和 pull_request 事件）。

3.设置一个 job，让它在 GitHub 托管的 Runner 上运行。

4.在 job 的步骤中调用 Playwright 命令来执行测试。

以下是简单的示例

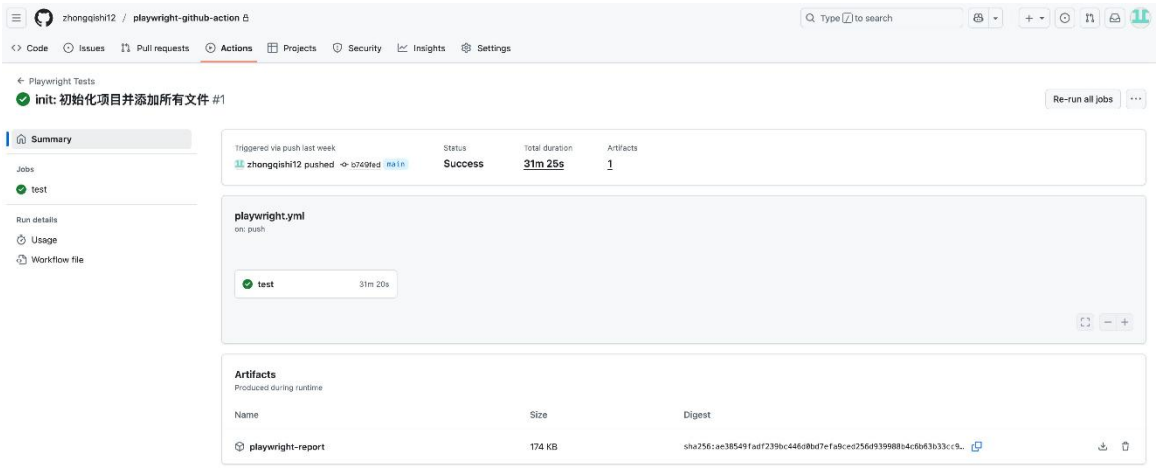
```
name: Playwright Tests
on:
  push:
    branches: [ main, master ]
  pull_request:
    branches: [ main, master ]
jobs:
  test:
    timeout-minutes: 60
    runs-on: ubuntu-latest
    steps:
    - name: Checkout repository
      uses: actions/checkout@v4
    - name: Set up Node.js
      uses: actions/setup-node@v4
      with:
```

```
      node-version: '20'
  - name: Install dependencies
    run: npm install

  - name: Install Playwright browsers
    run: npx playwright install --with-deps
  - name: Run Playwright tests
    run: npx playwright test

  - uses: actions/upload-artifact@v4
    if: ${{ !cancelled() }}
    with:
      name: playwright-report
      path: playwright-report/
      retention-days: 30
```

提交后 Job 运行成功并且生成了测试报告！



Step3 并发测试

在上一步中我们只是把测试跑起来了。但随着项目规模的增长，自动化测试用例的数量也会快速增加。想象一下：如果每次代码合并都需要运行几十分钟甚至更久，不仅会延迟部署，还会严重影响开发团队的工作效率。

那么，当然就需要使用并发测试了！

Playwright 的并发原理

Playwright 采用 基于 worker 的架构：

- 每个测试文件在一个独立的 worker 进程中运行；
- 默认情况下，worker 的数量 = 机器 CPU 的核心数；
- 也就是说，机器的计算资源越多，Playwright 并发的潜力就越大。

但是这里有个坑：GitHub Actions 默认使用的 ubuntu-latest Runner，只提供 2 个 vCPU。

如果直接在这里要求 Playwright 启动 8 个并发进程，就会出现 CPU 资源不足，进程之间不断争抢，最终导致执行时间不降反升。

解决方案：矩阵 + 分片（Matrix + Sharding）

那一台服务器不够，那我们就用多个小机器，每个机器只跑一部分测试。

1.修改 playwright.config.ts

首先 ，我们通过重新配置 Playwright，在 CI 环境下指定并发数并启用文件级并行：

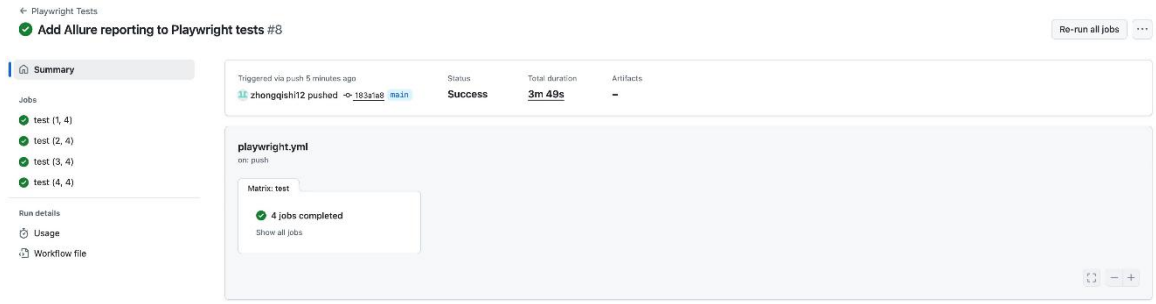
```
import { defineConfig } from '@playwright/test';
export default defineConfig({
  testDir: './tests',
  timeout: 60000,
  retries: 2,
  fullyParallel: true,
  reporter: [['html', { outputFolder: 'playwright-report' }]],
  workers: process.env.CI ? 8 : 2, // 本地 2 个进程
  // 确保 fullyParallel 为 true，以允许文件级的并行
  fullyParallel: true,
});
```

1.修改 .github/workflows/playwright.yml

在 workflow 中添加 strategy: matrix 来启动多个 Job，并让 Playwright 使用“分片”(--shard)功能。

```
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      # 当一个 job 失败时，不要取消其他 job
      fail-fast: false
      # 启动 4 个并行的 job
      matrix:
        shardIndex: [1, 2, 3, 4]
        shardTotal: [4]
    steps:
    # ... (安装依赖等步骤)
    - name: Run Playwright tests
      run: npx playwright test --shard=${{ matrix.shardIndex }}/${{ matrix.shardTotal }}
```

提交后可以看到测试被分到了 4 个 Job 中运行。



Step4 合并测试报告

在完成了分布式的运行后，接着又出现了一个问题。每个 job 都会各自生成一份报告，报告就会很零散。

使用的解决方法： 使用 Allure 报告，在所有的测试运行完成后进行合并。

1.安装 Allure 依赖

首先我们在项目中安装 allure-playwright

```
npm install --save-dev allure-playwright
npm install --save-dev allure-commandline
```

然后在 playwright.config.ts 里加上 reporter 配置：

```
import { defineConfig } from '@playwright/test';
export default defineConfig({
  reporter: [
    ['list'],  // 控制台输出
    ['allure-playwright'], // 生成 allure 结果文件
  ],
});
```

在 job 中上传 allure-results，添加以下步骤

```
- name: Upload allure-results
  uses: actions/upload-artifact@v4
  with:
    name: allure-results-${{ matrix.shardIndex }}
    path: allure-results
```

