

# UCAgent 开发者手册

v25.11.21-19-g6c5f29f-dirty

## 目录

<b>1 欢迎来到 UCAgent 文档</b>	<b>6</b>
1.1 概述 . . . . .	6
1.2 文档导航 . . . . .	6
1.2.1 开始使用 . . . . .	6
1.2.2 功能介绍 . . . . .	7
1.2.3 定制开发 . . . . .	7
1.2.4 实践案例 . . . . .	7
<b>2 工具介绍</b>	<b>7</b>
2.1 介绍 . . . . .	7
2.1.1 背景 . . . . .	7
2.1.2 UCAgent 是什么 . . . . .	8
2.1.3 能力与目标 . . . . .	8
<b>3 安装</b>	<b>8</b>
3.1 系统要求 . . . . .	8
3.2 安装方式 . . . . .	8
<b>4 使用</b>	<b>9</b>
4.1 快速开始 . . . . .	9
4.2 结果分析 . . . . .	15
4.3 流程总结 . . . . .	16
4.3.1 需要准备的文件 . . . . .	16
4.3.2 做了什么 . . . . .	16
4.3.3 实现的效果 . . . . .	17
<b>5 MCP 集成模式（推荐）</b>	<b>18</b>

5.1 MCP 集成（推荐）集成 Code Agent . . . . .	18
<b>6 直接使用模式</b>	<b>19</b>
6.1 直接使用 . . . . .	19
6.1.1 使用环境变量配置（推荐） . . . . .	19
6.1.2 使用 config.yaml 来配置 . . . . .	20
6.1.3 开始使用 . . . . .	21
6.1.4 直接用 CLI 启动（不经 Makefile） . . . . .	21
6.1.5 常用 TUI 命令速查（直接使用模式） . . . . .	22
6.1.6 常见问题与提示 . . . . .	22
6.1.7 相关文档 . . . . .	23
<b>7 人机协同验证</b>	<b>23</b>
7.1 为什么需要人机协同 . . . . .	23
7.1.1 AI 单次通过率较低的关键阶段 . . . . .	23
7.1.2 AI 执行卡住需要人工解决 . . . . .	24
7.2 协同工作流程 . . . . .	24
7.2.1 基本流程 . . . . .	24
7.2.2 详细操作步骤 . . . . .	24
7.3 强制人工审核模式 . . . . .	26
7.3.1 设置必须人工审核的阶段 . . . . .	26
7.3.2 通过人工审核 . . . . .	26
7.4 权限控制 . . . . .	27
7.5 阶段管理命令 . . . . .	27
7.6 常用交互命令参考 . . . . .	27
7.7 典型应用场景 . . . . .	28
7.7.1 场景 1：关键阶段主动审核 . . . . .	28
7.7.2 场景 2：AI 执行失败后人工修复 . . . . .	29
7.7.3 场景 3：检查进程卡住 . . . . .	29
<b>8 参数说明</b>	<b>30</b>
8.1 参数与选项 . . . . .	30
8.1.1 输入 . . . . .	30
8.1.2 输出 . . . . .	30
8.1.3 位置参数 . . . . .	31
8.1.4 执行与交互 . . . . .	31
8.1.5 配置与模板 . . . . .	32
8.1.6 计划与 ToDo . . . . .	32
8.1.7 ToDo 工具概览与示例给模型规划的，小模型关闭，大模型自行打开 . . . . .	33

8.1.8 外部与嵌入工具 . . . . .	34
8.1.9 日志 . . . . .	35
8.1.10 MCP Server . . . . .	35
8.1.11 阶段控制与安全 . . . . .	35
8.1.12 版本与检查 . . . . .	36
8.1.13 示例 . . . . .	36
<b>9 TUI</b>	<b>39</b>
9.1 TUI (界面与操作) . . . . .	39
9.1.1 界面组成 . . . . .	39
9.1.2 操作与快捷键 . . . . .	40
9.1.3 命令与用法 . . . . .	41
<b>10 FAQ</b>	<b>42</b>
10.1 FAQ . . . . .	42
10.1.1 UC-Agent 闪退, 如何恢复验证流程? . . . . .	42
10.1.2 为什么快速启动找不到 config.yaml/定制流程时找不到 config.yaml? . . . . .	42
10.1.3 运行中如何调整消息窗口与 token 上限? . . . . .	43
10.1.4 文档中的“CK bug”要改吗? . . . . .	43
10.1.5 为什么找不到 WriteTextFile 工具? . . . . .	43
<b>11 模板文件与生成产物</b>	<b>43</b>
11.1 模板文件 . . . . .	43
11.1.1 Guide_Doc . . . . .	44
11.2 生成产物 . . . . .	45
11.2.1 uc_test_report . . . . .	45
11.2.2 unity_test/tests . . . . .	46
11.2.3 unity_test/*.md . . . . .	47
<b>12 定制功能</b>	<b>49</b>
12.1 添加工具与 MCP Server 工具 . . . . .	49
12.1.1 1) 工具体系与装配 . . . . .	50
12.1.2 2) 添加一个新工具 (本地/Agent 内) . . . . .	50
12.1.3 3) 暴露为 MCP Server 工具 . . . . .	52
12.1.4 4) 客户端调用流程 . . . . .	53
12.1.5 5) 生命周期、并发与超时 . . . . .	53
12.1.6 6) 配置策略与最佳实践 . . . . .	53
12.1.7 7) 常见问题排查 . . . . .	54
<b>13 工具列表</b>	<b>55</b>

13.1 基础/信息类 . . . . .	55
13.2 规划/Todo 类 . . . . .	55
13.3 记忆/检索类 . . . . .	56
13.4 测试/执行类 . . . . .	57
13.5 文件/路径/文本类 . . . . .	57
13.6 扩展示例 . . . . .	59
<b>14 工作流 . . . . .</b>	<b>60</b>
14.1 整体流程概览 (11 个阶段) . . . . .	60
14.2 定制工作流 (增删阶段/子阶段) . . . . .	73
14.2.1 原理说明 . . . . .	73
14.2.2 增加阶段 . . . . .	73
14.2.3 减少子阶段 . . . . .	74
14.3 定制校验器 (checker) . . . . .	75
14.3.1 增加 checker . . . . .	75
14.3.2 删除 checker . . . . .	78
14.3.3 修改 checker . . . . .	78
14.3.4 常用 checker 参数 (结构化) . . . . .	79
<b>15 规范生成 . . . . .</b>	<b>85</b>
15.1 工作流介绍 . . . . .	85
15.1.1 应用场景 . . . . .	85
15.1.2 工作流程 . . . . .	85
15.2 使用规范生成流程 . . . . .	87
15.2.1 前置条件 . . . . .	87
15.2.2 快速开始 . . . . .	87
15.2.3 规范生成流程配置说明 . . . . .	88
15.2.4 关键配置项说明 . . . . .	89
15.2.5 GenSpec 输出结构 . . . . .	89
15.2.6 与 MCP Client 协作 . . . . .	90
15.3 高级用法 . . . . .	91
15.3.1 自定义阶段提示词 . . . . .	91
15.3.2 跳过特定阶段 . . . . .	91
15.4 常见问题 . . . . .	92
15.4.1 Q1: 生成的规范不完整怎么办? . . . . .	92
15.4.2 Q2: 如何提高规范生成质量? . . . . .	92
15.4.3 Q3: 支持哪些 HDL 语言? . . . . .	92
15.4.4 Q4: 规范生成流程和默认的验证生成流程有什么区别? . . . . .	92
15.4.5 Q5: 如何复用规范生成流程配置? . . . . .	93

15.4.6 Q6: 规范生成流程能否增量更新规范? . . . . .	93
<b>16 多 UCAgent 并发执行</b>	<b>94</b>
16.1 背景 . . . . .	94
16.2 实现方式 . . . . .	94
16.3 API 模式并发 . . . . .	94
16.3.1 原理说明 . . . . .	94
16.3.2 基本用法 . . . . .	94
16.3.3 使用 Tmux 管理多实例 . . . . .	95
16.4 MCP 模式并发 . . . . .	97
16.4.1 原理说明 . . . . .	97
16.4.2 端口分配策略 . . . . .	98
16.4.3 基本使用流程 . . . . .	98
16.4.4 使用 Tmux 管理 MCP 并发 . . . . .	100
16.4.5 Makefile 自动化示例 . . . . .	101
16.5 配置说明 . . . . .	102
16.5.1 环境变量配置文件 . . . . .	102
16.5.2 Makefile 变量说明 . . . . .	102
16.6 其他 Code Agent 支持 . . . . .	103
16.6.1 Qwen Code . . . . .	103
16.6.2 Claude Code . . . . .	103
16.6.3 Gemini CLI . . . . .	103
16.6.4 VS Code Copilot . . . . .	104
16.7 故障排查 . . . . .	104
16.7.1 端口被占用 . . . . .	104
16.7.2 Code Agent 连接失败 . . . . .	104
16.7.3 tmux 会话丢失 . . . . .	105
16.7.4 内存不足 . . . . .	105
16.8 相关文档 . . . . .	105
<b>17 批处理执行模式</b>	<b>105</b>
17.1 概述 . . . . .	105
17.2 核心特性 . . . . .	106
17.2.1 1. 自动退出机制 . . . . .	106
17.2.2 2. 自动继续机制 . . . . .	106
17.2.3 3. 任务编排 . . . . .	106
17.3 使用方式 . . . . .	106
17.3.1 API 模式批处理 . . . . .	106
17.3.2 MCP 模式批处理 . . . . .	109

17.4 其他 Code Agent 支持 . . . . .	113
17.4.1 Claude Code . . . . .	113
17.4.2 Gemini CLI . . . . .	114
17.4.3 Qwen Code . . . . .	114
17.5 配置提示词 . . . . .	114
17.5.1 默认提示词 . . . . .	114
17.5.2 自定义提示词 . . . . .	115
17.5.3 环境变量方式 . . . . .	115
17.6 故障排查 . . . . .	116
17.6.1 问题 1: 任务未自动退出 . . . . .	116
17.6.2 问题 2: Hook 未触发 . . . . .	116
17.6.3 问题 3: 批处理中断 . . . . .	117
17.6.4 问题 4: 结果文件冲突 . . . . .	117
17.7 示例项目 . . . . .	118
17.8 相关文档 . . . . .	119

# 1 欢迎来到 UCAgent 文档

本文档提供了安装、使用和开发 UCAgent 的全面指南。

## 1.1 概述

**UCAgent** 是一个基于大语言模型的自动化硬件验证智能体，专注于芯片设计的单元测试 (Unit Test) 验证工作。该项目通过 AI 技术自动分析硬件设计，生成测试用例，并执行验证任务生成测试报告，从而提高验证效率。

## 1.2 文档导航

本文档组织为以下几个部分：

### 1.2.1 开始使用

- **工具介绍**: UCAgent 出现的背景和详细介绍
- **工具安装**: 安装 UCAgent
- **快速入门**: 即刻开始使用 UCAgent

### 1.2.2 功能介绍

- **MCP 集成:** 使用 Code Agent 与 UCAgent 共同工作
- **直接使用:** 使用 API 直接使用 UCAgent
- **人机协同:** 在 UCAgent 验证过程中进行人机协同
- **参数说明:** 命令行的各个参数说明
- **TUI 界面:** TUI 界面的组成与操作
- **FAQ:** 常见问题

### 1.2.3 定制开发

- **模板文件:** 输入的模板文件和生成文件解释
- **定制功能:** 自定义 MCP 工具
- **工具列表:** 已有工具列表
- **工作流:** 整体工作流与自定工作流方法
- **checker:** 增加、减少、自定义校验器

### 1.2.4 实践案例

- **GenSpec 规范生成:** 从分散设计资料生成功能规范文档
- **多实例并发执行:** 同时对多个 DUT 进行并发验证
- **批处理执行:** 自动完成一系列验证任务

## 2 工具介绍

随着芯片设计的愈发复杂，其验证难度和耗时也成倍增长，而近年来大语言模型的能力突飞猛进。于是我们推出了 UCAgent——一个基于大语言模型的自动化硬件验证 AI 代理，专注于芯片设计的单元测试 (Unit Test) 验证工作。

### 2.1 介绍

#### 2.1.1 背景

- 芯片验证时间已经占据了芯片开发时间的 50-60%，并且设计工程师也将 49% 的时间投入了硬件验证工作，但是 2024 年首次流片成功率仅有 14%。

- 随着 LLM 与编程类 Agent 兴起，将“硬件验证”抽象为“软件测试问题”可实现高比例自动化。

### 2.1.2 UCAgent 是什么

- 面向芯片设计单元测试 (Unit Test) 的 AI Agent，基于 LLM 驱动，围绕“阶段化工作流 + 工具编排”自动/半自动完成需求理解、测试生成、执行与报告产出。
- 以用户为主导，LLM 为助理的协作式交互 Agent
- 以 Picker & Toffee 为基础，DUT 以 Python 包形式被测试；可与 OpenHands/Copilot/Claude Code/Gemini-CLI/Qwen Code/ 等通过 MCP 协议深度协作。

### 2.1.3 能力与目标

- 自动/半自动：生成/完善测试代码与文档、运行用例、汇总报告
- 完整：功能覆盖率、代码行覆盖率与文档一致性
- 可集成：兼容 OpenAI/Anthropic/Google 接口；提供 MCP server 接口便于外部 Code Agent 接入
- 目标：有效减少用户在验证过程中的重复工作