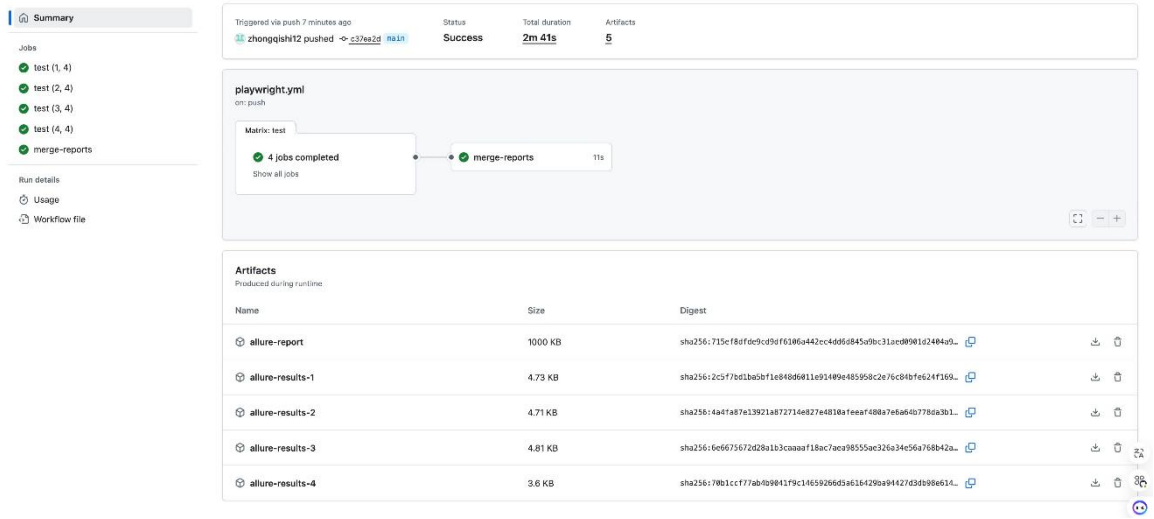
1.合并结果并生成报告

在所有分片完成后，新增一个 merge-reports job：

```
merge-reports:
  runs-on: ubuntu-latest
  needs: test
  steps:
    - uses: actions/download-artifact@v4
      with:
        path: ./allure-results
```

```
        # 下载所有 shard 的 allure-results，存放到同一目录下
      # 解压所有分片结果（每个 artifact 默认会打包成 zip）
        - name: Extract all allure-results
          run: |
            mkdir -p merged-results
            for d in ./allure-results/*; do
              echo "Looking into $d"
              zipfile=$(find "$d" -name "*.zip")
              echo "Extracting $zipfile"
              unzip -q "$zipfile" -d merged-results
            done
    - name: Set up Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '20'
    - name: Install Allure CLI
      run: npm install -g allure-commandline
    - name: Generate Allure report
      run: npx allure generate ./merged-results --clean -o ./allure-report
    - name: Upload Allure report
      uses: actions/upload-artifact@v4
      with:
        name: allure-report
        path: ./allure-report
```

执行完成后，Github Action 会生成一个最终的 allure-report 目录，并作为 artifact 上传，你就可以直接下载并查看完整的合并报告。



总结

通过本文的实战演练，我们完成了一个完整的现代化测试流水线，具体包括以下内容：

1.从零起步：快速初始化 Playwright 测试框架。

2.基础集成：让测试在 GitHub Actions 中自动运行。

3.并发优化：利用矩阵 + 分片提升执行效率。

4.结果合并：借助 Allure 报告实现分布式结果的合并与可视化，最终形成可追溯、可共享的测试报告。

完整项目请访问

https://github.com/bridgeshi85/playwright-github-action-demo

拓展学习

[5] 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

常见性能测试漫游与探索

◆ 作者：启航

前言：

本文是对于性能测试的一些漫游和探索。

常见的性能测试命令有一些有关内存，CPU，网速，磁盘，温度，cache 等等的指标，但是我们并不清楚， 收集日志的 2>&1 之类的命令是如何影响性能的，还有用户态和系统态是否与各种排序算法的时间复杂度相关，这个文章就初步的分析了一些用户态，系统态和时间复杂度的关联，得到一些猜想性质的结论，由于时间原因，这次分析还没有非常多的理论数据，有时间的话我将继续收集更多信息和数据进行分析。

(1)内存的使用情况，以及内存泄漏的知识

我们知道内存中的 free 一般指的是还没有被占用的物理内存，也就是说是空闲的内存，就好像一个人的空闲时间，或者说是一个人脑容量中可以继续活跃的部分，是动态的。

free 越小，则表示可以使用的物理内存越小，used 是已经在使用中的物理内存，也就是说非常忙碌的内存，当系统非常忙碌的时候，free 就会减少。

命令： free -h

```
-IdeaPad-U430p:~$ free -h
              total        used        free      shared  buff/cache   available
Mem:           3.8G        939M        1.9G        210M        1.0G        2.4G
Swap:          2.0G          0B        2.0G
```

如果 free 的数量持续减少，而 used 的数量持续增多，当程序不再运行的时候，内存依旧被占据着，则可以初步判定为是内存泄漏。

以下命令也是可以查看内存的

cat /proc/meminfo

```
MemTotal:        3957496 kB
MemFree:          835788 kB
MemAvailable:    2443988 kB
Buffers:         1454848 kB
Cached:           532996 kB
SwapCached:         2996 kB
Active:          2111668 kB
Inactive:         717676 kB
Active(anon):     583240 kB
Inactive(anon):   436040 kB
Active(file):    1528428 kB
Inactive(file):   281636 kB
Unevictable:       39524 kB
Mlocked:              16 kB
SwapTotal:       2097148 kB
```

(2)有关 CPU 的性能测试

命令：top

```
top - 21:40:45 up 37 min,  2 users,  load average: 0.00, 0.00, 0.05
Tasks: 247 total,   1 running, 201 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  3957496 total,  1905216 free,   962932 used,  1089348 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  2547292 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 4573 [redacted]  20   0   51320   4012   3348 R   0.3  0.1   0:00.09 top
    1 root      20   0  225788   9428   6588 S   0.0  0.2   0:02.16 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    3 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 rcu_par_gp
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:0H-kb
    8 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_wq
    9 root      20   0       0      0      0 S   0.0  0.0   0:00.05 ksoftirqd/0
   10 root      20   0       0      0      0 I   0.0  0.0   0:00.66 rcu_sched
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.03 migration/0
   12 root     -51   0       0      0      0 S   0.0  0.0   0:00.00 idle_inject/0
   13 root      20   0       0      0      0 I   0.0  0.0   0:00.29 kworker/0:1-eve
   14 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/0
   15 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
   16 root     -51   0       0      0      0 S   0.0  0.0   0:00.00 idle_inject/1
   17 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/1
   18 root      20   0       0      0      0 S   0.0  0.0   0:00.04 ksoftirqd/1
   20 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/1:0H-kb
   21 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/2
   22 root     -51   0       0      0      0 S   0.0  0.0   0:00.00 idle_inject/2
   23 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/2
   24 root      20   0       0      0      0 S   0.0  0.0   0:00.05 ksoftirqd/2
   26 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/2:0H-kb
   27 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/3
   28 root     -51   0       0      0      0 S   0.0  0.0   0:00.00 idle_inject/3
   29 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/3
   30 root      20   0       0      0      0 S   0.0  0.0   0:00.05 ksoftirqd/3
   32 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/3:0H-kb
   33 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   34 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 netns
   35 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_tasks_kthre
   36 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kauditd
   37 root      20   0       0      0      0 S   0.0  0.0   0:00.00 khungtaskd
```

• us 用户空间（User）进程占用 CPU 的时间百分比

用户态与用户进程的关系

用户态是操作系统的一种执行模式，用户进程（如普通应用程序）运行时处于该状态。

• sy 内核（System）占用 CPU 的时间百分比，就好比我们常常说的一个人是不是内核很稳定，sy 数值越大可以想象为内核越稳定（这是作者自己的想象）

系统态（内核态）

*定义：CPU 执行系统程序（如内核代码、驱动程序、中断处理程序等）时的状态，拥有访问所有硬件资源的权限。

*功能：负责资源管理（如进程调度、内存分配）、设备控制（如硬盘读写、网络通信）等核心任务。

系统进程与系统态的区别：

*定义：系统运行的程序实例，如操作系统自身或用户启动的应用程序。 ？

*状态：可能处于用户态（普通程序）或内核态（如执行系统级操作时）。 ？

• ni 改变过优先级进程占用 CPU 百分比

• id 空闲 CPU 占比

• wa io 等待占用 CPU 占比

• hi 硬件中断占用 CPU 时间

• si 软件中断占用 CPU 时间

• PID 进程 id

• User 进程所有者

• PR 动态优先级（Linux 值），?数值越小优先级越高

• NI 进程的静态 Nice 值，负值为高优先级，正值为低优先级

• %CPU 进程占用 CPU 百分比

• %MEM 进程占用物理内存百分比

• Time 进程的 CPU 时间的总计

• Command 进程的名称

(3)内存中，有关 Buff，cache 的思想

```
[redacted]-IdeaPad-U430p:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:           3864         919        1750         198        1195        2516
Swap:          2047           0        2047
```

Buff 展示了缓冲区高速缓存的大小,而 cache 一列展示了页缓存大小,选项-m 把结果以 MB 为单位进行显示。——引用自 P327《性能之巅》

(4)有关网速的测试

查网速

命令：cat /proc/net/dev

是 Linux 系统中用于 ?实时查看所有网络接口的详细统计信息

查找接收和发送的字节数、包数、错误数等

```
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
wlp2s0: 12749717   10466    0    0    0     0          0         0   582486    3743    0    0    0     0       0          0
enp1s0:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
    lo:   20695     213    0    0    0     0          0         0    20695     213    0    0    0     0       0          0
```

命令：ifconfig

是 Linux 系统中用于 ?配置和显示网络接口信息? 的核心命令，其功能涵盖接口状态查看、IP 地址设置、启停网卡等操作

命令： sudo ethtool 网卡的名称 | grep Speed

查看网络配置，如果是 1000Mb/s?：表示网卡协商为千兆速率

(5)磁盘的测试

命令: df -h

主要用于查看磁盘空间容量

```
Filesystem      Size  Used Avail Use% Mounted on
udev            1.9G     0  1.9G   0% /dev
tmpfs           387M  2.2M  385M   1% /run
/dev/sda2       110G  8.7G   95G   9% /
tmpfs           1.9G     0  1.9G   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           1.9G     0  1.9G   0% /sys/fs/cgroup
/dev/loop2      9.5M  9.5M     0 100% /snap/gnome-characters/805
/dev/loop3       67M   67M     0 100% /snap/core24/988
/dev/loop5       56M   56M     0 100% /snap/core18/2855
/dev/loop1      128K  128K     0 100% /snap/bare/5
/dev/loop6      896K  896K     0 100% /snap/gnome-logs/129
/dev/loop0       15M   15M     0 100% /snap/gnome-characters/296
/dev/loop7       55M   55M     0 100% /snap/core18/1066
/dev/loop8       43M   43M     0 100% /snap/gtk-common-themes/1313
/dev/loop10      92M   92M     0 100% /snap/gtk-common-themes/1535
/dev/loop9      4.2M  4.2M     0 100% /snap/gnome-calculator/406
/dev/loop11     150M  150M     0 100% /snap/gnome-3-28-1804/67
/dev/loop13     1.0M  1.0M     0 100% /snap/gnome-logs/61
/dev/loop12     105M  105M     0 100% /snap/core/17210
/dev/loop15     3.8M  3.8M     0 100% /snap/gnome-system-monitor/100
/dev/loop14     1.8M  1.8M     0 100% /snap/gnome-system-monitor/193
/dev/loop16     2.2M  2.2M     0 100% /snap/gnome-calculator/972
/dev/sda1       511M  6.1M  505M   2% /boot/efi
tmpfs           387M   16K  387M   1% /run/user/121
tmpfs           387M   36K  387M   1% /run/user/1000
tmpfs           387M     0  387M   0% /run/user/0
```

(6)负载的模拟

命令 1:【系统态】free 变小（系统进程占用的?CPU?时间百分比更多， 含有内存泄漏）

sudo dd if=/dev/sda2 of=/dev/null bs=1024k count=2000

该命令使用?dd?工具从?/dev/sda2?设备读取数据，每次读取 1024KB 的数据块，总共读取 2000 个数据块，然后将这些数据写入?/dev/null?设备。/dev/null?是一个特殊的设备文件，它会丢弃所有写入其中的数据，因此这个命令实际上不会在?/dev/null?中创建任何输出文件。

命令参数解释

- if=/dev/sda2：指定输入文件为?/dev/sda2?设备。
- of=/dev/null：指定输出文件为?/dev/null?设备。
- bs=1024k：设置块大小为 1024KB。
- count=2000：指定读取和写入的块数为 2000。

命令参考自 P406《性能之巅》

检测命令： vmstat 1

检测结果

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 850516 1122904 1018456    0    0     0     0   38   61  0  0 100  0  0
 0  0      0 850516 1122904 1018456    0    0     0     0   62   68  0  0 100  0  0
 0  0      0 850516 1122904 1018456    0    0     0     0   39   61  0  0 100  0  0
 0  0      0 850516 1122904 1018456    0    0     0     0   28   36  0  0 100  0  0
 0  0      0 850516 1122904 1018456    0    0     0    24   42   54  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0    12   56   61  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     0   42   64  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     0   52   81  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     0   42   63  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     0   62   87  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     0   69  157  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     0   30   44  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     0   40   55  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     0   45   82  0  0 100  0  0
 0  0      0 850516 1122912 1018456    0    0     0     4   59   94  0  0 100  0  0
 0  0      0 850516 1122912 1018460    0    0     0     0   39   59  0  0 100  0  0
 0  0      0 850516 1122912 1018460    0    0     0     0   48   59  0  0 100  0  0
 0  0      0 850516 1122912 1018460    0    0     0     0   70   86  0  0 100  0  0
 0  0      0 850516 1122912 1018460    0    0     0     0   38   52  0  0 100  0  0
 0  0      0 850516 1122912 1018460    0    0     0     0   63  127  0  0 100  0  0
 0  0      0 850516 1122916 1018460    0    0     0     4   81  118  0  0 100  0  0
 0  0      0 850516 1122916 1018460    0    0     0     0   41   54  0  0 100  0  0
 0  0      0 850516 1122916 1018460    0    0     0     0   30   44  0  0 100  0  0
 0  0      0 850516 1122916 1018460    0    0     0     0   59  108  0  0 100  0  0
 0  0      0 850516 1122916 1018460    0    0     0     0   52  109  0  0 100  0  0
 1  0      0 848964 1122916 1018460    0    0     0     0   98  141  0  1 99  0  0
 0  1      0 437040 1534052 1019376    0    0 411136     0 1028 1302  0 10 87  3  0
 1  0      0 117588 1916196 957628     0    0 468480     0 1083 1583  0 10 87  3  0
 0  0      0 117204 1916876 957604     0    0 119808     0  386  551  0  3 96  1  0
 0  0      0 117212 1916876 957604     0    0     0     0   44   59  0  0 100  0  0
```

分析：

从上图我们可以看到，id:空闲 CPU 百分比明显减少，从 100 变为了 87 左右。

Sy 明显增大，既是内核空间占用 CPU 百分比也是系统进程占用的 CPU 时间百分比。

Us 不变，保持 0，us 是用户空间未改变优先级的进程占用 cpu 的百分比。

cs 和 in 都增大到了 1000 左右。

cs 和 in 分别代表每秒上下文切换次数和每秒中断次数。

bi 明显增大，bi 列表示从块设备（如硬盘）读取的数据块数，单位是块/秒??（磁盘 IO）。

bo 表示每秒写入到块设备的数据块数（磁盘 IO）。

wa 表示 IO 等待的时间。

free 变小，free 是可用物理内存的大小。

free 变小的时候我们可以想象为 “内核稳定”，因为 sy 增大而 us 不变，因此我们可以想象为这种情况是内核稳定。(此为作者想象)

如果程序运行之后 free 又恢复了，说明不是内存泄漏，但是我们发现这个命令运行之后内存没有恢复，我觉得这还是有一些内存泄露的问题的。

以下是内存泄漏的判定

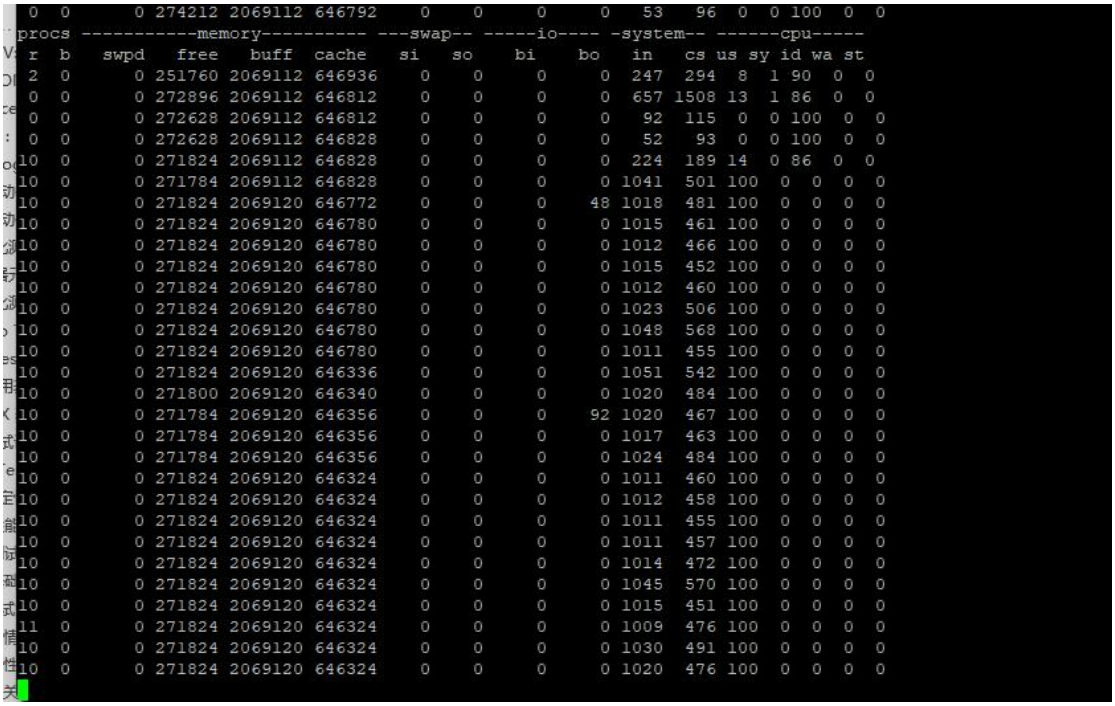
在编程和软件开发中，内存泄漏是指程序中不再使用的内存没有被正确释放，导致这部分内存一直占用，进而可能导致内存耗尽，影响程序的性能或稳定性。

命令 2:【用户态】free 不变的压力测试 （用户进程使用时间变多）

测试当 cpu 数量过多时候的压力测试情况

stress -c 10 --timeout 100

以上使用了 10 个 cpu 核心，但是实际上的 cpu 只有 4 个，容易导致系统不稳定。