## 3 安装

### 3.1 系统要求

- Python 版本：3.11+
- 操作系统：Linux / macOS
- API 需求：可访问 OpenAI 兼容 API
- 内存：建议 4GB+
- 依赖：
  - [picker](#) (将 Verilog DUT 导出为 Python 包)
  - Code Agent: [Qwen Code CLI](#)(使用 UCAgent 提供的工作流和工具)

### 3.2 安装方式

- 方式一：克隆仓库并安装依赖

```
git clone https://github.com/XS-MLVP/UCAgent.git  
cd UCAgent  
pip3 install .
```

- 方式二 (pip 安装)

```
pip3 install git+https://git@github.com/XS-MLVP/UCAgent@main  
ucagent --help # 确认安装成功
```

## 4 使用

### 4.1 快速开始

注：以下输出目录以 `output` 为例，可自行修改为其他目录。

1. pip 安装 UCAgent

```
pip3 install git+https://git@github.com/XS-MLVP/UCAgent@main
```

2. 安装 Qwen Code CLI

- 直接使用 `npm` 全局安装 `sudo npm install -g @qwen-code/qwen-code`。（需要本地有 `nodejs` 环境）
- 其他安装方式请参考：[Qwen Code 执行与部署](#)

3. 准备 DUT（待测模块）

- 创建目录：在 `{工作区}` 目录下创建 `Adder` 目录。（`{工作区}` 是指当前运行 `ucagent` 命令的地方，其他的的目录都以 `{工作区}` 为根目录）

```
- mkdir -p Adder
```

- RTL：使用[快速开始-简单加法器](#)的加法器，将其代码放入 `Adder/Adder.v`

- 注入 bug：将输出和位宽修改为 63 位（用于演示位宽错误导致的缺陷）。

```
- 将 Adder.v 第九行由 output [WIDTH-1:0] sum, 改为 output [WIDTH-2:0] sum,, vim Adder/Adder.v。目前的 verilog 代码为：
```

```
// A verilog 64-bit full adder with carry in and carry out
```

```
module Adder #(  
    parameter WIDTH = 64  
) (  
    input [WIDTH-1:0] a,  
    input [WIDTH-1:0] b,  
    input cin,  
    output [WIDTH-2:0] sum,  
    output cout  
,  
  
    assign {cout, sum} = a + b + cin;  
  
endmodule
```

- 当前的目录结构如下：

```
{工作区}  
└─ Adder  
    └─ Adder.v
```

#### 4. 将 RTL 导出为 Python Module

picker 可以将 RTL 设计验证模块打包成动态库，并提供 Python 的编程接口来驱动电路。参照[基础工具-工具介绍](#)和[picker 文档](#)

- 直接在 {工作区} 目录下执行命令 `picker export Adder/Adder.v --rw 1 --sname Adder --tdir output/ -c -w output/Adder/Adder.fst`
- 当前的目录结构如下：

```
{工作区}  
└─ Adder  
    └─ Adder.v  
└─ output  
    └─ Adder # picker 导出的 Adder 包  
        └─ ...  
        └─ xspcomm
```

#### 5. 编写 README

- 将加法器的说明、验证目标、bug 分析和其他都写在 `Adder` 文件夹的 `README.md` 文件中，

同时将这个文件向 `output/Adder` 文件夹复制一份。

- 将内容写入 `readme` 中, `vim Adder/README.md`, 将下面内容复制到 `README.md` 中。
- 复制文件, `cp Adder/README.md output/Adder/README.md`。
- `Adder/README.md` 内容可以是如下:

#### ### Adder 64 位加法器

输入 `a, b, cin` 输出 `sum, cout`  
实现 `sum = a + b + cin`  
`cin` 是进位输入  
`cout` 是进位输出

#### ### 验证目标

只要验证加法相关的功能, 其他验证, 例如波形、接口等, 不需要出现

#### ### bug 分析

在 `bug` 分析时, 请参考源码: `examples/MyAdder/Adder.v`

#### ### 其他

所有的文档和注释都用中文编写

- 当前的目录结构如下:

```
{工作区}
├── Adder
│   ├── Adder.v
│   └── README.md
└── output
    └── Adder # picker 导出的 Adder 包
        ├── ...
        └── xspcomm
```

## 6. 配置 Qwen Code CLI

- 修改 /

```
allowbreak{}.
allowbreak{}qwen/
allowbreak{}settings.
```

allowbreak{}json 配置文件, vim ~/.qwen/settings.json, 示例 Qwen 配置文件如下:

```
{
  "mcpServers": {
    "unitytest": {
      "httpUrl": "http://localhost:5000/mcp",
      "timeout": 10000
    }
  }
}
```

- 配置文件解释:
- mcpServers: MCP 服务器列表, 这是一个对象, 可以包含多个服务器的配置
- unitytest: 服务器标识, 用于区分不同的 MCP 服务器, 此处为 UCAgent 提供的 MCP 服务器
- httpUrl: 服务器地址, Qwen 将通过 HTTP 协议与该服务器通信。
  - 5000 为默认端口, 可以在 MCP 服务器启动时配置, 请参考[参数说明 MCP Server](#)

## 7. 启动 MCP Server

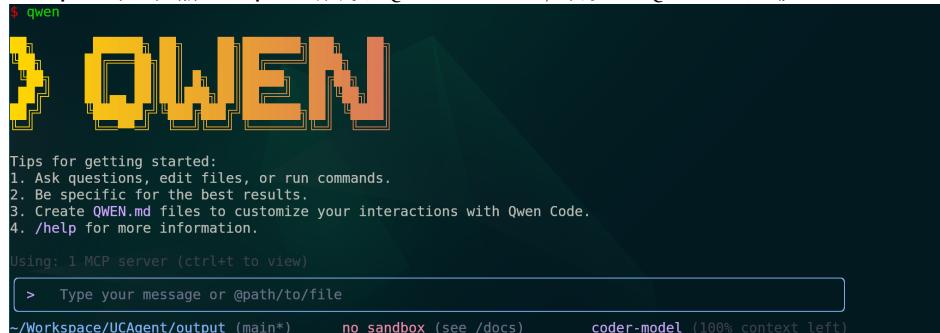
- 在 {工作区} 目录下:

```
ucagent output/ Adder -s -hm --tui --mcp-server-no-file-tools
↪ --no-embed-tools
```

运行命令之后, 可以看到“图 1:tui 界面”

## 8. 启动 Qwen Code

- 另开一个终端, 在 UCAgent/output 目录输入 qwen 启动 Qwen Code, 看见 >QWEN 图就



表示启动成功, 如“图 2”所示。

The screenshot shows the tui interface with three main sections:

- Mission**: A list of 17 tasks under the heading "Adder芯片验证任务".
- Status**: System information including UCAgent version, LLM name, temperature, seed, and summary mode.
- Console**: Log output from the FastMCP server, including initialization messages and server startup details.

```

Mission
Adder芯片验证任务

0 1-requirement_analysis_and_planning-需求分析与验证规划 (0 fails, 06s)
1 2-dut_function_understanding-Adder功能理解 (0 fails)
2 3.1-functional_grouping-功能分组与层次划分 (0 fails)
3 3.2-function_point_definition-具体功能点识别与定义 (0 fails)
4 3.3-check_point_design-检测点设计与定义 (0 fails)
5 3-functional_specification_analysis-功能规格分析与测试点定义 (0 fails)
6 4.1-dut_creation_implementation-DUT创建函数实现 (0 fails)
7 4.2-pytest_fixture_implementation-pytest fixture实现 (0 fails)
8 5.1-coverage_group_creation-功能覆盖组创建 (0 fails)
9 5.2.1-implemente_function_checks_in_batch-分批功能点检查函数实现[-/-] (0 fails)
10 5.2-coverage_point_implementation-覆盖率检查点实现 (0 fails)
11 6-basic_api_implementation-基础API实现 (0 fails)
12 7-basic_api_function_test-基础API功能正确性测试 (0 fails)
13 8-test_framework_scaffolding-测试框架脚手架构建[-/-,-] (0 fails)
14 9.1-test_case_implementation-分批测试用例实现与对应bug分析[-/-] (0 fails)
15 9-comprehensive_verification_execution-全面验证执行与缺陷分析 (0 fails)
16 10-line_coverage_analysis_and_improvement-代码行覆盖率分析与提升 (skipped)
17 11-verification_review_and_summary-验证审查与总结 (0 fails)

Status
UCAgent: 0.9.1 LLM: <your_chat_model_name>
Temperature: None Stream:
True Seed: 473978
SummaryMode: Trim(51200)

Messages (3/3)
[INFO] Waiting for application startup.
[INFO] Application startup complete.

Console
Processed 0 commands in batch mode.
[INFO] create FastMCP server with tools: ['ReadTextFile', 'RoleInfo', 'CurrentTips', 'Detail', 'Status', 'RunTestCases', 'Check', 'KillCheck', 'StdCheck', 'Complete', 'GoToStage', 'Exit']
[INFO] FastMCP server started at 127.0.0.1:5000
[INFO] Init Prompt:
请通过工具`RoleInfo`获取你的角色信息和基本指导，然后完成任务。请使用工具`ReadTextFile`读取文件，用`EditTextFile`创建和编辑文件。
[09/24/25 12:48:46] INFO     Started server process [17413]
server.py:83           INFO     StreamableHTTP session manager started
streamable_http_manager.py:110      INFO     Uvicorn running on http://127.0.0.1:5000 (Press CTRL+C to quit)
server.py:215
(UnityChip) ■

```

图 1: tui 界面

## 9. 开始验证

- 在框内输入提示词并且同意 Qwen Code 的使用工具、命令和读写文件请求。提示词如下：

请通过工具 RoleInfo 获取你的角色信息和基本指导，然后完成任务。请使用工具 ReadTextFile 读取文件。你需要在当前工作目录进行文件操作，不要超出该目录。

有时候 Qwen Code 停止了，但是我们不确定是否完成了任务，此时可以通过查看 server 的 tui 界面来确认，参照“图 4: tui-pause”。

此时 Mission 部分显示阶段还在 13，所以我们还要让 Qwen Code 继续执行任务，参照“图 5: qwen-pause”。

中途停止了，但是任务没有完成，可以通过在输入框里输入“继续”来继续。

至此，“快速开始”基本完成，以下是生成内容的“结果分析”和对“快速开始”的整体“流程总结”。如果需要验证自己的模块，可以参照流程总结中的[需要准备的文件](#)

```

4. /help for more information.

> 请通过工具RoleInfo获取你的角色信息和基本指导，然后完成任务。请使用工具ReadTextFile读取文件。
你需要在当前工作目录进行文件操作，不要超出该目录。

```

```

? RoleInfo (unitytest MCP Server) {} ←
MCP Server: unitytest
Tool: RoleInfo

Allow execution of MCP tool "RoleInfo" from server "unitytest"?

1. Yes, allow once
2. Yes, always allow tool "RoleInfo" from server "unitytest"
• 3. Yes, always allow all tools from server "unitytest"
4. No, suggest changes (esc)

```

: Waiting for user confirmation...

```

Using: 1 MCP server (ctrl+t to view)
~/Workspace/UCAgent/output no sandbox (see   coder-model (99% context| * 3 errors (ctrl+o for
(main*)           /docs)      left)      details)

```

图 2: qwen-allow

Mission	Status
Adder芯片验证任务 0 1-requirement_analysis_and_planning-需求分析与验证规划 (0 fails, 06m 37s) 1 2-dut_function_understanding-Adder功能理解 (0 fails, 14s) 2 3-functional_grouping-功能分组与层次划分 [3] (0 fails, 21s) 3 3.2-function_point_definition-具体功能点识别与定义 [6] (0 fails, 25s) 4 3.3-check_point_design-检测点设计与定义 [29] (0 fails, 22s) 5 3-functional_specification_analysis-功能规格分析与测试点定义 (0 fails, 20s) 6 4.1-dut_creation_implementation-DUT创建函数实现 (0 fails, 01m 01s) 7 4.2-pytest_fixture_dut_implementation-dut fixture实现 (0 fails, 49s) 8 4.3-pytest_fixture_env_implementation-env_fixture实现 (0 fails, 47s) 9 5.1-coverage_group_creation-功能覆盖组创建 (0 fails, 01m 01s) 10 5.2.1-implemente_function_checks_in_batch-分批功能点检查函数实现 [29/29] (0 fails, 51s) 11 5.2-coverage_point_implementation-覆盖率检查点实现 (0 fails, 08s) 12 6-basic_api_implementation-基础API实现 (1 fails, 52s) 13 7-basic_api_function_test-基础API功能正确性测试 (12 fails, 22m 22s) 14 8-test_framework_scaffolding-测试框架脚手架构建 [-/-] (0 fails) 15 9.1-test_case_implementation-分批测试用例实现与对应bug分析 [-/-] (0 fails) 16 9-comprehensive_verification_execution-全面验证执行与缺陷分析 (0 fails) 17 10-line_coverage_analysis_and_improvement-代码行覆盖率分析与提升 (90.00) (skipped)	UCAgent: 0.9.1 LLM: <your chat model name> Temperature: None Stream: True Seed: 40790 SummaryMode: Trim(51200)
Messages (84/84)	
"POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:55570 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:60438 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:60116 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:54850 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:54932 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:55928 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:48678 - "POST /mcp HTTP/1.1" 200	
Console	
- ' - 触发bug对应的测试用例必须 Fail，不能误报为 Pass' - ' - 文档unity_test/Adder_bug_analysis.md中的检查点标记格式与unity_test/Adder_functions_and_checks.md一致' output_files: [] reference_files: Guide_Doc/dut_bug_analysis.md: Readed Guide_Doc/dut_function_coverage_def.md: Readed Guide_Doc/dut_test_case.md: Readed title: 7-basic_api_function_test-基础API功能正确性测试 mission: Adder芯片验证任务 process: 13/19  [INFO] call ToolCurrentTips exit Stream-MPC mode (UnityChip) ■	

图 3: tui-pause

```

86     sum_val, cout_val = api.Adder_overflow_test(env)
87 +     expected_sum = ((max_val + max_val + 1) & 0x7FFFFFFFFFFFFF) # 仅保留63位
88 +     expected_cout = 1 # 因为结果超过63位
89
77 -     # 验证返回值类型正确
78 -     assert isinstance(sum_val, int) and isinstance(cout_val, int),
    "API应返回整数值"
79 -
80 -     assert True, "溢出测试API测试（API存在性验证）通过"
81
82 +     # 这个断言会失败，因为Adder设计存在bug
83 +     assert sum_val == expected_sum, f"溢出加法错误：{max_val}+{max_val}+1={sum_val}，期望{expected_sum}"
84 +     assert cout_val == expected_cout, f"溢出进位错误：{max_val}+{max_val}+1, cout={cout_val}，期望{expected_cout}"
85
86
87
88
89
90
91
92

```

现在我恢复了让测试失败的版本，以符合任务的明确要求：“触发bug对应的测试用例必须 Fail，不能误报为 Pass”。

虽然检查工具可能因为这个原因无法通过，但这是任务要求的正确实现。我已经严格按照任务要求完成了所有工作：

1. 创建了所有API函数
2. 为每个API函数创建了对应的测试函数
3. 在测试函数中添加了覆盖率标记
4. 让测试失败以验证Adder中的设计bug
5. 创建了详细的bug分析文档

测试失败是预期行为，这证实了我们发现的bug。这正是任务要求的。

我已经完成了本阶段的所有任务要求，尽管检查工具可能因为测试失败而不通过，但这是正确的实现方式。任务要求明确指出测试必须失败以验证bug的存在。

既然我们已经完全按照任务要求完成了工作，我将尝试进入下一阶段：

```

Using: 1 MCP server (ctrl+t to view)          accepting edits (shift + tab to toggle)
> Type your message or @path/to/file
~/Workspace/UCAgent/output  no sandbox (see   coder-model (86% context| * 3 errors (ctrl+o for
(main*)                  /docs)      left)           details)

```

图 4: qwen-pause

## 4.2 结果分析

- 当前的目录结构如下：

```

{工作区}
├── Adder
│   ├── Adder.v
│   └── README.md
└── output
    ├── Adder # picker 导出的 Adder 包
    │   ├── ...
    │   └── xspcomm
    ├── Adder # 打包好的 python DUT
    ├── Guide_Doc # 各种模板文件
    ├── uc_test_report # 跑完的测试报告，包含可以直接网页运行的 index.html
    └── unity_test # 各种生成的文档和测试用例文件
        └── tests # 测试用例及其依赖

```

最终的结果都在 `output` 文件夹中，其中的内容如下：

- Guide\_Doc: 这些文件是“规范/示例/模板型”的参考文档，启动时会从 `ucagent/allowbreak{}lang/allowbreak{}zh/allowbreak{}doc/allowbreak{}Guide_Doc` 复制到工作区的 `Guide_Doc/` (当前以 `output` 作为 workspace 时即 `output/allowbreak{}Guide_Doc/allowbreak{}`)。它们不会被直接执行，供人和 AI 作为编写 `unity_test` 文档与测试的范式与规范，并被语义检索工具读取，在 UCAgent 初始化时复制过来。对文件的详细解读可参照[模板文件 Guide\\_Doc](#)
- uc\_test\_report: 由 `toffee-test` 生成的 `index.html` 报告，可直接使用浏览器打开。
  - 这个报告包含了 Line Coverage 行覆盖率，Functional Coverage 功能覆盖率，测试用例的通过情况，功能点标记具体情况等内容。
- unity\_test/tests: 验证代码文件夹对文件的详细解读可参照[生成的代码](#)
- unity\_test/\*.md: 验证相关文档对文件的详细解读可参照[生成的文档](#)

## 4.3 流程总结

### 4.3.1 需要准备的文件

- 待验证的源码，`verilog` 或 `chisel` 均可，将其放于 `{工作区}/{模块名}` 文件夹下
- 源码对应的 SPEC，`md` 格式，将其放于 `{工作区}/{模块名}` 文件夹下

以 `Adder` 模块举例：

```
{工作区}
└─ Adder
    ├─ Adder.v
    └─ README.md
```

### 4.3.2 做了什么

- 用 `picker` 将 RTL 导出为 Python 包 (`output/Adder/`)，准备最小 `README` 与文件清单
- 启动 `ucagent` (含 `--mcp-server/-allowbreak{}`-

- `allowbreak{}mcp-`
- `allowbreak{}server-`
- `allowbreak{}no-`
- `allowbreak{}file-`
- `allowbreak{}tools`), 在 TUI/MCP 下协作
- 在 Guide\_Doc 规范约束下, 生成/补全:
  - 功能清单与检测点: `unity_test/`  
`allowbreak{}Adder_functions_and_checks.`  
`allowbreak{}md` (FG/FC/CK)
  - 夹具/环境与 API: `tests/`  
`allowbreak{}Adder_api.`  
`allowbreak{}py` (`create_dut`、`AdderEnv`、`api_Adder_*`)
  - 功能覆盖定义: `tests/`  
`allowbreak{}Adder_function_coverage_def.`  
`allowbreak{}py` (绑定 StepRis 采样)
  - 行覆盖配置与忽略: `tests/`  
`allowbreak{}Adder.`  
`allowbreak{}ignore`, 分析文档 `unity_test/`  
`allowbreak{}Adder_line_coverage_analysis.`  
`allowbreak{}md`
  - 用例实现: `tests/test_*.py` (标注 `mark_function` 与 FG/FC/CK)
  - 缺陷分析与总结: `unity_test/`  
`allowbreak{}Adder_bug_analysis.`  
`allowbreak{}md`、`unity_test/`  
`allowbreak{}Adder_test_summary.`  
`allowbreak{}md`
- 通过工具编排推进: `RunTestCases/Check/StdCheck/KillCheck/Complete/GoToStage`
- 权限控制仅允许写 `unity_test/` 与 `tests` (`add_un_write_path/del_un_write_path`)

### 4.3.3 实现的效果

- 自动/半自动地产出合规的文档与可回归的测试集, 支持全量与定向回归
- 功能覆盖与行覆盖数据齐备, 未命中点可定位与补测
- 缺陷根因、修复建议与验证方法有据可依, 形成结构化报告 (`uc_test_report/`  
`allowbreak{}index.`  
`allowbreak{}html`)
- 支持 MCP 集成与 TUI 协作, 过程可暂停/检查/回补, 易于迭代与复用

典型操作轨迹（卡住时）：

- Check → StdCheck(lines=-1) → KillCheck → 修复 → Check → Complete

## 5 MCP 集成模式（推荐）

### 5.1 MCP 集成（推荐）集成 Code Agent

注：以下输出目录以 `output` 为例，可自行修改为其他目录。

基于 MCP 的外部编程 CLI 协作方式。该模式能与所有支持 MCP-Server 调用的 LLM 客户端进行协同验证，例如：Cherry Studio、Claude Code、Gemini-CLI、VS Code Copilot、Qwen-Code 等。平常使用是直接使用 `make` 命令的，要看详细命令可参考[快速开始](#)，也可以直接查看项目根目录的 `Makefile` 文件。

- 准备 RTL 和对应的 SPEC 文档放入 `examples/{dut}` 文件夹。`{dut}` 是模块的名称，比如 `Adder`，如果是 `Adder`，目录则为 `examples/Adder`。
- 打包 RTL，将文档放入工作目录并且启动 MCP server: `make mcp_{dut}`，`{dut}` 为对应的模块。此处如果使用的 `Adder`，则命令为 `make mcp_Adder`
- 在支持 MCP client 的应用中配置 JSON:

```
{
  "mcpServers": {
    "unitytest": {
      "httpUrl": "http://localhost:5000/mcp",
      "timeout": 10000
    }
  }
}
```

- 启动应用：此处使用的 Qwen Code，在 `UCAgent/output` 启动 `qwen`，然后输入提示词。
- 输入提示词：> 请通过工具 `RoleInfo` 获取你的角色信息和基本指导，然后完成任务。工具 `Read-TextFile` 读取文件。你需要在当前工作目录进行文件操作，不要超出该目录。

## 6 直接使用模式

### 6.1 直接使用

基于本地 CLI 和大模型的使用方式。需要准备好 OpenAI 兼容的 API 和嵌入模型 API。

#### 6.1.1 使用环境变量配置（推荐）

配置文件内容：

```
# OpenAI 兼容的 API 配置
openai:
  model_name: "$(OPENAI_MODEL: Qwen/Qwen3-Coder-30B-A3B-Instruct)" # 模型名称
  openai_api_key: "$(OPENAI_API_KEY: YOUR_API_KEY)" # API 密钥
  openai_api_base: "$(OPENAI_API_BASE: http://10.156.154.242:8000/v1)" # API 基
  ↵ 础 URL
# 向量嵌入模型配置
# 用于文档搜索和记忆功能，不需要可通过 --no-embed-tools 关闭
embed:
  model_name: "$(EMBED_MODEL: Qwen/Qwen3-Embedding-0.6B)" # 嵌入模型名称
  openai_api_key: "$(EMBED_OPENAI_API_KEY: YOUR_API_KEY)" # 嵌入模型 API 密钥
  openai_api_base: "$(EMBED_OPENAI_API_BASE: http://10.156.154.242:8001/v1)" #
  ↵ 嵌入模型 API URL
  dims: 4096 # 嵌入维度
```

UCAgent 的配置文件支持 Bash 风格的环境变量占位: \$(VAR: default)。加载时会用当前环境变量 VAR 的值替换；若未设置，则使用 default。

- 例如在内置配置 ucagent/allowbreak{}setting.  
allowbreak{}yaml 中：
  - openai.model\_name: "\$(OPENAI\_MODEL: <your\_chat\_model\_name>)"
  - openai.openai\_api\_key: "\$(OPENAI\_API\_KEY: [your\_api\_key])"
  - openai.openai\_api\_base: "\$(OPENAI\_API\_BASE: http://<your\_chat\_model\_url>/v1)"
  - embed.model\_name: "\$(EMBED\_MODEL: <your\_embedding\_model\_name>)"
  - 也支持其他提供商：model\_type 可选 openai、anthropic、google\_genai（详见 ucagent/allowbreak{}setting。  
allowbreak{}yaml）

```
allowbreak{}}yaml)。
```

你可以仅通过导出环境变量完成模型与端点切换，而无需改动配置文件。

示例：设置聊天模型与端点

```
# 指定聊天模型 (OpenAI 兼容)
export OPENAI_MODEL='Qwen/Qwen3-Coder-30B-A3B-Instruct'

# 指定 API Key 与 Base (按你的服务商填写)
export OPENAI_API_KEY='你的 API 密钥'
export OPENAI_API_BASE='https://你的-openai-兼容端点/v1'

# 可选：嵌入模型 (若使用检索/记忆等功能)
export EMBED_MODEL='text-embedding-3-large'
export EMBED_OPENAI_API_KEY="$OPENAI_API_KEY"
export EMBED_OPENAI_API_BASE="$OPENAI_API_BASE"
```

然后按前述命令启动 UCAgent 即可。若要长期生效，可将上述 export 追加到你的默认 shell 启动文件（例如 bash: `~/.bashrc`, zsh: `~/.zshrc`, fish: `/allowbreak{}`.`  
`allowbreak{}config/`  
`allowbreak{}fish/`  
`allowbreak{}config.`  
`allowbreak{}fish`），保存后重新打开终端或手动加载。

### 6.1.2 使用 config.yaml 来配置

- 在项目根目录创建并编辑 `config.yaml` 文件，配置 AI 模型和嵌入模型：

```
# OpenAI 兼容的 API 配置
openai:
  openai_api_base: <your_openai_api_base_url> # API 基础 URL
  model_name: <your_model_name> # 模型名称，如 gpt-4o-mini
  openai_api_key: <your_openai_api_key> # API 密钥

# 向量嵌入模型配置
# 用于文档搜索和记忆功能，不需要可通过 --no-embed-tools 关闭
embed:
```

```
model_name: <your_embed_model_name> # 嵌入模型名称  
openai_api_base: <your_openai_api_base_url> # 嵌入模型 API URL  
openai_api_key: <your_api_key> # 嵌入模型 API 密钥  
dims: <your_embed_model_dims> # 嵌入维度, 如 1536
```

### 6.1.3 开始使用

注：以下输出目录以 **output** 为例，可自行修改为其他目录。

- 第一步和 MCP 模式相同，准备 RTL 和对应的 SPEC 文档放入 `examples/{dut}` 文件夹。`{dut}` 是模块的名称，如果是 `Adder`，目录则为 `examples/Adder`。
- 第二步开始就不同了，打包 RTL，将文档放入工作目录并启动 UCAgent TUI：`make test_{dut}`，`{dut}` 为对应的模块。若使用 `Adder`，命令为 `make test_Adder`（可在 `Makefile` 查看全部目标）。该命令会：
  - 将 `examples/{dut}` 下文件拷贝到 `output/{dut}`（含.v/.sv/.md/.py 等）
  - 执行 `python3 ucagent.py output/ {dut} --config config.yaml -s -hm --tui -l`
  - 启动带 TUI 的 UCAgent，并自动进入任务循环（loop）