```
 0  0      0 274212 2069112 646792    0    0     0     0   53   96  0  0 100  0  0
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  0      0 251760 2069112 646936    0    0     0     0  247  294  8  1 90  0  0
 0  0      0 272896 2069112 646812    0    0     0     0  657 1508 13  1 86  0  0
 0  0      0 272628 2069112 646812    0    0     0     0   92  115  0  0 100  0  0
 0  0      0 272628 2069112 646828    0    0     0     0   52   93  0  0 100  0  0
10  0      0 271824 2069112 646828    0    0     0     0  224  189 14  0 86  0  0
10  0      0 271784 2069112 646828    0    0     0     0 1041  501 100  0  0  0  0
10  0      0 271824 2069120 646772    0    0     0    48 1018  481 100  0  0  0  0
10  0      0 271824 2069120 646780    0    0     0     0 1015  461 100  0  0  0  0
10  0      0 271824 2069120 646780    0    0     0     0 1012  466 100  0  0  0  0
10  0      0 271824 2069120 646780    0    0     0     0 1015  452 100  0  0  0  0
10  0      0 271824 2069120 646780    0    0     0     0 1012  460 100  0  0  0  0
10  0      0 271824 2069120 646780    0    0     0     0 1023  506 100  0  0  0  0
10  0      0 271824 2069120 646780    0    0     0     0 1048  568 100  0  0  0  0
10  0      0 271824 2069120 646780    0    0     0     0 1011  455 100  0  0  0  0
10  0      0 271824 2069120 646336    0    0     0     0 1051  542 100  0  0  0  0
10  0      0 271800 2069120 646340    0    0     0     0 1020  484 100  0  0  0  0
10  0      0 271784 2069120 646356    0    0     0    92 1020  467 100  0  0  0  0
10  0      0 271784 2069120 646356    0    0     0     0 1017  463 100  0  0  0  0
10  0      0 271784 2069120 646356    0    0     0     0 1024  484 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1011  460 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1012  458 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1011  455 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1011  457 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1014  472 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1045  570 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1015  451 100  0  0  0  0
11  0      0 271824 2069120 646324    0    0     0     0 1009  476 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1030  491 100  0  0  0  0
10  0      0 271824 2069120 646324    0    0     0     0 1020  476 100  0  0  0  0
```

我们看到 us 增大，sy 不变， cs 和 in 增加，但是 free 不变，这说明此时内核空间（sy）并没有使用，就不用动用”真气”， 也就是说 free 不变。（以上是作者的想象）

我们可以看到 r 的数值变大， 而 r 代表的是进程等待的数量。

bi，bo 和 wa 几乎为 0，说明没有 IO 的交换和 IO 等待。

用户进程使用过多是不会造成内存泄露的，因为内存不会变少。

但是系统进程使用过多，有可能造成内存泄漏，也有可能不会有内存泄漏，以下的例子就没有造成内存泄漏。

命令 3:【系统态】free 变小 (系统进程占用的?CPU?时间百分比更多，没有内存泄漏)

而以下是动用 “真气”的命令，运行之后，free 会变小的。

stress -i 1 --hdd 1 --timeout 100

检测命令： vmstat 1

解释

--hdd 1：这个选项用于指定硬盘压力测试的权重。数字 1 表示硬盘操作的相对权重。

这意味着在压力测试中，硬盘 I/O 将占据一定的比重。

```
                                    stress -i 1 --hdd 1 --timeout 100
stress: info: [6917] dispatching hogs: 0 cpu, 1 io, 0 vm, 1 hdd
```

发现运行完成之后 free 是会恢复的 ，所以没有内存泄漏。（非常明显的恢复）

(7)系统温度的测试

cat /sys/class/thermal/thermal_zone0/temp

返回值为整数，当前温度-摄氏度是此数值的千分之一?（如?44000 表示 44°C）

```
44000
```

(8)查 CPU 数量和频率

查询有多少个逻辑 cpu:

cat /proc/cpuinfo | grep "processor" | wc -l

```
4
```

查询 cpu 频率命令：

cat /sys/devices/system/cpu/cpu2/cpufreq/scaling_cur_freq

```
796563
```

(9)查看磁盘 IO 性能的统计命令

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.00    0.00   13.86    2.48    0.00   83.66

Device            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
loop0            0.00         0.00         0.00          0          0
loop1            0.00         0.00         0.00          0          0
loop2            0.00         0.00         0.00          0          0
loop3            0.00         0.00         0.00          0          0
loop4            0.00         0.00         0.00          0          0
loop5            0.00         0.00         0.00          0          0
loop6            0.00         0.00         0.00          0          0
loop7            0.00         0.00         0.00          0          0
sda            539.00    367628.00         0.00     367628          0
loop8            0.00         0.00         0.00          0          0
loop9            0.00         0.00         0.00          0          0
loop10           0.00         0.00         0.00          0          0
loop11           0.00         0.00         0.00          0          0
loop12           0.00         0.00         0.00          0          0
loop13           0.00         0.00         0.00          0          0
loop14           0.00         0.00         0.00          0          0
loop15           0.00         0.00         0.00          0          0
loop16           0.00         0.00         0.00          0          0
loop17           0.00         0.00         0.00          0          0
```

每隔 1 秒输出一次系统当前的 CPU 使用率及磁盘 I/O 性能统计信息。

从上图我们可以看到，每秒读取的块数的 kb 数量 367628kB

tps （表示该设备每秒传输的 IO 请求的数量）数量是 539

(10)清空内存和缓存的情况

清空缓存 －引用自 P337《性能之巅》

echo 3 > /proc/sys/vm/drop_caches

清除内存缓存以释放资源之后， 再次运行命令，buff 会有一个递增的趋势

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 730624 1846080 413640    0    0   282    16   22   40  2  0 98  0  0
 0  0      0 730624 1846080 413640    0    0     0     0   37   43  0  0 100  0  0
 0  0      0 730608 1846080 413676    0    0     0     0   44   67  0  0 100  0  0
 0  0      0 730608 1846088 413676    0    0     0    20   41   61  0  0 100  0  0
 0  0      0 730608 1846088 413676    0    0     0     0   33   50  0  0 100  0  0
 0  0      0 730608 1846088 413676    0    0     0     0   39   49  0  0 100  0  0
 0  0      0 730608 1846088 413676    0    0     0     0   46   76  0  0 100  0  0
 0  0      0 2578280    376 410296    0    0   104     0  131  173  0  4 96  0  0
 0  0      0 2577776    924 410296    0    0   548     0   94  334  0  0 100  0  0
 0  0      0 2577776   1068 410296    0    0   144     0   66  207  1  0 99  0  0
 0  1      0 2378932 198948 410976    0    0 198516     0  675 1798  0  3 89  8  0
 0  1      0 2105800 473952 410468    0    0 471552     0 1076 1882  0  9 88  4  0
 1  0      0 1639068 939604 410628    0    0 466500   156 1016 1493  0  8 89  4  0
 1  0      0 1169796 1408368 411652    0    0 467968     0  977 1419  0  7 89  3  0
 0  0      0 722244 1854984 412904    0    0 446748    24 1024 1533  0  8 89  3  0
 0  0      0 722512 1854984 413040    0    0     0     0   55   91  0  0 100  0  0
 0  0      0 722512 1854984 413040    0    0     0     0   42   55  0  0 100  0  0
 0  0      0 722512 1854984 413040    0    0     0     0   30   43  0  0 100  0  0
^C
```

(11)自己编写脚本查看性能

1.系统态与用户态的 shell 脚本

以下是我自己编写的 linux 脚本，循环增加 100 万次

```
#!/bin/bash
#set +e
#set -x
sum=1
while [ $sum -lt 1000000 ]
do

        sum=$(($sum + 1))
        echo "count number" $sum
done
```

命令 **vmstat 1**

发现， sy 和 us 都是有变大的趋势，us > sy，但是 free 变小不是很明显，内存没有明显的占用，in 和 cs 都有明显的变大趋势，id 明显变小，说明占用的 cpu 非常多。

进程等待数量也变多了，IO 等待-wa 数量不变。

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 2316828   6636 582724    0    0    52   846  124  253 29  2 68  1  0
 1  0      0 2272428  10508 600540    0    0 21236   428 2613 6580 16  5 77  2  0
 1  0      0 2073536  13868 744732    0    0 76040  6788 2605 6210  8  8 78  6  0
 1  0      0 2099516  14352 748196    0    0  3808     0  772 1213 22  2 76  0  0
 1  0      0 2074296  14352 748340    0    0     0     0  532  475 24  1 75  0  0
 4  0      0 2212920  14356 677732    0    0   180     4  642  722 23  3 75  0  0
 3  0      0 2211528  14360 678864    0    0  1048     0 5692 154530 55 18 27  0  0
 3  1      0 2210408  14368 678920    0    0     0   128 4795 165763 59 18 23  0  0
 1  0      0 2209264  14372 679004    0    0    88     0 5312 170882 56 17 27  0  0
 2  0      0 2209088  14372 679100    0    0     0     0 3657 255402 34 15 52  0  0
 2  0      0 2208108  14372 679180    0    0     0     0 2848 286916 33 16 51  0  0
 2  0      0 2208436  14372 679100    0    0     0     0 2791 268037 31 15 54  0  0
 3  0      0 2204672  14372 679100    0    0     0    12 3947 253099 52 19 30  0  0
 3  0      0 2068704  14940 749536    0    0   528  5232 3741 218778 50 22 25  3  0
 3  0      0 2100308  14944 749592    0    0     4     0 4745 220126 58 17 25  0  0
 3  0      0 2079000  14944 749592    0    0     0     0 5308 202192 58 16 26  0  0
 3  0      0 2073448  14944 749592    0    0     0     0 6675 183456 60 18 22  0  0
 3  0      0 2211336  14944 680916    0    0     0     4 5797 175303 56 18 26  0  0
 3  1      0 2211576  14948 680820    0    0     0   120 6909 178648 59 16 25  0  0
 3  0      0 2211040  14952 680856    0    0     0     4 5846 174272 57 20 23  0  0
 3  0      0 2211616  14952 680712    0    0     0     0 6891 186745 59 18 23  0  0
 3  0      0 2209420  14952 680712    0    0    72     0 5053 225607 48 17 35  0  0
 0  0      0 2209688  14952 681004    0    0     0     0 1619 103581 12  5 83  0  0
 1  0      0 2193080  14952 682624    0    0     0  1732  672  839  9  4 86  0  0
 0  1      0 2190292  16296 688428    0    0   848 10948 1553 5941 13  6 75  6  0
 1  0      0 2092980  16336 757604    0    0     0  2068 1013 2093 21 10 70  0  0
 1  0      0 2067508  16336 757748    0    0     0     0  418  356 25  1 75  0  0
 1  0      0 2134452  16336 755812    0    0     0     4  686  682 22  3 75  0  0
 1  0      0 2203424  16336 687292    0    0     0   268  416  235 23  2 75  0  0
 1  0      0 2203612  16336 687100    0    0     0     0  360   88 23  3 75  0  0
 1  0      0 2203580  16348 687308    0    0   332   128  330  151 23  2 75  0  0
 0  0      0 2203308  16348 687496    0    0     0     0  452 1256  4  1 95  0  0
^C
```

2.系统态的 shell 脚本

```
#!/bin/bash
set +e
set -x
mkdir -p dir
while true
do
        rmdir -p dir
        mkdir -p dir
        if [ $? -eq 0 ]; then
                echo "Directory changed successfully."
```

```
        else
                echo "Failed to change directory."
        fi
Done
```

发现， sy 有明显的变大的趋势，free 有明显的变小趋势， in 和 cs 都有明显的变大趋势, id 明显变小，说明占用的 cpu 非常多。

进程等待数量维持在 1，说明几乎没有什么进程等待。

运行结束之后，内存没有恢复，因此这个程序存在内存泄漏的现象。

运行 命令 echo 3 > /proc/sys/vm/drop_caches 之后，内存明显的恢复了。

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 2392832   2544 511892    0    0    92   202  120  426 11  2 87  0  0
 0  0      0 2392832   2552 511912    0    0     0    24   53   60  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   28   43  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   47   68  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   57  144  1  0 99  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   34   56  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   48   90  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   39   55  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   32   55  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   29   45  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   30   44  0  0 100  0  0
 0  0      0 2392832   2552 511912    0    0     0     0   57  103  1  0 100  0  0
 1  0      0 2388004   4904 512620    0    0  1248     0 2049 11071  6 17 77  0  0
 1  0      0 2385220   7220 512596    0    0     0     0 2544 14350  9 21 70  0  0
 1  0      0 2383448   9660 512452    0    0     0     0 2563 14676  9 24 67  0  0
 1  0      0 2381192  12096 512492    0    0     0   300 2444 14415  9 23 69  0  0
 1  0      0 2378164  14512 512616    0    0     0     0 2634 14487  9 22 69  0  0
 1  0      0 2377108  15684 512480    0    0     0    64 2583 14423  8 25 67  0  0
 1  0      0 2376048  16708 512584    0    0     0     0 2610 14553  8 24 68  0  0
 1  0      0 2375900  18032 512632    0    0     0     0 2619 14445  8 22 69  0  0
 2  0      0 2373068  20520 512476    0    0     0     0 2606 14779  8 23 69  0  0
 1  0      0 2371124  22852 512700    0    0     0     0 2696 14393  8 21 71  0  0
 1  0      0 2371108  22884 512560    0    0     0    60 2757 14799  9 20 71  0  0
 1  0      0 2370124  23028 512568    0    0   148     0 2878 15900  8 23 69  0  0
 1  0      0 2369572  23028 512576    0    0     0     0 2485 14553  8 24 68  0  0
 1  0      0 2369824  23028 512584    0    0     0    32 2794 15768  9 22 69  0  0
 1  0      0 2369804  23028 512592    0    0     0     0 2685 15375  8 24 68  0  0
 2  0      0 2369804  23064 513188    0    0     0    68 2461 14699  9 24 67  0  0
 0  0      0 2370612  23068 513204    0    0     4     0 1509 8674  6 12 81  0  0
 0  0      0 2369372  24892 513524    0    0  2080     0  235 1156  0  1 99  0  0
 0  0      0 2369356  24892 513540    0    0     0   152   59  106  0  0 100  0  0
 0  0      0 2369412  24892 513480    0    0     0     0   76   81  0  0 100  0  0
 0  0      0 2369412  24908 513480    0    0     0    76   46   63  0  0 100  0  0
 0  0      0 2369412  24908 513480    0    0     0     0   34   51  0  0 100  0  0
 0  0      0 2369412  24908 513480    0    0     0     0   31   53  0  0 100  0  0
 0  0      0 2369412  24908 513484    0    0     4     0   87   97  0  0 100  0  0
 0  0      0 2369412  24908 513484    0    0     0     0   40   95  0  0 99  0  0
 0  0      0 2369412  24908 513484    0    0     0    60   50   61  0  0 100  0  0
 0  0      0 2369412  24908 513484    0    0     0     0   29   47  0  0 100  0  0
```

3.纯用户态的 shell 脚本

```
#!/bin/bash
#set +e
#set -x
#纯用户态
while true
do
        sum=$(($sum + 1))
Done
```

(echo 命令是系统态的，将前面第一个 shell 脚本中的 echo 命令取消之后，剩下的就都是用户态的，从这个例子来看我们发现计算类的命令是用户态的，没有内存泄漏， 用 pidstat 命令检查，几乎都是非自愿上下文切换）

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 2049196 122288 747980    0    0     5    11   19   54  1  0 99  0  0
 1  0      0 2049200 122288 747980    0    0     0     0  335   83 25  0 75  0  0
 1  0      0 2049200 122288 747980    0    0     0     0  289   53 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0    12  304   77 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  291   49 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  312   79 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0    64  302   59 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  288   47 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  325   96 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  339  107 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  329  163 26  0 74  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  283   51 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  295   63 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  311   71 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  284   49 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  279   47 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  297   64 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  298   54 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  294   73 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  280   45 25  0 75  0  0
 1  0      0 2049200 122296 747980    0    0     0     0  293   59 25  0 75  0  0
 2  0      0 2049200 122296 747980    0    0     0     0  297   56 25  0 75  0  0
 1  0      0 2049200 122296 747984    0    0     0    16  297   44 25  0 75  0  0
 1  0      0 2049200 122296 747984    0    0     0     0  292   61 25  0 75  0  0
 1  0      0 2049200 122296 747984    0    0     0     0  300   87 25  0 75  0  0
 1  0      0 2049200 122296 747984    0    0     0     0  302   69 25  0 75  0  0
 1  0      0 2049200 122296 747984    0    0     0     0  294   54 25  0 75  0  0
 1  0      0 2049200 122304 747976    0    0     0    20  295   57 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  325  152 26  0 74  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  302   65 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  298   72 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  282   49 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  295   64 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  334   87 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  288   44 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  284   51 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  299   65 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  305   65 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  284   43 25  0 75  0  0
 1  0      0 2049200 122304 747984    0    0     0     0  283   45 25  0 75  0  0
 0  0      0 2049200 122304 747984    0    0     0     0  328  213 19  0 81  0  0
 0  0      0 2049208 122304 747984    0    0     0     0   83  112  0  0 100  0  0
```