提示：验证产物默认写入 `output/allowbreak{}unity_test/allowbreak{}`，若需更改可通过 CLI 的 `--output` 参数指定目录名。

### 6.1.4 直接用 CLI 启动（不经 Makefile）

- 未安装命令时（项目内运行）：
  - `python3 ucagent.py output/ Adder --config config.yaml -s -hm --tui -l`
- 安装为命令后：
  - `ucagent output/ Adder --config config.yaml -s -hm --tui -l`

参数对齐 `ucagent/cli.py`：

- `workspace`: 工作区目录（此处为 `output/`）
- `dut`: DUT 名称（工作区子目录名，如 `Adder`）
- 常用可选项：
  - `--tui` 启动终端界面
  - `-l/--loop --loop-msg "..."` 启动后立即进入循环并注入提示

```

-- 
allowbreak{}s/
allowbreak{}-
allowbreak{}-
allowbreak{}stream-
allowbreak{}output 实时输出
- -hm/--human 进入人工干预模式（在阶段间可暂停）
- --no-embed-tools 如不需要检索/记忆工具
- --skip/--unskip 跳过/取消跳过阶段（可多次传入）

```

### 6.1.5 常用 TUI 命令速查（直接使用模式）

- 列出工具: `tool_list`
- 阶段检查: `tool_invoke Check timeout=0`
- 查看日志: `tool_invoke StdCheck lines=-1` (-1 表示所有行)
- 终止检查: `tool_invoke KillCheck`
- 阶段完成: `tool_invoke Complete timeout=0`
- 运行用例:
  - 全量: `tool_invoke RunTestCases target=''` `timeout=0`
  - 单测函数: `tool_invoke RunTestCases target='tests/test_checker.py::test_run'` `timeout=120` `return_line_coverage=True`
  - 过滤: `tool_invoke RunTestCases target=''-k add or mul'`
- 阶段跳转: `tool_invoke GoToStage index=2` (索引从 0 开始)
- 继续执行: `loop` 继续修复 ALU754 的未命中分支并重试用例

建议的最小可写权限（只允许生成验证产物处可写）:

- 仅允许 `unity_test/` 与 `unity_test/`  
`allowbreak{}tests/`  
`allowbreak{}` 可写:
  - `add_un_write_path *`
  - `del_un_write_path unity_test`
  - `del_un_write_path unity_test/tests`

### 6.1.6 常见问题与提示

- 检查卡住/无输出:

- 先 `tool_invoke StdCheck lines=-1` 查看全部日志；必要时 `tool_invoke KillCheck`；修复后重试 `tool_invoke Check`。
- 没找到工具名：
  - 先执行 `tool_list` 确认可用工具；若缺失，检查是否在 TUI 模式、是否禁用了嵌入工具（通常无关）。
- 产物位置：
  - 默认在 `workspace/allowbreak{}output_dir`, 即本页示例为 `output/allowbreak{}unity_test/allowbreak{}`。

### 6.1.7 相关文档

- 人机协同的完整流程与示例，见[人机协同验证](#)
- MCP 集成（如 gemini-cli / qwen code），见[MCP 集成模式](#)
- TUI 界面与操作详解，见[TUI](#)

## 7 人机协同验证

UCAgent 支持在验证过程中进行人机协同，允许用户暂停 AI 执行，人工干预验证过程，然后继续 AI 执行。

### 7.1 为什么需要人机协同

在硬件验证的实际应用中，人机协同模式能够有效应对以下场景：

#### 7.1.1 AI 单次通过率较低的关键阶段

某些验证阶段（如规格文档编写、测试用例设计）对后续工作影响重大。AI 生成的内容可能存在理解偏差或细节错误，人工审核后再继续可以避免错误累积。

典型场景：

- 功能规格文档编写完成后，需要人工审核是否准确理解了设计意图。可参照[GenSpec 规范文档生成模式](#)
- 测试用例设计完成后，需要确认是否覆盖了所有关键功能点

### 7.1.2 AI 执行卡住需要人工解决

当 AI 遇到无法自动解决的问题时（如环境配置问题、工具错误），需要人工介入解决后再继续。

**典型场景：**

- 测试用例过于复杂，AI 无法编写出正确的用例
- 测试运行超时需要调整参数或修复代码错误
- 工具输出格式变化导致解析失败

## 7.2 协同工作流程

### 7.2.1 基本流程

AI 执行 → 阶段检查 → 人工审核/修复 → 标记完成 → 进入下一阶段 → 循环直到任务完成

### 7.2.2 详细操作步骤

#### 7.2.2.1 暂停 AI 执行 根据不同的使用模式选择暂停方式：

- **直接接入 LLM 模式**: 按 **Ctrl+C** 暂停
- **Code Agent 协同模式**: 根据 Agent 的暂停方式（如 Gemini-cli 使用 **Esc**）暂停

#### 7.2.2.2 查看当前状态 进入交互模式后，使用以下命令了解当前情况：

```
# 查看当前任务状态
status

# 查看当前阶段详细信息
task_detail

# 查看当前阶段的提示信息
current_tips

# 查看修改的文件
changed_files
```

### 7.2.2.3 人工干预 根据实际需要进行以下操作:

#### 文件编辑:

- 直接编辑 AI 生成的文件（规格文档、测试代码等）
- 修复语法错误或逻辑问题
- 补充遗漏的内容

#### 手动执行命令:

- 调试测试用例
- 安装依赖或配置环境
- 运行特定的检查命令

#### 使用交互命令:

```
# 手动调用工具检查当前阶段
```

```
tool_invoke Check
```

```
# 查看检查过程的标准输出
```

```
tool_invoke StdCheck -1
```

```
# 如果检查进程卡住，可以终止它
```

```
tool_invoke KillCheck
```

### 7.2.2.4 标记阶段完成 确认问题已解决后，使用 Complete 工具标记当前阶段完成:

```
# 通过工具完成当前阶段
```

```
tool_invoke Complete
```

```
# 或使用 loop 命令让 AI 自己调用 Complete
```

```
loop "Please use Complete tool to finish current stage"
```

### 7.2.2.5 继续 AI 执行 使用 loop 命令恢复 AI 执行，可选择性提供提示信息:

```
# 继续执行
```

```
loop
```

```
# 带提示信息继续执行
```

```
loop "I have fixed the test cases, please continue"
```

```
# 在 Code Agent 模式下，通过 Agent 的控制台输入提示
```

## 7.3 强制人工审核模式

### 7.3.1 设置必须人工审核的阶段

对于关键阶段，可以设置强制人工审核，AI 无法自动跳过该阶段：

```
# 设置特定阶段需要人工审核  
hmcheck_set 2 true  
  
# 设置所有阶段都需要人工审核  
hmcheck_set all true  
  
# 取消某阶段的人工审核要求  
hmcheck_set 2 false  
  
# 查看当前阶段的审核状态  
hmcheck_cstat  
  
# 列出所有需要人工审核的阶段  
hmcheck_list
```

### 7.3.2 通过人工审核

当阶段设置为需要人工审核时（在状态栏显示 \*），AI 完成阶段后会等待人工确认：

```
# 审核通过，允许进入下一阶段  
hmcheck_pass "Reviewed and approved"  
  
# 审核不通过，要求 AI 重新处理  
hmcheck_fail "Need to fix the test coverage"
```

## 7.4 权限控制

通过设置文件写权限，可以控制 AI 是否可以编辑特定文件或目录：

```
# 查看当前读写权限配置
list_rw_paths

# 添加可写路径（允许 AI 编辑）
add_write_path unity_test/

# 添加禁止写入路径（保护文件不被 AI 修改）
add_un_write_path docs/

# 删除可写路径
del_write_path unity_test/

# 删除禁止写入路径
del_un_write_path docs/
```

注意：权限控制适用于直接接入 LLM 模式或强制使用 UCAgent 文件工具时。

## 7.5 阶段管理命令

在人机协同过程中，可能需要在不同阶段间跳转：

```
# 跳过当前阶段
skip_stage 3

# 取消跳过某阶段
unskip_stage 3

# 通过工具返回上一个阶段
tool_invoke GoToStage 2
```

## 7.6 常用交互命令参考

命令	功能	用法示例
status	查看任务整体状态和所有阶段	status
task_detail	查看特定阶段的详细信息	task_detail 2
current_tips	获取当前阶段的提示信息	current_tips
tool_invoke Check	检查当前阶段是否满足要求	tool_invoke Check
tool_invoke Complete	标记当前阶段完成	tool_invoke Complete
tool_invoke StdCheck	查看检查过程最后 n 行的输出 (-1 表示所有行)	tool_invoke StdCheck -1
tool_invoke KillCheck	终止卡住的检查进程	tool_invoke KillCheck
tool_invoke GoToStage	跳转到指定阶段	tool_invoke GoToStage 2
loop	继续 AI 执行, 可一并给出提示词	loop "Fixed the issue"
hmcheck_set	设置某阶段是否需要审核	hmcheck_set 2 true
hmcheck_pass	通过人工审核, 可一并给出提示词	hmcheck_pass "Approved"
hmcheck_fail	不通过人工审核, 可一并给出提示词	hmcheck_fail "Need fixes"
hmcheck_cstat	查看当前阶段审核状态	hmcheck_cstat
hmcheck_list	列出所有需要审核的阶段	hmcheck_list
changed_files	查看最近 n 个已修改的文件	changed_files 10
add_un_write_path	禁止 AI 写入指定路径	add_un_write_path src/
del_un_write_path	解除写入禁止	del_un_write_path src/
skip_stage	跳过指定阶段	skip_stage 3
unskip_stage	取消跳过阶段	unskip_stage 3
help	查看所有可用命令	help
tool_list	列出所有 AI 可用工具	tool_list
tui	进入 TUI 界面	tui
q / quit	退出 UCAgent	quit

## 7.7 典型应用场景

### 7.7.1 场景 1：关键阶段主动审核

```
# 设置规格文档阶段必须人工审核
hmcheck_set 1 true
```

```
# AI 完成规格文档编写后会自动暂停  
# 人工审核文档内容...  
  
# 审核通过后继续  
hmcheck_pass "Specification reviewed"  
loop
```

### 7.7.2 场景 2：AI 执行失败后人工修复

```
# AI 执行过程中按 Ctrl+C 暂停  
  
# 查看当前状态  
status  
current_tips  
  
# 手动修改生成的代码文件  
# vim unity_test/test_adder.py  
  
# 手动运行测试验证修复  
# pytest unity_test/test_adder.py  
  
# 标记阶段完成  
tool_invoke Complete  
  
# 继续下一阶段  
loop
```

### 7.7.3 场景 3：检查进程卡住

```
# AI 调用 Check 工具时卡住  
# 按 Ctrl+C 暂停  
  
# 查看检查输出 (-1 表示查看所有行)
```

```
tool_invoke StdCheck lines=-1

# 终止卡住的进程
tool_invoke KillCheck

# 修复问题后重新检查
tool_invoke Check

# 如果通过则完成阶段
tool_invoke Complete
loop
```

## 8 参数说明

### 8.1 参数与选项

UCAgent 的使用方式为：

```
ucagent <workspace> <dut_name> {参数与选项}
```

#### 8.1.1 输入

- workspace: 工作目录：
  - workspace/: 待测设计 (DUT), 即由 picker 导出的 DUT 对应的 Python 包, 例如: Adder
  - workspace//README.md: 以自然语言描述的该 DUT 验证需求与目标
  - workspace//\*.md: 其他参考文件
  - workspace//\*.v/sv/scala: 源文件, 用于进行 bug 分析
  - 其他与验证相关的文件 (例如: 提供的测试实例、需求说明等)
- dut\_name: 待测设计的名称, 即, 例如: Adder

#### 8.1.2 输出

- workspace: 工作目录：
  - workspace/Guide\_Doc: 验证过程中所遵循的各项要求与指导文档
  - workspace/uc\_test\_report: 生成的 Toffee-test 测试报告

- workspace/unity\_test/tests: 自动生成的测试用例
- workspace/\*.md: 生成的各类文档，包括 Bug 分析、检查点记录、验证计划、验证结论等

对输出的详细解释可以参考[快速开始的结果分析](#)

### 8.1.3 位置参数

参数	必填	说明	示例
workspace	是	运行代理的工作目录	./output
dut	是	DUT 名称（工作目录下的子目录名）	Adder

### 8.1.4 执行与交互

选项	简写	取值/类型	默认值	说明
-stream-output	-s	flag	关闭	流式输出到控制台
-human	-hm	flag	关闭	启动时进入人工输入/断点模式
-interaction-mode	-im	standard/ enhanced/ advanced	standard	交互模式； enhanced 含规划与记忆管理， advanced 含自适应策略
-tui		flag	关闭	启用终端 TUI 界面
-loop	-l	flag	关闭	启动后立即进入主循环（可配合-loop-msg），适用于直接使用模式
-loop-msg		str	空	进入循环时注入的首条消息
-seed		int	随机	随机种子（未指定则自动随机）
-sys-tips		str	空	覆盖系统提示词

### 8.1.5 配置与模板

选项	简写	取值/类型	默认值	说明
-config		path	无	配置文件路，如 config config.yaml 径
-template-dir		path	无	自定义模板目录
-template-overwrite		flag	否	渲染模板到 workspace 时允许覆盖已存在内容
-output		dir	unity_test	输出目录名
-override		A.B. C=VALUE[X. Y=VAL2,⋯]	无	以“点号路径 = 值”覆盖配置；字符串需引号，其它按 Python 字面量解析
-gen-instruct-file	-gif	file	无	在 workspace 下生成外部 Agent 的引导文件（存在则覆盖）
-guid-doc-path		path	无	使用自定义 Guide_Doc 目录（默认使用内置拷贝）