4.有关各种收集日志信息的对于系统态还是用户态的分析

【系统态和用户态都较少】-以下命令将 2>&1 放在最后，就是将不管是错误输出还是标准输出都保存在 log.txt 中，前台不显示日志。

```
[redacted]# rm 1.txt > log.txt 2>&1
[redacted]# cat log.txt
rm: cannot remove '1.txt': No such file or directory
[redacted]# ls > log.txt 2>&1
[redacted]# cat log.txt
log.txt
test2.sh
test3_kernal.sh
test3.sh
test_kernal.sh
test_last_kernal.sh
test.py
test.sh
test_user.sh
test_user_simulate.sh
```

以下是只存储 log 的 shell 脚本

```
#!/bin/bash
while true
do
        ls > log.txt 2>&1
Done
```

以下是只存储 log 内容的性能情况

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  0      0 2010916 163176 746988    0    0     5    11   20   63  1  0 99  0  0
 1  0      0 2009812 163176 746988    0    0     0  4868 2287 7742  8 19 73  0  0
 1  0      0 2010504 163176 746992    0    0     0  4532 2202 7235  7 20 73  0  0
 1  0      0 2009524 163184 746992    0    0     0  4504 2155 7207  7 19 73  0  0
 1  0      0 2009552 163184 746996    0    0     0  4608 2246 7481  9 20 71  0  0
 1  0      0 2009280 163184 746988    0    0     0  4748 2253 7556  7 20 73  0  0
 1  0      0 2010604 163184 746996    0    0     0  4700 2227 7523  9 18 73  0  0
 1  0      0 2009868 163184 746984    0    0     0  4780 2237 7591  8 18 74  0  0
 1  0      0 2009552 163192 746996    0    0     0  4752 2316 7552  6 25 69  0  0
 1  0      0 2009300 163192 746992    0    0     0  4624 2346 7370  7 25 68  0  0
 1  0      0 2009476 163192 746992    0    0     0  4620 2357 7421  8 25 67  0  0
 2  0      0 2009768 163192 746992    0    0     0  4824 2232 7767  9 18 73  0  0
 1  0      0 2010056 163192 746988    0    0     0  4760 2224 7676  9 17 74  0  0
 1  0      0 2009272 163200 746988    0    0     0  4848 2239 7602 10 16 74  0  0
 1  0      0 2009212 163200 746996    0    0     0  4832 2256 7632  9 18 73  0  0
 1  0      0 2009544 163200 746988    0    0     0  4872 2271 7735  9 18 73  0  0
 1  0      0 2009532 163200 746996    0    0     0  4860 2305 7790  9 21 70  0  0
 2  0      0 2009504 163200 746988    0    0     0  4416 2289 7056  7 26 68  0  0
 1  0      0 2009496 163208 746984    0    0     0  4784 2245 7678  8 20 72  0  0
 1  0      0 2009348 163208 746988    0    0     0  4788 2294 7723  9 20 71  0  0
 1  0      0 2009588 163208 746988    0    0     0  4748 2242 7573  8 19 73  0  0
 1  0      0 2009532 163208 746992    0    0     0  4628 2232 7394  8 20 72  0  0
 1  0      0 2009252 163208 746992    0    0     0  4848 2286 7739  8 20 72  0  0
 1  0      0 2009524 163216 746992    0    0     0  4836 2332 7735  9 21 70  0  0
 1  0      0 2010764 163216 746992    0    0     0  4808 2252 7668  9 18 73  0  0
 1  0      0 2010820 163216 746988    0    0     0  4320 2121 6982  7 19 73  0  0
 1  0      0 2010504 163216 746988    0    0     0  4544 2238 7251  8 22 70  0  0
 1  0      0 2010792 163216 746996    0    0     0  4392 2332 7039  6 28 66  0  0
 1  0      0 2010836 163224 746988    0    0     0  4532 2196 7174  8 20 72  0  0
 2  0      0 2009776 163224 746988    0    0     0  4468 2195 7120  7 21 72  0  0
 1  0      0 2010292 163224 746992    0    0     0  4728 2243 7607  8 20 72  0  0
 1  0      0 2010812 163224 746988    0    0     0  4616 2294 7370  8 23 69  0  0
 1  0      0 2010664 163224 746992    0    0     0  4856 2290 7728 10 19 71  0  0
 1  0      0 2008776 163232 747064    0    0     0  4628 2209 7418  7 20 73  0  0
 1  0      0 2009316 163232 747024    0    0     0  4732 2258 7518  8 20 72  0  0
 2  0      0 2009320 163232 747020    0    0     0  4856 2315 7717  9 20 71  0  0
 1  0      0 2008940 163232 747020    0    0     0  4660 2314 7486  7 23 70  0  0
 1  0      0 2010252 163232 747020    0    0     0  4420 2221 7081  8 22 70  0  0
 1  0      0 2009788 163240 746980    0    0     0  4720 2361 7559  7 24 69  0  0
 1  0      0 2009028 163240 746984    0    0     0  4724 2264 7533  9 19 72  0  0
 1  0      0 2010372 163240 746992    0    0     0  4272 2113 6841  9 18 73  0  0
 1  0      0 2010084 163240 746992    0    0     0  4424 2137 7054  7 19 74  0  0
 1  0      0 2009020 163240 746984    0    0     0  4428 2138 7141  8 18 74  0  0
 1  0      0 2009540 163248 746984    0    0     0  4680 2192 7519  7 19 74  0  0
 1  0      0 2008544 163248 747004    0    0     0  4712 2267 7680  9 20 71  0  0
 1  0      0 2010100 163248 746992    0    0     0  4756 2244 7648  9 18 73  0  0
 1  0      0 2009840 163248 746992    0    0     0  4672 2191 7539  9 19 72  0  0
 1  0      0 2009628 163248 746992    0    0     0  4500 2176 7243  7 20 72  0  0
 1  0      0 2010092 163256 746984    0    0     0  4548 2283 7249  8 24 69  0  0
 1  0      0 2010044 163256 746992    0    0     0  4560 2366 7294  8 26 66  0  0
 1  0      0 2010036 163256 746996    0    0     0  4620 2349 7392  7 26 67  0  0
 1  0      0 2010044 163256 746992    0    0     0  4412 2287 7117  8 25 68  0  0
 1  0      0 2009808 163256 746996    0    0     0  4232 2321 6807  6 29 65  0  0
```

【系统态与用户态显著增加】-以下命令使用 command 2>&1 | tee log.txt 则无论是错误输出还是标准输出，都在前台以及 log.txt 中显示。

```
[redacted]# ls 2>&1 | tee log.txt
log.txt
test2.sh
test3_kernal.sh
test3.sh
test_kernal.sh
test_last_kernal.sh
test.py
test.sh
test_user.sh
test_user_simulate.sh
[redacted]# cat log.txt
log.txt
test2.sh
test3_kernal.sh
test3.sh
test_kernal.sh
test_last_kernal.sh
test.py
test.sh
test_user.sh
test_user_simulate.sh
[redacted]# rm 1.txt 2>&1 | tee log.txt
rm: cannot remove '1.txt': No such file or directory
[redacted]# cat log.txt
rm: cannot remove '1.txt': No such file or directory
```

以下是前台输出并且同时存储 log 内容的 shell 脚本

```
#!/bin/bash
while true
do
        ls 2>&1 | tee log.txt
Done
```

以下是前台输出并且同时存储 log 内容的性能情况【系统态与用户态显著增加】

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  0      0 2010892 163336 747000    0    0     5    12   20   64  1  0 99  0  0
 2  0      0 2010640 163336 747016    0    0     0  4408 3759 22079 17 33 51  0  0
 2  0      0 2010672 163336 747032    0    0     0  4424 3830 22274 14 36 50  0  0
 1  0      0 2010336 163344 747028    0    0     0  4408 3854 22235 16 34 50  0  0
 3  0      0 2008964 163344 747032    0    0     0  4416 3726 22205 13 36 50  0  0
 2  0      0 2009320 163344 746992    0    0     0  4408 3836 22272 14 36 50  0  0
 2  0      0 2010372 163344 746996    0    0     0  4420 3837 22264 16 34 50  0  0
 1  0      0 2009956 163344 746996    0    0     0  4368 3766 22159 16 35 49  0  0
 2  0      0 2009992 163352 746992    0    0     0  4384 3721 22056 13 37 51  0  0
 2  0      0 2009636 163352 746996    0    0     0  4408 3800 22305 14 35 51  0  0
 2  0      0 2009476 163352 746992    0    0     0  4412 3886 22213 16 33 51  0  0
 2  0      0 2010272 163352 747000    0    0     0  4412 3842 22130 15 35 50  0  0
 2  0      0 2010252 163352 747000    0    0     0  4400 3866 22388 15 34 51  0  0
 2  0      0 2010292 163360 747000    0    0     0  4432 3835 22352 14 35 50  0  0
 3  0      0 2010060 163360 747000    0    0     0  4332 3853 21999 15 34 51  0  0
 2  0      0 2009988 163360 746996    0    0     0  4384 3804 22218 14 35 51  0  0
 3  0      0 2010292 163360 747000    0    0     0  4416 3869 22199 13 37 50  0  0
 1  0      0 2010708 163360 747000    0    0     0  4416 3815 22347 15 35 50  0  0
 2  0      0 2010040 163368 747000    0    0     0  4420 3799 22241 14 36 50  0  0
 3  0      0 2009572 163368 747000    0    0     0  4416 3855 22272 15 35 50  0  0
```

5.有关各种排序算法是系统态还是用户态的猜想

我们通常知道各种不同的排序算法的时间复杂度来说是不同的，但是我们除了依据时间复杂度来查看各种不同的排序算法，我们是否可以用性能测试的指标来区别不同的复杂度的排序算法呢？

<第一部分 前置调查>

1)调查直接运行排序算法的 python 是否会增加 us 或者 sy 的数值

问：直接运行排序的 python 脚本是否可以

答：

同时运行排序算法的 python 脚本以及 vmstat 检查 us 或者 sy 的数值是否会变化

Python 举例，循环 100000 次

```
for count in range(100000):
    <中间是排序算法>
```

在 python 脚本中去掉 print 的内容，则 sy 的数值可以一直保持为 0（print 内容会增加 sy 的数值），仅 us 含有数值，这种方法虽然清晰，但是会导致运行时 us 保持不变，无法检测出来 us，sy 变动的区别。因为 us 为固定数值，sy 一直是 0。

所以不直接运行 python 的排序算法。

2)调查持续不断地用 shell 脚本调用排序算法的 python 是否会增加 us或者 sy 的数值

对于四种排序，如果希望用 shell 调用 python，并且 sy 有数值，发现 python 中排序列表长度不能过长，过长的话也会导致 sy 为 0，us 不变。因为稍微短一些的列表长度可以触发调用系统态，同时，用户态还能有一定的变动幅度，我们这里定义 python 中排序列表长度为 1000。

3)调查 vmstat 与 2>&1 | tee log.txt 的结合是否会增加 us 或者 sy 的数值

结论是 ./test.sh 2>&1 | tee log.txt 是不会影响 sy 和 us 的数值的，sy 和 us 一直都保持为 0

（test.sh 里面内容是 vmstat 1 40 -t）

<第二部分 开始运行>

注意，运行每一种排序之前都运行一下以下的清除缓存并释放资源的命令

echo 3 > /proc/sys/vm/drop_caches

以下持续不断地运行某种排序，比如：冒泡排序，每一种排序的元素个数都是 1000 个元素排序，并且持续运行 40 秒收集 sy 和 us 的数据，收集方法为收集 vmstat 1 40 -t 所列数据：

1：冒泡排序

计算冒泡排序的 us（用户态）的平均值 （去掉最开始的异常值）

1000 个元素：23.923

计算冒泡排序的 sy（系统态）的平均值 （去掉最开始的异常值）

1000 个元素：3.717

2：选择排序

计算选择排序的 us（用户态）的平均值 （去掉最开始的异常值）

1000 个元素：22.58

计算选择排序的 sy（系统态）的平均值 （去掉最开始的异常值）

1000 个元素：7.10

3：插入排序

计算插入排序的 us（用户态）的平均值 （去掉最开始的异常值）

1000 个元素：23.53

计算插入排序的 sy（系统态）的平均值 （去掉最开始的异常值）

1000 个元素：4.17

4： Shell 排序

计算 shell 排序的 us（用户态）的平均值 （去掉最开始的异常值）

1000 个元素：19.64

计算 shell 排序的 sy（系统态）的平均值 （去掉最开始的异常值）

1000 个元素：7.51

5： 总结四种排序

算法时间复杂度的概念

时间复杂度是执行程序需要的时间，算法的时间复杂度（Time Complexity）是衡量算法执行效率的重要指标，它表示算法运行时间随输入规模增长的变化趋势，通常用大 O 符号（O-notation）表示。

从用户态分析四种排序的复杂程度

从前置调查中，我们知道用户态是程序运行所真正涉及的 CPU 部分，通过运行我们可以看到如下的排序结果：

Shell （19.64） ＜ 选择 （22.58）＜ 插入 （23.53）＜ 冒泡（23.923）

这个排序结果与其程序的复杂程度的排序非常接近。

因为使用时间复杂度 O(n2) 这个数值不是很清晰究竟哪个程序更为复杂，而使用本文的方法，可以将程序的复杂程度显示得更为精确。

并且多次运行结果都是一致的，所以这个结果的稳定性还是比较高的。

6.Vmstat 相关的脚本

(1)收集 vmstat 的信息

```
#!/bin/bash
vmstat 1 40 -t
```

(2)将 vmstat 的 log 信息存储到 excel 中

```
import    xlwt
import re
book=xlwt.Workbook(encoding='utf-8',style_compression=0)
sheet=book.add_sheet('Output',cell_overwrite_ok=True)
a=0
with open('./log.txt','r') as raw:
      # for line in raw:
      lines = raw.readlines()
with open('./log.txt', 'r') as raw:
      for index, line in enumerate(raw):
            if(lines[index].find("--")==-1 and lines[index].find("sy")==-1):
                  text=str(lines[index])
                  result3 = re.sub(r' +'," ", text)
                  time=result3.split(" ")[18]+result3.split(" ")[19]
                  print(result3.split(" ")[14])
                  print(result3.split(" ")[13])
                  sheet.write(a,0,time)
                  sheet.write(a,1,result3.split(" ")[14])
                  sheet.write(a,2,result3.split(" ")[13])
                  a=a+1
book.save('./Output_Vmstat.xls')
```

(3)Excel 作图

```
import pandas as pd
import matplotlib.pyplot as plt
df=pd.read_excel("./Output_Vmstat.xls",sheet_name="Output")
plt.rcParams['font.sans-serif']='SimHei'
plt.rcParams['axes.unicode_minus']=False
x=df.iloc[:, 0]
y=df.iloc[:, 1]
plt.plot(x,y)
```

```
plt.xlabel("time")
plt.xticks(rotation=20)
plt.tick_params(axis='x',labelsize=8)
plt.ylabel("sy")
y_mean = y.mean()
print(y_mean)
plt.title("average"+ str(y_mean), fontsize=12, pad=20)
plt.legend()
plt.show()
```

7.系统态与用户态通常的规律总结

1】一般来说系统态的命令是与操作系统有关的，而用户态的命令是与用户本身有关的命令，（如算术运算、普通内存读写）。

2】用户态的数值排序则近似于复杂度的排序（猜想，可靠性比较高）。

3】使用 command 2>&1 | tee log.txt -前台和 log 都显示错误和标准输出，则大大增加了系统态和用户态的数值。

4】系统态的增加更容易引发内存泄漏。

后记：

这次研究总结了一些常见的 linux 性能测试命令，接着围绕用户态和系统态的数值，对于不同程序进行分析，发现有些程序可以明显增加用户态以及系统态的使用，比如“ command 2>&1 | tee log.txt”。

有一些程序可以明显增加用户态，比如计算类的 sum=$(($sum + 1))。而另外有一些程序明显增加系统态，比如 echo 之类的程序。此次研究比较有价值的结论是找到了一个测定程序复杂程度的方法。