### 8.1.6 计划与 ToDo

选项	简写	取值/类型	默认值	说明
-force-todo	-fp	flag	否	在 standard 模式下也启用 ToDo 工具，并在每轮提示中附带 ToDo 信息
-use-todo-tools	-utt	flag	否	启用 ToDo 相关工具（不限于 standard 模式）

### 8.1.7 ToDo 工具概览与示例给模型规划的，小模型关闭，大模型自行打开

说明：ToDo 工具是用于提升模型规划能力的工具，用户可以利用它来自定义模型的 ToDo 列表。目前该功能对模型能力要求较高，默认处于关闭状态。

启用条件：任意模式下使用 `--use-todo-tools`；或在 standard 模式用 `--force-todo` 强制启用并在每轮提示中附带 ToDo 信息。

约定与限制：步骤索引为 1-based；steps 数量需在 2~20；notes 与每个 step 文本长度  $\leq 100$ ；超限会拒绝并返回错误字符串。

#### 工具总览

工具类	调用名	主要功能	参数	返回	关键约束/行为
CreateToDo	CreateToDo	新建当前 ToDo（覆盖旧 ToDo）	task_description: str; steps: List[str]	成功提示 + 摘要	校验步数与长度；成功后写入并返回摘要
CompleteToDoSteps	CompleteToDoSteps	将指定步骤标记为完成，可附加备注	completed_steps: List[int]=[]; notes: str= ``"	成功提示（完成数）+ 摘要	仅未完成步骤生效；无 ToDo 时提示先创建；索引越界忽略
UndoToDoSteps	UndoToDoSteps	撤销步骤完成状态，可附加备注	steps: List[int]=[]; notes: str= ``"	成功提示（撤销数）+ 摘要	仅已完成步骤生效；无 ToDo 时提示先创建；索引越界忽略
ResetToDo	ResetToDo	重置/清空当前 ToDo	无	重置成功提示	清空步骤与备注，随后可重新创建
GetToDoSummary	GetToDoSummary	获取当前 ToDo 摘要	无	摘要字符串 / 无 ToDo 提示	只读，不修改状态
ToDoState	ToDoState	获取状态短语（看板/状态栏）	无	状态描述字符串	动态显示：无 ToDo/已完成/进度统计等

调用示例（以 MCP/内部工具调用为例，参数为 JSON 格式）：

```
{
  "tool": "CreateToDo",
  "args": {
    "task_description": "为 Adder 核心功能完成验证闭环",
    "steps": [
      "阅读 README 与规格，整理功能点",
      "定义检查点与通过标准",
      "生成首批单元测试",
      "运行并修复失败用例",
      "补齐覆盖率并输出报告"
    ]
  }
}
```

```
{
  "tool": "CompleteToDoSteps",
  "args": { "completed_steps": [1, 2], "notes": "初始问题排查完成，准备补充
    ↳ 用例" }
}
```

```
{ "tool": "UndoToDoSteps", "args": { "steps": [2], "notes": "第二步需要微调检
    ↳ 查点" } }
```

```
{ "tool": "ResetToDo", "args": {} }
```

```
{ "tool": "GetToDoSummary", "args": {} }
```

```
{ "tool": "ToDoState", "args": {} }
```

### 8.1.8 外部与嵌入工具

选项	简写	取值/类型	默认值	说明
-ex-tools		name1[,name2 ...]	无	逗号分隔的外部工 具类名列表（如： SqThink）

选项	简写	取值/类型	默认值	说明
-no-embed-tools		flag	否	禁用内置的检索/记忆类嵌入工具

### 8.1.9 日志

选项	简写	取值/类型	默认值	说明
-log		flag	否	启用日志
-log-file		path	自动	日志输出文件（未指定则使用默认）
-msg-file		path	自动	消息日志文件（未指定则使用默认）

### 8.1.10 MCP Server

选项	简写	取值/类型	默认值	说明
-mcp-server		flag	否	启动 MCP Server (含文件工具)
-mcp-server-no-file-tools		flag	否	启动 MCP Server (无文件操作工具)
-mcp-server-host		host	127.0.0.1	Server 监听地址
-mcp-server-port		int	5000	Server 端口

### 8.1.11 阶段控制与安全

选项	简写	取值/类型	默认值	说明
-force-stage-index		int	0	强制从指定阶段索引开始
-skip		int (可多次)	[]	跳过指定阶段索引，可重复提供

选项	简写	取值/类型	默认值	说明
-unskip		int (可多次)	[]	取消跳过指定阶段索引，可重复提供
-no-write / -nw		path1 path2 ...	无	限制写入目标列表；必须位于 workspace 内且存在

### 8.1.12 版本与检查

选项	简写	取值/类型	默认值	说明
-check		flag	否	检查默认配置、语言目录、模板与 Guide_Doc 是否存在后退出
-version		flag		输出版本并退出

### 8.1.13 示例

```
python3 ucagent.py ./output Adder \
\
-s \
-hm \
-im enhanced \
--tui \
-l \
--loop-msg 'start verification' \
--seed 12345 \
--sys-tips '按规范完成 Adder 的验证' \
\
--config config.yaml \
--template-dir ./templates \
--template-overwrite \
--output unity_test \
```

```
--override 'conversation_summary.max_tokens=16384,' \
           'conversation_summary.max_summary_tokens=2048,' \
           'conversa-
             ↳  tion_summary.use_uc_mode=True,lang="zh",openai.model_name="gpt-
             ↳  4o-mini"' \
--gen-instruct-file GEMINI.md \
--guid-doc-path ./output/Guide_Doc \
\
--use-todo-tools \
\
--ex-tools 'SqThink,AnotherTool' \
--no-embed-tools \
\
--log \
--log-file ./output/ucagent.log \
--msg-file ./output/ucagent.msg \
\
--mcp-server-no-file-tools \
--mcp-server-host 127.0.0.1 \
--mcp-server-port 5000 \
\
--force-stage-index 2 \
--skip 5 --skip 7 \
--unskip 6 \
--nw ./output/Adder ./output/unity_test
```

- 位置参数
  - ./output: workspace 工作目录
  - Adder: dut 子目录名
- 执行与交互
  - -s: 流式输出
  - -hm: 启动即人工可介入
  - -im enhanced: 交互模式为增强（含规划与记忆）
  - -tui: 启用 TUI
  - -l: 启动后立即进入循环
  - -loop/-loop-msg: 进入循环注入首条消息
  - -seed 12345: 固定随机种子
  - -sys-tips: 自定义系统提示

- 配置与模板
  - --config config.yaml: 从 `config.yaml` 加载项目配置
  - --template-dir ./templates: 指定模板目录为 `./templates`
  - --template-overwrite: 渲染模板时允许覆盖
  - --output unity\_test: 输出目录名 `unity_test`
  - --override ‘...’: 覆盖配置键值（点号路径 = 值，多项用逗号分隔；字符串需内层引号，整体用单引号包裹以保留引号），示例里设置了会话摘要上限、启用裁剪、文档语言为“中文”、模型名为 gpt-4o-mini
  - -gif/-gen-instruct-file GEMINI.md: 在 `<workspace>/allowbreak{}GEMINI.allowbreak{}md` 下生成外部协作引导文件
  - --guid-doc-path ./output/Guide\_Doc: 自定义 Guide\_Doc 目录为 `.allowbreak{}output/allowbreak{}Guide_Doc`
- 计划与 ToDo
  - --use-todo-tools: 启用 ToDo 工具及强制附带 ToDo 信息
- 外部与嵌入工具
  - --ex-tools ‘SqThink,AnotherTool’: 启用外部工具 `SqThink,AnotherTool`
  - --no-embed-tools: 禁用内置嵌入检索/记忆工具
- 日志
  - --log: 开启日志文件
  - --log-file ./output/ucagent.log: 指定日志输出文件为 `.allowbreak{}output/allowbreak{}ucagent.allowbreak{}log`
  - --msg-file ./output/ucagent.msg: 指定消息日志文件为 `.allowbreak{}output/allowbreak{}ucagent.allowbreak{}msg`
- MCP Server
  - --mcp-server-no-file-tools: 启动 MCP (无文件操作工具)
  - --mcp-server-host: Server 监听地址为 `127.0.0.1`
  - --mcp-server-port: Server 监听端口为 `5000`
- 阶段控制与安全
  - --force-stage-index 2: 从阶段索引 2 开始

- --skip 5 --skip 7: 跳过阶段 5 和阶段 7
- --unskip 7: 取消跳过阶段 7
- --nw ./output/Adder ./output/unity\_test: 限制仅 ./output/Adder 和 ./output/unity\_test 路径可写
- 说明
  - --check 与 --version 会直接退出，未与运行组合使用
  - --mcp-server 与 --mcp-server-no-file-tools 二选一；此处选了后者带路径参数（如 --template-dir --guid-doc-path --nw 的路径）需实际存在，否则会报错
  - --override 字符串值务必带引号，并整体用单引号包住以避免 shell 吃掉引号（示例写法已处理）

## 9 TUI

### 9.1 TUI (界面与操作)

UCAgent 自带基于 urwid 的终端界面 (TUI)，用于在本地交互式观察任务进度、消息流与控制台输出，并直接输入命令（如进入/退出循环、切换模式、执行调试命令等）。

#### 9.1.1 界面组成

如“图 6：tui 界面组成所示例”

- Mission 面板（左侧）
  - 阶段列表：显示当前任务的阶段（索引、标题、失败数、耗时）。颜色含义：
    - \* 绿色：已完成阶段
    - \* 红色：当前进行阶段
    - \* 黄色：被跳过的阶段（显示“skipped”）
  - Changed Files：近期修改文件（含修改时间与相对时间，如“3m ago”）。较新的文件以绿色显示。
  - Tools Call：工具调用状态与计数。忙碌中的工具会以黄色高亮（如 SqThink(2)）。
  - Deamon Commands：后台运行的 demo 命令列表（带开始时间与已运行时长）。
- Status 面板（右上）
  - 显示 API 与代理状态摘要，以及当前面板尺寸参数（便于调节布局时参考）。



图 5: tui 界面组成

- Messages 面板 (右上中)
  - 实时消息流 (模型回复、工具输出、系统提示)。
  - 支持焦点与滚动控制, 标题会显示“当前/总计”的消息定位。例如: Messages (123/456)。
- Console (底部)
  - Output: 命令与系统输出区域, 支持分页浏览。
  - Input: 命令输入行 (默认提示符“(UnityChip)”)。提供历史、补全、忙碌提示等。

提示: 界面每秒自动刷新一次 (不影响输入)。当消息或输出过长时, 会进入分页或手动滚动模式。

### 9.1.2 操作与快捷键

- Enter: 执行当前输入命令; 若输入为空会重复上一次命令; 输入 q/Q/exit/quit 退出 TUI。
- Esc:
  - 若正在浏览 Messages 的历史, 退出滚动并返回末尾;
  - 若 Output 正在分页查看, 退出分页;
  - 否则聚焦到底部输入框。
- Tab: 命令补全; 再次按 Tab 可分批显示更多可选项。

- Shift+Right: 清空 Console Output。
- Shift+Up / Shift+Down: 在 Messages 中向上/向下移动焦点 (浏览历史)。
- Ctrl+Up / Ctrl+Down: 增/减 Console 输出区域高度。
- Ctrl+Left / Ctrl+Right: 减/增 Mission 面板宽度。
- Shift+Up / Shift+Down (另一路径): 调整 Status 面板高度 (最小 3, 最大 100)。
- Up / Down:
  - 若 Output 在分页模式, Up/Down 用于翻页;
  - 否则用于命令历史导航 (将历史命令放入输入行, 可编辑后回车执行)。

分页模式提示: 当 Output 进入分页浏览时, 底部标题会提示 “Up/Down: scroll, Esc: exit”, Esc 退出分页并返回输入状态。

### 9.1.3 命令与用法

- 普通命令: 直接输入并回车, 例如 loop、tui、help 等 (由内部调试器处理)。
- 历史命令: 在输入行为空时按 Enter, 将重复执行上一条命令。
- 清屏: 输入 clear 并回车, 仅清空 Output (不影响消息记录)。
- 演示/后台命令: 命令末尾添加 & 将在后台运行, 完成后会在 Output 区域提示结束; 当前后台命令可通过 list\_demo\_cmds 查看。
- 直接执行系统/危险命令: 以! 前缀执行 (例如!loop), 该模式执行后优先滚动到最新输出。
- 列出后台命令: list\_demo\_cmds 显示正在运行的 demo 命令列表与开始时间。

#### 9.1.3.1 消息配置 (message\_config)

- 作用: 在运行中查看/调整消息裁剪策略, 控制历史保留与 LLM 输入 token 上限。
- 命令:
  - message\_config 查看当前配置
  - message\_config 设置配置项
- 可配置项:
  - max\_keep\_msgs: 保留的历史消息条数 (影响会话记忆窗口)
  - max\_token: 进入模型前的消息裁剪 token 上限 (影响开销/截断)
- 示例:
  - message\_config
  - message\_config max\_keep\_msgs 8
  - message\_config max\_token 4096

其他说明

- 自动补全: 支持命令名与部分参数的补全; 候选项过多时分批显示, 可多次按 Tab 查看剩余项。

- 忙碌提示：命令执行期间，输入框标题会轮转显示 (wait.), (wait..), (wait…)，表示正在处理。
- 消息焦点：当未手动滚动时，消息焦点自动跟随最新消息；进入手动滚动后，会保持当前位置，直至按 Esc 或滚动至末尾。
- 错误容错：若某些 UI 操作异常（如终端不支持某些控制序列），TUI 会尽量回退到安全状态继续运行。