本人试图将用户态和系统态的数值与时间复杂度等常见复杂度指标进行关联，以得出了一个比较可靠稳定的测算程序复杂程度的方法，这个方法是使用 shell 脚本循环调用 python 排序脚本，然后通过 vmstat 1 40 -t 命令，获取 us 的数值的平均值，这个 us 的数值

就可以用来测定程序复杂程度。

测定的检测数据选择方法是， 运行之前需要确保，python 排序脚本不存在 print 内容，shell 脚本不存在 echo 等增加 sy 的信息，且仅仅改变 python 排序脚本的元素个数，就可以使得检测结果的 vmstat 中 sy 变为 0，说明，影响 sy 的因素只有 python 排序脚本的元素个数，然后找到合适的 python 排序脚本的元素个数，确保检测结果的 vmstat 中 sy 不是一直是 0。进行测定。

测定了很多次，目前四种排序每次测定的排序顺序都是一致的。说明稳定性还是比较高的。

测试展望，由于时间复杂度与 sy，us 的数值会影响响应时间，TPS，等数值，以后将结合 Jmeter 进行更多的研究。

拓展学习

[6] 全方位多终端，领高级性能实战包

咨询：微信 atstudy-js 备注：11

。

从自动化测试脚本到框架：循序渐进的 AI 辅助学习路径

◆作者：风落几番

在自动化测试领域，许多初学者感到迷茫，他们不知道如何从编写简单的测试脚本过渡到构建复杂的自动化测试框架。这种困惑源于对自动化测试技能进阶路径的不清晰。本文将为你提供一条清晰的学习路径，帮助你从基础的 Selenium 脚本编写逐步过渡到构建高效的自动化测试框架，并引入 AI 技术来提升测试效率和质量。

为什么从基础脚本开始？

编写自动化测试脚本是自动化测试的基石。只有当你熟练掌握了如何编写脚本，才能更好地理解测试框架的构建和应用。Selenium 作为目前最流行的自动化测试工具之一，提供了强大的 API，能够满足各种复杂的测试需求。通过学习 Selenium 的 API，你可以编写出灵活多样的测试脚本，为后续的框架搭建打下坚实的基础。

如何开始学习？

学习 Selenium API

Selenium 的 API 是编写自动化测试脚本的关键。它提供了丰富的功能，包括但不限于元素定位、表单操作、窗口切换、断言等。通过学习这些 API，你可以实现各种测试场景的自动化操作。

编写完整的 Selenium 脚本

在掌握了 Selenium API 之后，下一步就是编写完整的测试脚本。一个完整的测试脚

本应该包括测试的初始化、测试步骤的执行以及测试结束后的清理工作。通过编写这样的脚本，你可以熟悉整个测试流程，为后续的框架搭建积累经验。

使用单元测试框架整合脚本

单元测试框架如 JUnit（Java）、pytest（Python）等，可以帮助你更好地组织和运行测试脚本。通过将 Selenium 脚本与单元测试框架整合，你可以实现测试的自动化执行、结果的报告以及测试的持续集成。

AI 如何助力学习？

AI 辅助代码生成

AI 工具可以帮助你快速生成 Selenium 脚本的模板代码。通过输入简单的指令或描述，AI 可以生成基本的脚本框架，包括导入必要的库、初始化 WebDriver、定位元素等。这不仅可以提高你的开发效率，还可以帮助你更快地熟悉 Selenium 的 API。

AI 代码优化建议

AI 不仅可以生成代码，还可以提供代码优化的建议。它可以检查你的脚本，指出潜在的问题，并提供改进的方案。例如，AI 可以建议你使用更稳定的选择器、优化测试步骤的顺序、减少冗余代码等。

AI 智能调试

在编写和运行测试脚本的过程中，难免会遇到各种问题。AI 可以提供智能调试功能，帮助你快速定位问题的根源。通过分析错误日志和代码执行路径，AI 可以提供详细的调试信息和解决方案。

实战项目：构建一个简单的自动化测试框架

为了更好地理解如何从简单的脚本升级到自动化测试框架，我们可以通过一个实战

项目来展示这个过程。假设我们需要为一个电商网站构建一个自动化测试框架，以下是构建过程的概述：

项目需求

- 登录功能测试：验证不同用户登录的正确性和错误处理。
- 商品搜索功能测试：验证搜索功能的准确性和性能。
- 购物车功能测试：验证购物车的添加、删除和结算功能。

选择测试驱动方式

根据测试需求，我们选择数据驱动的方式，使用 Excel 文件存储测试数据。

设计框架结构

框架结构包括：

- 公共方法模块：封装常用的测试操作。
- 页面对象模块：封装页面元素和操作。
- 测试用例模块：定义具体的测试用例。
- 测试数据模块：管理测试数据。
- 测试报告模块：生成测试报告。

实现框架功能

以下是框架的部分实现代码：

公共方法模块

```
python
from selenium import webdriver
import time
class CommonMethods:
```

```
    def __init__(self, browser):
        if browser == "IE":
            self.driver = webdriver.Ie()
        elif browser == "Chrome":
            self.driver = webdriver.Chrome()
        else:
            raise ValueError("Unsupported browser")
    def open_url(self, url):
        try:
            self.driver.get(url)
        except Exception as e:
            print("网页加载失败，可能是链接无效或网络问题，请检查网页链接的合法性，并适当重试。")
            raise e
    def find_element(self, locator):
        return self.driver.find_element(*locator)
    def send_keys(self, element, text):
        element.clear()
        element.send_keys(text)
    def click(self, element):
        element.click()
    def close_browser(self):
        self.driver.quit()
```

页面对象模块

```
python
from selenium.webdriver.common.by import By
from common_methods import CommonMethods
class LoginPage:
    def __init__(self, common_methods):
        self.common_methods = common_methods
        self.url = "http://XXXX.com"
        self.username_locator = (By.ID, 'txtLoginCode')
```

```
        self.password_locator = (By.ID, 'txtPwd')
        self.login_button_locator = (By.ID, 'btnLogin')
    def open(self):
        self.common_methods.open_url(self.url)
    def login(self, username, password):
        username_element = self.common_methods.find_element(self.username_locator)
        self.common_methods.send_keys(username_element, username)
        password_element = self.common_methods.find_element(self.password_locator)
        self.common_methods.send_keys(password_element, password)
        login_button = self.common_methods.find_element(self.login_button_locator)
        self.common_methods.click(login_button)
```

测试用例模块

```
python
import unittest
from login_page import LoginPage
from common_methods import CommonMethods
class TestLogin(unittest.TestCase):
    def setUp(self):
        self.common_methods = CommonMethods("IE")
        self.login_page = LoginPage(self.common_methods)
    def test_login_success(self):
        self.login_page.open()
        self.login_page.login("nice_xp", "*****")
        # 添加断言验证登录是否成功
        self.assertTrue(True)  # 示例断言
    def tearDown(self):
        self.common_methods.close_browser()
if __name__ == "__main__":
    unittest.main()
```

运行测试并生成报告

使用 unittest 框架运行测试用例，并生成测试报告。可以使用 HTMLTestRunner 等工具生成详细的 HTML 报告。

总结

在自动化测试领域，数据驱动、关键字驱动和行为驱动这三种驱动思想各有优势，适用于不同的应用场景。每种驱动模式都有其独特的价值和适用范围，因此不存在绝对的优劣之分。选择合适的驱动模式取决于具体的测试需求、项目特点以及团队的技术背景。

• 数据驱动：通过将测试数据从脚本中分离出来，存储在独立的数据文件或数据库中，使得测试脚本更加通用和可复用。这种方法特别适合模块化设计，能够显著降低维护成本，提高测试效率。

• 关键字驱动：进一步扩展了数据驱动的概念，将测试逻辑分解为关键字，每个关键字对应封装的逻辑业务。这种方法使得测试用例更加通用，易于理解和维护，特别适合构建通用的测试框架或平台。

• 行为驱动：是一种新兴的软件工程实践，强调使用自然语言描述测试用例，使得测试用例更加直观、易于理解。BDD 不仅是一种测试方法，更是一种沟通工具，能够将用户故事、需求和测试用例一体化，减少需求传递过程中的误解和浪费。

随着自动化测试技术的不断发展，测试人员需要不断学习和掌握新的技术和方法。从数据驱动开始，逐步优化和完善测试框架设计，最终全面掌握三种驱动模式，是每个测试人员的成长路径。未来，测试领域可能会出现更多更新的设计模式，等待着测试人员去发掘和创造。如果你有任何好的想法或实践经验，欢迎与大家共同讨论，共同推动测试技术的发展。

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