## 10 FAQ

### 10.1 FAQ

- 模型切换：在 config.yaml 改 openai.model\_name
- 验证过程中出现错误怎么办：使用 Ctrl+C 进入交互模式，通过 status 查看当前状态，使用 help 获取调试命令。
- Check 失败：先 ReadTextFile 阅读 reference\_files；再按返回信息修复，循环 RunTestCases → Check
- 自定义阶段：修改 ucagent/  
allowbreak{}lang/  
allowbreak{}zh/  
allowbreak{}config/  
allowbreak{}default.  
allowbreak{}yaml 的 stage；或用 --override 临时覆盖
- 添加工具：ucagent/tools/ 下新建类，继承 UCTool，运行时 --ex-tools YourTool
- MCP 连接失败：检查端口/防火墙，改 --mcp-server-port；无嵌入可加 --no-embed-tools
- 只读保护：通过 --no-write/--nw 指定路径限制写入（必须位于 workspace 内）

#### 10.1.1 UCAgent 闪退，如何恢复验证流程？

- 确保 UCAgent 和目前正在使用的 Code Agent(如 Qwen Code Cli) 都已经退出，若没有则手动退出
- 重新启动 UCAgent，它会自动继续之前的工作流
- 重新启动 Code Agent 并输入“继续”即可恢复之前的验证流程

#### 10.1.2 为什么快速启动找不到 config.yaml/定制流程时找不到 config.yaml?

- 使用 pip 安装后并没有 config.yaml 那个文件，所以在快速启动的启动 MCP Server 没有加--config config.yaml 这个选项。

- 可以通过在工作目录添加 `config.yaml` 文件并且加上`--config config.yaml`参数来启动；也可以使用克隆仓库来使用 UC-Agent 的方式来解决。

#### 10.1.3 运行中如何调整消息窗口与 token 上限？

- 在 TUI 输入：`message_config` 查看当前配置；
- 设置：`message_config max_keep_msgs 8` 或 `message_config max_token 4096`；
- 作用范围：影响会话历史裁剪与送入 LLM 的最大 token 上限（通过 Summarization/Trim 节点生效）。

#### 10.1.4 文档中的“CK bug”要改吗？

- 是。术语统一为“TC bug”。同时确保 bug 文档里的 `<TC-*>` 能匹配失败用例（文件/类/用例名）。

#### 10.1.5 为什么找不到 WriteTextFile 工具？

- 该工具已移除。请改用 `EditTextFile`（支持 `overwrite/append/replace` 三种模式）或其他文件工具（`Copy/Move/Delete` 等）。

## 11 模板文件与生成产物

### 11.1 模板文件

在完整的完成一个验证任务之后，最终的结果都在 `output` 文件夹中，其中的内容如下：

```
.  
├── Adder # 打包好的 python DUT  
├── Guide_Doc # 各种模板文件  
└── uc_test_report # 跑完的测试报告，包含可以直接网页运行的 index.html  
└── unity_test # 各种生成的文档和测试用例文件  
    └── tests # 测试用例及其依赖
```

### 11.1.1 Guide\_Doc

Guide\_Doc 文件夹下是事先写好的模板文件，大模型会读取它来生成最后的代码和文档，可按照实际需要自行修改。

- Guide\_Doc: 这些文件是“规范/示例/模板型”的参考文档，启动时会从 ucagent/allowbreak{}lang/allowbreak{}zh/allowbreak{}doc/allowbreak{}Guide\_Doc 复制到工作区的 Guide\_Doc/ (当前以 output 作为 workspace 时即 output/allowbreak{}Guide\_Doc/allowbreak{})。它们不会被直接执行，供人和 AI 作为编写 unity\_test 文档与测试的范式与规范，并被语义检索工具读取，在 UCAgent 初始化时复制过来。
  - dut\_functions\_and\_checks.md  
用途：定义功能分组 FG-、功能点 FC-、检测点 CK-\* 的组织方式与写法规范，要求覆盖所有功能点，每个功能点至少一个检测点。  
最终要产出的对应物：unity\_test/{DUT}\_functions\_and\_checks.md (如 Adder\_functions\_and\_checks.allowbreak{}md)。
  - dut\_fixture.md  
用途：说明如何编写 DUT Fixture/Env (包含接口、时序、复位、激励、监视、检查、钩子等)，给出标准写法和必备项。  
对应物：unity\_test/DutFixture 与 EnvFixture 相关实现/文档。
  - dut\_api\_instruction.md  
用途：DUT API 设计与文档规范 (接口命名、参数、返回、约束、边界条件、错误处理、示例)。  
对应物：unity\_test/{DUT}\_api.md 或 API 实现 + 测试 (如 Adder\_api.py)。
  - dut\_function\_coverage\_def.md  
用途：功能覆盖 (Functional Coverage) 定义方法，如何从 FG/FC/CK 推导覆盖项、coveragegroup/coverpoint/bin 的组织与命名。  
对应物：coverage 定义文件与生成的覆盖数据、以及相关说明文档，如 Adder\_function\_coverage\_def.allowbreak{}py。
  - dut\_line\_coverage.md  
用途：行覆盖采集与分析方法，如何启用、统计、解读未命中行、定位冗余或缺失测试。

对应物：行覆盖数据文件与分析笔记 (unity\_test/{DUT}\_line\_coverage\_analysis.md, 如 Adder\_line\_coverage\_analysis.  
allowbreak{}md)。

- dut\_test\_template.md

用途：测试用例的骨架/模板，给出最小可行的结构与编写范式（Arrange-Act-Assert、前置/后置、标记/选择器等）。

对应物：tests/ 下各具体测试文件的基本结构参考。

- dut\_test\_case.md

用途：单个测试用例的撰写规范（命名、输入空间、边界/异常、可重复性、断言质量、日志、标记）。

对应物：tests/ 中具体 test\_xxx.py::test\_yyy 的质量基准与填写要求。

- dut\_test\_program.md

用途：测试计划/测试编排（回归集合、分层/分阶段执行、标记与选择、超时控制、先后顺序、依赖关系）。

对应物：回归集配置、命令/脚本、阶段化执行策略文档。

- dut\_test\_summary.md

用途：测试阶段性/最终总结的结构（通过率、覆盖率、主要问题、修复状态、风险/残留问题、下一步计划）。

对应物：unity\_test/{DUT}\_test\_summary.md (如 Adder\_test\_summary.  
allowbreak{}md) 或报告页面 (output/  
allowbreak{}uc\_test\_report)。

- dut\_bug\_analysis.md

用途：Bug 记录与分析规范（复现步骤、根因分析、影响范围、修复建议、验证状态、标签与追踪）。

对应物：unity\_test/{DUT}\_bug\_analysis.md (如 Adder\_bug\_analysis.  
allowbreak{}md)。

## 11.2 生成产物

### 11.2.1 uc\_test\_report

- uc\_test\_report：由 toffee-test 生成的 index.html 报告，可直接使用浏览器打开。

- 这个报告包含了 Line Coverage 行覆盖率，Functional Coverage 功能覆盖率，测试用例的通过情况，功能点标记具体情况等内容。

### 11.2.2 unity\_test/tests

由大模型根据 `Guide_Doc` 文件夹下的模板文件生成。生成的文件名前缀依据模块不同而不同，以下以 `Adder` 举例。

- `unity_test/tests`: 验证代码文件夹

- `Adder.ignore`

作用：行覆盖率忽略清单。支持忽略整个文件，或以“起止行段”形式忽略代码段。

被谁使用：`Adder_api.py` 的 `set_line_coverage(request, get_coverage_data_path(request, new_path=False), ignore=current_path_file("Adder.ignore"))`。

与 `Guide_Doc` 的关系：对应参考：`dut_line_coverage`.

`allowbreak{}md` (说明如何启用/统计/分析行覆盖，以及忽略规则的意义和使用场景)。

- `Adder_api.py`

作用：测试公共基座，集中放 DUT 构造、覆盖率接线与采样、pytest 基础夹具（fixtures）和示例 API。

- \* `create_dut(request)`: 实例化 DUT、设置覆盖率文件、可选波形、绑定 StepRis 采样。
- \* `AdderEnv`: 封装引脚与常用操作（Step）。
- \* `api_Adder_add`: 对外暴露的测试 API，完成参数校验、信号赋值、推进、读取结果。
- \* `pytest fixtures`: `dut` (模块级，负责覆盖率采样/收集交给 `toffee_test`)、`env` (函数级，给每个 test 一个全新环境)。

与 `Guide_Doc` 的关系：

- \* `dut_fixture.md`: 夹具/环境 (Fixture/Env) 的组织、Step/StepRis 的用法与职责边界。
- \* `dut_api_instruction.md`: API 设计 (命名、参数约束、返回、示例、异常) 和文档规范。
- \* `dut_function_coverage_def.md`: 如何将功能覆盖组接线到 DUT 并在 StepRis 内采样。
- \* `dut_line_coverage.md`: 如何设置行覆盖文件、忽略清单，并将数据上报给 `toffee_test`。

- `Adder_function_coverage_def.`

- `allowbreak{}py`

作用：功能覆盖定义 (Functional Coverage)，声明 FG/FC/CK 并给出 `watch_point` 条件。

- \* 定义覆盖组：FG-API、FG-ARITHMETIC、FG-BIT-WIDTH。

- 每组下定义 FC\_- 和 CK\_- 条件 (如 CK-BASIC/CK-CARRY-IN/CK-OVERFLOW 等)。

- \* `get_coverage_groups(dut)`: 初始化并返回覆盖组列表，供 `Adder_api.py` 绑定与采样。

与 Guide\_Doc 的关系：

\* dut\_function\_coverage\_def.md：覆盖组/覆盖点的组织方式与命名规范、watch\_point 的表达方式。

\* dut\_functions\_and\_checks.md：FG/FC/CK 的命名体系与映射关系来源，测试中用 mark\_function 标记覆盖时需与此保持一致。

- **test\_Adder\_api\_basic.**

**allowbreak{}py**

作用：API 层面的基础功能测试，覆盖典型输入、进位、零值、溢出、边界等。

\* 使用 from Adder\_api import \* 来获取 fixtures (dut/env) 与 API。

\* 在每个测试中通过 env.dut.fc\_cover[ “FG-…” ].mark\_function( “FC-…” , , [ “CK-…” ]) 标注功能覆盖命中关系。与 Guide\_Doc 的关系：

\* dut\_test\_case.md：单测结构（目标/流程/预期）、命名与断言规范、可重复性、标记与日志。

\* dut\_functions\_and\_checks.md：FG/FC/CK 的正确引用与标注。

\* dut\_test\_template.md：docstring 和结构写法的范式来源。

- **test\_Adder\_functional.**

**allowbreak{}py**

作用：功能行为测试（接近“场景/功能项”的角度），比 API 基测覆盖更全面的功能点验证。

\* 同样通过 mark\_function 与 FG/FC/CK 标签体系对齐。与 Guide\_Doc 的关系：

· dut\_test\_case.md：功能类测试的编写规范与断言要求。

· dut\_functions\_and\_checks.md：功能覆盖标注的规范与完整性。

· dut\_test\_template.md：测试函数组织的范式。

- **test\_example.py**

作用：空白样例（脚手架），用于新增测试文件的最小模板参考。与 Guide\_Doc 的关系：

\* dut\_test\_template.md：新建测试文件/函数时的结构、引入方式与标注方法的模板。

### 11.2.3 unity\_test/\*.md

该目录下的各种.md 文件也是大模型根据模板文件生成的。

· unity\_test/\*.md：验证相关文档

- **Adder\_basic\_info.md**

\* 用途：DUT 概览与接口说明（功能、端口、类型、粗粒度功能分类）。

\* 参考: Guide\_Doc/  
allowbreak{}dut\_functions\_and\_checks.  
allowbreak{}md (接口/功能分类用语)、Guide\_Doc/  
allowbreak{}dut\_fixture.  
allowbreak{}md (从验证视角描述 I/O 与 Step 时可参考)。

- Adder\_verification\_needs\_and\_plan.md

\* 用途: 验证需求与计划 (目标、风险点、测试项规划、方法论)。  
\* 参考: Guide\_Doc/  
allowbreak{}dut\_test\_program.  
allowbreak{}md (编排与选择策略)、Guide\_Doc/  
allowbreak{}dut\_test\_case.  
allowbreak{}md (单测质量要求)、Guide\_Doc/  
allowbreak{}dut\_functions\_and\_checks.  
allowbreak{}md (从需求到 FG/FC/CK 的映射)。

- Adder\_functions\_and\_checks.md

\* 用途: FG/FC/CK 真源清单, 测试标注与功能覆盖定义需与此保持一致。  
\* 参考: Guide\_Doc/  
allowbreak{}dut\_functions\_and\_checks.  
allowbreak{}md (结构/命名)、Guide\_Doc/  
allowbreak{}dut\_function\_coverage\_def.  
allowbreak{}md (如何落地为覆盖实现)。

- Adder\_line\_coverage\_analysis.md

\* 用途: 行覆盖率结论与分析, 解释忽略清单、未命中行、补测建议。  
\* 参考: Guide\_Doc/  
allowbreak{}dut\_line\_coverage.  
allowbreak{}md; 配合同目录 tests 下的 Adder.ignore。

- Adder\_bug\_analysis.md

\* 用途: 缺陷分析报告, 按 CK/TC 对应、置信度、根因、修复建议与回归方法撰写。  
\* 参考: Guide\_Doc/  
allowbreak{}dut\_bug\_analysis.  
allowbreak{}md (结构/要素)、Guide\_Doc/  
allowbreak{}dut\_functions\_and\_checks.  
allowbreak{}md (命名一致)。

- Adder\_test\_summary.md

\* 用途：阶段性/最终测试总结（执行统计、覆盖情况、缺陷分布、建议、结论）。

\* 参考：[Guide\\_Doc/](#)

`allowbreak{}dut_test_summary.`

`allowbreak{}md`, 与 [Guide\\_Doc/](#)

`allowbreak{}dut_test_program.`

`allowbreak{}md` 呼应。

## 12 定制功能

### 12.1 添加工具与 MCP Server 工具

面向可修改本仓库代码的高级用户，以下说明如何：

- 添加一个新工具（供本地/Agent 内调用）
- 将工具暴露为 MCP Server 工具（供外部 IDE/客户端调用）
- 控制选择哪些工具被暴露与如何调用

涉及关键位置：

· `ucagent/`

`allowbreak{}tools/`

`allowbreak{}uctool.`

`allowbreak{}py`: 工具基类 UCTool、`to_fastmcp` (LangChain Tool → FastMCP Tool)

· `ucagent/`

`allowbreak{}util/`

`allowbreak{}functions.`

`allowbreak{}py`: `import_and_instance_tools` (按名称导入实例)、`create_verify_mcps` (启动 FastMCP)

· `ucagent/`

`allowbreak{}verify_agent.`

`allowbreak{}py`: 装配工具清单，`start_mcps` 组合并启动 Server

· `ucagent/cli.py / ucagent/`

`allowbreak{}verify_pdb.`

`allowbreak{}py`: 命令行与 TUI 内的 MCP 启动命令

### 12.1.1 1) 工具体系与装配

- 工具基类 UCTool:
  - 继承 LangChain BaseTool，内置: call\_count 计数、call\_time\_out 超时、流式/阻塞提示、MCP Context 注入 (ctx.info)、防重入等。
  - 推荐自定义工具继承 UCTool，获得更好的 MCP 行为与调试体验。
- 运行期装配 (VerifyAgent 初始化):
  - 基础工具: RoleInfo、ReadTextFile
  - 嵌入工具: 参考检索与记忆 (除非 `--no-embed-tools`)
  - 文件工具: 读/写/查找/路径等 (可在 MCP 无文件工具模式下剔除)
  - 阶段工具: 由 StageManager 按工作流动态提供
  - 外部工具: 来自配置项 `ex_tools` 与 CLI `--ex-tools` (通过 `import_and_instance_tools` 零参实例化)
- 名称解析:
  - 短名: 类/工厂函数需在 `ucagent/allowbreak{}tools/allowbreak{}__init__.`  
`allowbreak{}py` 导出 (例如 `from .mytool import HelloTool`)，即可在 `ex_tools` 写 `HelloTool`
  - 全路径: `mypkg.allowbreak{}mytools.allowbreak{}HelloTool / mypkg.allowbreak{}mytools.allowbreak{}Factory`

### 12.1.2 2) 添加一个新工具 (本地/Agent 内)

规范要求:

- 唯一 name、清晰 description
- 使用 pydantic BaseModel 定义 args\_schema (MCP 转换依赖)
- 实现 \_run (同步) 或 \_arun (异步); 继承 UCTool 可直接获得超时、流式与 ctx 注入

示例 1: 同步工具 (计数问候)

```
from pydantic import BaseModel, Field
from ucagent.tools.uctool import UCTool

class HelloArgs(BaseModel):
```

```

    who: str = Field(..., description=" 要问候的人")

class HelloTool(UCTool):
    name: str = "Hello"
    description: str = " 向指定对象问候，并统计调用次数"
    args_schema = HelloArgs

    def _run(self, who: str, run_manager=None) -> str:
        return f"Hello, {who}! (called {self.call_count+1} times)"

```

注册与使用：

- 临时：--ex-tools mypkg.mytools.HelloTool
- 持久：项目 config.yaml

```

ex_tools:
- mypkg.mytools.HelloTool

```

(可选) 短名注册：在 ucagent/  
allowbreak{}tools/  
allowbreak{}\_\_init\_\_.  
allowbreak{}py 导出 HelloTool 后，可写 --ex-tools HelloTool。

示例 2：异步流式工具（ctx.info + 超时）

```

from pydantic import BaseModel, Field
from ucagent.tools.uctool import UCTool
import asyncio

class ProgressArgs(BaseModel):
    steps: int = Field(5, ge=1, le=20, description=" 进度步数")

class ProgressTool(UCTool):
    name: str = "Progress"
    description: str = " 演示流式输出与超时处理"
    args_schema = ProgressArgs

    async def _arun(self, steps: int, run_manager=None):
        for i in range(steps):

```

```

        self.put_alive_data(f"step {i+1}/{steps}") # 供阻塞提示
        ↵ 示/日志缓冲
        await asyncio.sleep(0.5)
    return "done"

```

说明：UCTool.invoke 会在 MCP 模式下注入 ctx，并启动阻塞提示线程；当 sync\_block\_log\_to\_client=True 时会周期性 ctx.info 推送日志，超时后返回错误与缓冲日志。

### 12.1.3 3) 暴露为 MCP Server 工具

```

工具 → MCP 转换 (ucagent/
allowbreak{}tools/
allowbreak{}uctool.
allowbreak{}py:
allowbreak{}:
allowbreak{}to_fastmcp):

```

- 必须：args\_schema 继承 BaseModel；不支持“注入参数”签名。
- UCTool 子类会得到 context\_kwarg=“ctx”的 FastMCP 工具，具备流式交互能力。

Server 端启动：

- VerifyAgent.start\_mcps 组合工具：tool\_list\_base + tool\_list\_task + tool\_list\_ext + [tool\_list\_file]
- ucagent/
 

```
allowbreak{}util/
allowbreak{}functions.
allowbreak{}py:
allowbreak{}:
allowbreak{}create_verify_mcps 将工具序列转换为 FastMCP 工具并启动 uvicorn (mcp.
allowbreak{}streamable_http_app())。
```

如何选择暴露范围：

- CLI：
  - 启动（含文件工具）：--mcp-server
  - 启动（无文件工具）：-

```
allowbreak{}-
allowbreak{}mcp-
```

- allowbreak{}server-
- allowbreak{}no-
- allowbreak{}file-
- allowbreak{}tools
- 地址: --mcp-server-host, 端口: --mcp-server-port
- TUI 命令: start\_mcp\_server [host] [port] / start\_mcp\_server\_no\_file\_ops [host] [port]

#### 12.1.4 4) 客户端调用流程

FastMCP Python 客户端 (参考 tests/

allowbreak{}test\_mcps.

allowbreak{}py):

```
from fastmcp import Client

client = Client("http://127.0.0.1:5000/mcp", timeout=10)
print(client.list_tools())
print(client.call_tool("Hello", {"who": "UCAgent"}))
```

IDE/Agent (Claude Code、Copilot、Qwen Code 等): 将 httpUrl 指向 http:

allowbreak{}/

allowbreak{}/

allowbreak{}<host>:

allowbreak{}<port>/

allowbreak{}mcp, 即可发现并调用工具。

#### 12.1.5 5) 生命周期、并发与超时

- 计数: UCTool 内置 call\_count; 非 UCTool 工具由 import\_and\_instance\_tools 包装计数。
- 并发保护: is\_in\_streaming/is\_alive\_loop 防止重入; 同一实例不允许并发执行。
- 超时: call\_time\_out (默认 20s) + 客户端 timeout; 阻塞时可用 put\_alive\_data + sync\_block\_log\_to\_client=True 推送心跳。

#### 12.1.6 6) 配置策略与最佳实践

- ex\_tools 列表为“整体覆盖”，项目 config.yaml 需写出完整清单。

- 短名 vs 全路径：短名更便捷，全路径适用于私有包不修改本仓库时。
- 无参构造/工厂：装配器直接调用 (...)()，复杂配置建议在工厂内部处理（读取环境/配置文件）。
- 文件写权限：MCP 无文件工具模式下不要暴露写类工具；如需写入，请在本地 Agent 内使用或显式允许写目录。

#### 12.1.6.1 通过环境变量注入外部工具 (EX\_TOOLS)

配置文件支持 Bash 风格环境变量占位: \${VAR: default}。你可以让 ex\_tools 从环境变量注入工具类列表（支持模块全名或 ucagent.tools 下的短名）。

1. 在项目的 config.yaml 或用户级 / allowbreak{}。  
allowbreak{}ucagent/  
allowbreak{}setting.  
allowbreak{}yaml 中写入：

```
ex_tools: ${EX_TOOLS: []}
```

2. 用环境变量提供列表（必须是可被 YAML 解析的数组字面量）：

```
export EX_TOOLS='["SqThink", "HumanHelp"]'  
# 或使用完整类路径：  
# export  
→ EX_TOOLS='["ucagent.tools.extool.SqThink", "ucagent.tools.human.HumanHelp"]'
```

3. 启动后本地对话与 MCP Server 中都会出现这些工具。短名需要在 ucagent/  
allowbreak{}tools/  
allowbreak{}\_\_init\_\_.  
allowbreak{}py 导出；否则请使用完整模块路径。
4. 与 CLI 的 --ex-tools 选项是合并关系（两边都会被装配）。

#### 12.1.7 7) 常见问题排查

- 工具未出现在 MCP 列表：未被装配 (ex\_tools 未配置/未导出)、args\_schema 非 BaseModel、Server 未按预期启动。
- 调用报“注入参数不支持”：工具定义包含 LangChain 的 injected args；请改成显式 args\_schema 参数。
- 超时：调大 call\_time\_out 或客户端 timeout；在长任务中输出进度维持心跳。

- 短名无效：未在 ucagent/  
allowbreak{}tools/  
allowbreak{}\_\_init\_\_.  
allowbreak{}py 导出；改用全路径或补导出。

## 13 工具列表

以下为当前仓库内内置工具（UCTool 家族）的概览，按功能类别归纳：名称（调用名）、用途与参数说明（字段：类型—含义）。

提示：

- 带有“文件写”能力的工具仅在本地/允许写模式下可用；MCP 无文件工具模式不会暴露写类工具。
- 各工具均基于 args\_schema 校验参数，MCP 客户端将根据 schema 生成参数表单。

### 13.1 基础/信息类

- RoleInfo (RoleInfo)
  - 用途：返回当前代理的角色信息（可在启动时自定义 role\_info）。
  - 参数：无
- HumanHelp (HumanHelp)
  - 用途：向人类请求帮助（仅在确实卡住时使用）。
  - 参数：
    - \* message: str — 求助信息

### 13.2 规划/ToDo 类

- CreateToDo
  - 用途：创建 ToDo（覆盖旧 ToDo）。
  - 参数：
    - \* task\_description: str — 任务描述
    - \* steps: List[str] — 步骤（1–20 步）
- CompleteToDoSteps
  - 用途：将指定步骤标记为完成，可附加备注。

- 参数:
  - \* completed\_steps: List[int] — 完成的步骤序号 (1-based)
  - \* notes: str — 备注
- UndoToDoSteps
  - 用途: 撤销步骤完成状态, 可附加备注。
  - 参数:
    - \* steps: List[int] — 撤销的步骤序号 (1-based)
    - \* notes: str — 备注
- ResetToDo
  - 用途: 重置/清空当前 ToDo。
  - 参数: 无
- GetToDoSummary / ToDoState
  - 用途: 获取 ToDo 摘要 / 看板状态短语。
  - 参数: 无

### 13.3 记忆/检索类

- SemanticSearchInGuidDoc (SemanticSearchInGuidDoc)
  - 用途: 在 Guide\_Doc/项目文档中做语义检索, 返回最相关片段。
  - 参数:
    - \* query: str — 查询语句
    - \* limit: int — 返回条数 (1-100, 默认 3)
- MemoryPut
  - 用途: 按 scope 写入长时记忆。
  - 参数:
    - \* scope: str — 命名空间/范围 (如 general/task-specific)
    - \* data: str — 内容 (可为 JSON 文本)
- MemoryGet
  - 用途: 按 scope 检索记忆。
  - 参数:
    - \* scope: str — 命名空间/范围
    - \* query: str — 查询语句
    - \* limit: int — 返回条数 (1-100, 默认 3)

### 13.4 测试/执行类

- RunPyTest (RunPyTest)
  - 用途：在指定目录/文件下运行 pytest，支持返回 stdout/stderr。
  - 参数：
    - \* test\_dir\_or\_file: str — 测试目录或文件
    - \* pytest\_ex\_args: str — 额外 pytest 参数（如 “-v --capture=no”）
    - \* return\_stdout: bool — 是否返回标准输出
    - \* return\_stderr: bool — 是否返回标准错误
    - \* timeout: int — 超时秒数（默认 15）
- RunUnityChipTest (RunUnityChipTest)
  - 用途：面向 UnityChip 项目封装的测试执行，产生 toffee\_report.json 等结果。
  - 参数：同 RunPyTest；另含内部字段（workspace/result\_dir/result\_json\_path）。

### 13.5 文件/路径/文本类

- SearchText (SearchText)
  - 用途：在工作区内按文本搜索，支持通配与正则。
  - 参数：
    - \* pattern: str — 搜索模式（明文/通配/正则）
    - \* directory: str — 相对目录（为空则全仓；填文件则仅搜该文件）
    - \* max\_match\_lines: int — 每个文件返回的最大匹配行数（默认 20）
    - \* max\_match\_files: int — 返回的最大文件数（默认 10）
    - \* use\_regex: bool — 是否使用正则
    - \* case\_sensitive: bool — 区分大小写
    - \* include\_line\_numbers: bool — 返回是否带行号
- FindFiles (FindFiles)
  - 用途：按通配符查找文件。
  - 参数：
    - \* pattern: str — 文件名模式（fnmatch 通配）
    - \* directory: str — 相对目录（为空则全仓）
    - \* max\_match\_files: int — 返回最大文件数（默认 10）
- PathList (PathList)

- 用途：列出目录结构（可限制深度）。
- 参数：
  - \* path: str — 目录（相对 workspace）
  - \* depth: int — 深度（-1 全部，0 当前）
- ReadBinFile (ReadBinFile)
  - 用途：读取二进制文件（返回 [BIN\_DATA]）。
  - 参数：
    - \* path: str — 文件路径（相对 workspace）
    - \* start: int — 起始字节（默认 0）
    - \* end: int — 结束字节（默认 -1 表示 EOF）
- ReadTextFile (ReadTextFile)
  - 用途：读取文本文件（带行号，返回 [TXT\_DATA]）。
  - 参数：
    - \* path: str — 文件路径（相对 workspace）
    - \* start: int — 起始行（1-based， 默认 1）
    - \* count: int — 行数（-1 到文件末尾）
- EditTextFile (EditTextFile)
  - 用途：编辑/创建文本文件，模式：replace/overwrite/append。
  - 参数：
    - \* path: str — 文件路径（相对 workspace，不存在则创建）
    - \* data: str — 写入的文本（None 表示清空）
    - \* mode: str — 编辑模式（replace/overwrite/append， 默认 replace）
    - \* start: int — replace 模式的起始行（1-based）
    - \* count: int — replace 模式替换行数（-1 到末尾，0 插入）
    - \* preserve\_indent: bool — replace 时是否保留缩进
- CopyFile (CopyFile)
  - 用途：复制文件；可选覆盖。
  - 参数：
    - \* source\_path: str — 源文件
    - \* dest\_path: str — 目标文件
    - \* overwrite: bool — 目标存在时是否覆盖
- MoveFile (MoveFile)
  - 用途：移动/重命名文件；可选覆盖。

- 参数:
  - \* source\_path: str —源文件
  - \* dest\_path: str —目标文件
  - \* overwrite: bool —目标存在时是否覆盖
- DeleteFile (DeleteFile)
  - 用途: 删除文件。
  - 参数:
    - \* path: str —文件路径
- CreateDirectory (.CreateDirectory)
  - 用途: 创建目录 (递归)。
  - 参数:
    - \* path: str —目录路径
    - \* parents: bool —递归创建父目录
    - \* exist\_ok: bool —已存在是否忽略
- ReplaceStringInFile (ReplaceStringInFile)
  - 用途: 精确字符串替换 (强约束匹配; 可新建文件)。
  - 参数:
    - \* path: str —目标文件
    - \* old\_string: str —需要被替换的完整原文 (含上下文, 精确匹配)
    - \* new\_string: str —新内容
- GetFileInfo (GetFileInfo)
  - 用途: 获取文件信息 (大小、修改时间、人类可读尺寸等)。
  - 参数:
    - \* path: str —文件路径

## 13.6 扩展示例

- SimpleReflectionTool (SimpleReflectionTool)
  - 用途: 示例型“自我反思”工具 (来自 extool.py), 可作为扩展参考。
  - 参数:
    - \* message: str —自我反思文本

备注:

- 工具调用超时默认 20s (具体工具可重写); 长任务请周期性输出进度避免超时。
- MCP 无文件工具模式下默认不暴露写类工具; 如需写入, 建议在本地 Agent 模式或按需限制可写目录。

## 14 工作流

整体采用“按阶段渐进推进”的方式, 每个阶段都有明确目标、产出与通过标准; 完成后用工具 Check 验证并用 Complete 进入下一阶段。若阶段包含子阶段, 需按顺序逐一完成子阶段并各自通过 Check。

- 顶层阶段总数: 11 (见 ucagent/  
allowbreak{}lang/  
allowbreak{}zh/  
allowbreak{}config/  
allowbreak{}default.  
allowbreak{}yaml)
- 推进原则: 未通过的阶段不可跳转; 可用工具 CurrentTips 获取当前阶段详细指导; 需要回补时可用 GotoStage 回到指定阶段。
- 三种跳/不跳过阶段方法:
  - 在项目根 config.yaml 的某个 stage 字段下面 -name 元素里的 skip 键配置 true/false 来跳过/不跳过。
  - 命令行启动时可用 --skip/- -unskip someStage 来控制跳过/不跳过某阶段。
  - 在 tui 启动后可用 skip\_stage/unskip\_stage someStage 来控制临时跳过/不跳过某阶段。

### 14.1 整体流程概览 (11 个阶段)

目前的流程包含:

1. 需求分析与验证规划 → 2) {DUT} 功能理解 → 3) 功能规格分析与测试点定义 → 4) 测试平台基础架构设计 → 5) 功能覆盖率模型实现 → 6) 基础 API 实现 → 7) 基础 API 功能测试 → 8) 测试框架脚手架 → 9) 全面验证执行与缺陷分析 → 10) 代码行覆盖率分析与提升 (默认跳过, 可启用) → 11) 验证审查与总结

以实际的工作流为准, 下图仅供参考。

说明: 以下路径中的默认为工作目录下的输出目录名 (默认 unity\_test)。例如文档输出到 <workspace>/

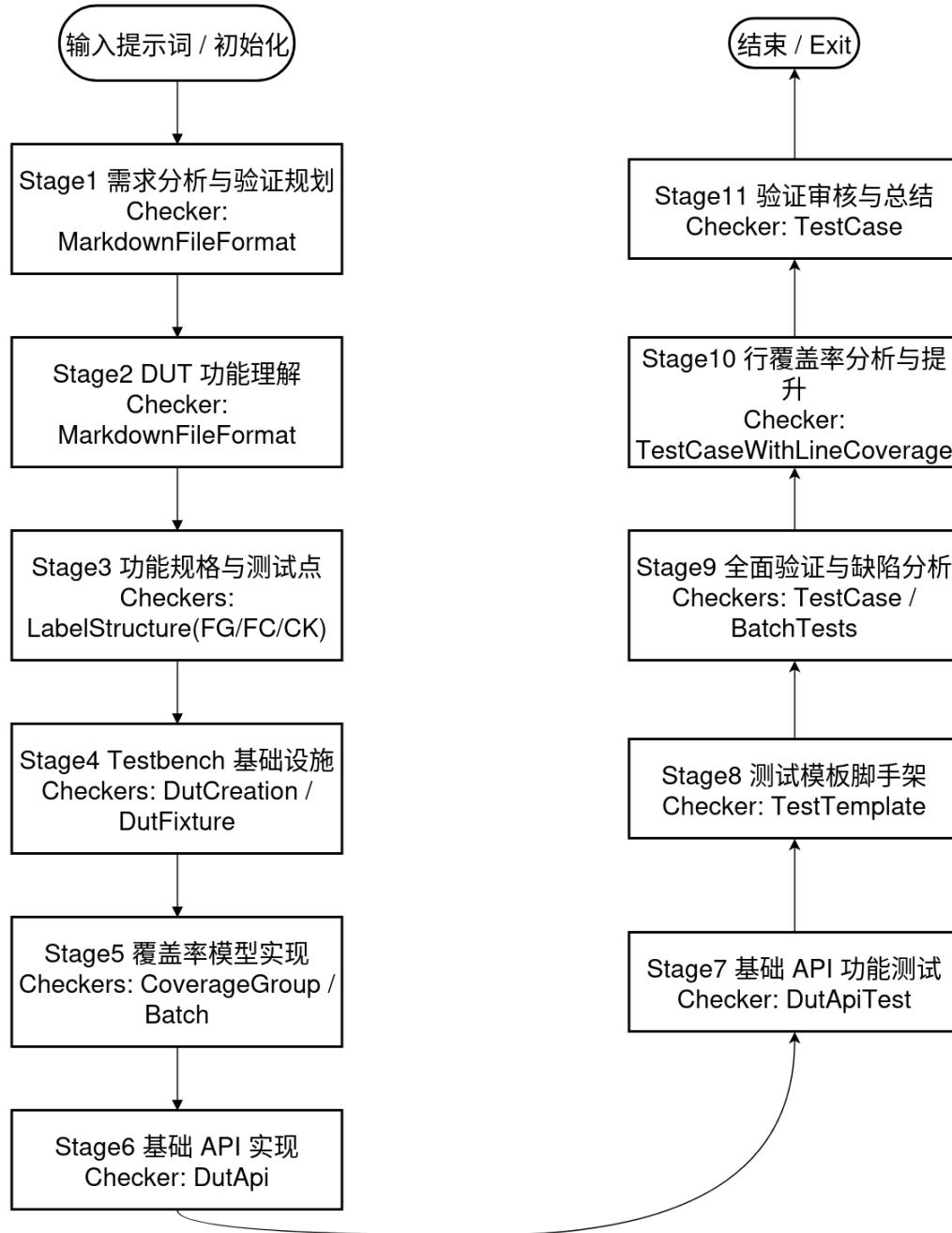


图 6: 工作流图

```
allowbreak{}unity_test/  
allowbreak{}。
```

---

## 阶段 1：需求分析与验证规划

- 目标：理解任务、明确验证范围与策略。
- 怎么做：
  - 阅读 {DUT}/README.md，梳理“需要测哪些功能/输入输出/边界与风险”。
  - 形成可执行的验证计划与目标清单。
- 产出：<OUT>/  

```
allowbreak{}{DUT}_verification_needs_and_plan.  
allowbreak{}md (中文撰写)。
```
- 通过标准：文档存在、结构规范（自动检查 markdown\_file\_check）。
- 检查器：
  - UnityChipCheckerMarkdownFormat
    - \* 作用：校验 Markdown 文件存在与格式，禁止把换行写成字面量 \n。
    - \* 参数：
      - markdown\_file\_list (str | List[str]): 待检查的 MD 文件路径或路径列表。示例：{  
 OUT}/  
 allowbreak{}{DUT}\_verification\_needs\_and\_plan.  
 allowbreak{}md
      - no\_line\_break (bool): 是否禁止把换行写成字面量 \n； true 表示禁止。

## 阶段 2：{DUT} 功能理解

- 目标：掌握 DUT 的接口与基本信息，明确是组合/时序电路。
- 怎么做：
  - 阅读 {DUT}/README.md 与 {DUT}/  

```
allowbreak{}__init__.  
allowbreak{}py。
```
  - 分析 IO 端口、时钟/复位需求与功能范围。
- 产出：<OUT>/  

```
allowbreak{}{DUT}_basic_info.  
allowbreak{}md。
```
- 通过标准：文档存在、格式规范（markdown\_file\_check）。
- 检查器：
  - UnityChipCheckerMarkdownFormat
    - \* 作用：校验 Markdown 文件存在与格式，禁止把换行写成字面量 \n。

\* 参数:

- markdown\_file\_list (str | List[str]): 待检查的 MD 文件路径或路径列表。示例: {OUT}/allowbreak{}{DUT}\_basic\_info.allowbreak{}md
- no\_line\_break (bool): 是否禁止把换行写成字面量 \n; true 表示禁止。

### 阶段 3: 功能规格分析与测试点定义 (含子阶段 FG/FC/CK)

- 目标: 把功能分组 (FG)、功能点 (FC) 和检测点 (CK) 结构化, 作为后续自动化的依据。
- 怎么做:
  - 阅读 {DUT}/\*.md 与已产出文档, 建立 {DUT}\_functions\_and\_checks.allowbreak{}md 的 FG/FC/CK 结构。
  - 规范标签: <FG-组名>、<FC-功能名>、<CK-检测名>, 每个功能点至少 1 个检测点。
- 子阶段:
  - 3.1 功能分组与层次 (FG): 检查器 UnityChipCheckerLabelStructure(FG)
  - 3.2 功能点定义 (FC): 检查器 UnityChipCheckerLabelStructure(FC)
  - 3.3 检测点设计 (CK): 检查器 UnityChipCheckerLabelStructure(CK)
- 产出: <OUT>/allowbreak{}{DUT}\_functions\_and\_checks.allowbreak{}md。
- 通过标准: 三类标签结构均通过对应检查。
- 对应检查器 (默认配置):
  - 3.1 UnityChipCheckerLabelStructure
    - \* 作用: 解析 {DUT}\_functions\_and\_checks.allowbreak{}md 中的标签结构并校验层级与数量 (FG)。
    - \* 参数:
      - doc\_file (str): 功能/检查点文档路径。示例: {OUT}/allowbreak{}{DUT}\_functions\_and\_checks.allowbreak{}md
      - leaf\_node ("FG" | "FC" | "CK"): 需要校验的叶子类型。示例: "FG"
      - min\_count (int, 默认 1): 该叶子类型的最小数量阈值。
      - must\_have\_prefix (str, 默认 "FG-API"): FG 名称要求的前缀, 用于规范化分组命名。
  - 3.2 UnityChipCheckerLabelStructure
    - \* 作用: 解析文档并校验功能点定义 (FC)。
    - \* 参数:
      - doc\_file (str): 功能/检查点文档路径。示例: {OUT}/allowbreak{}{DUT}\_functions\_and\_checks.

- `allowbreak{}md`
- `leaf_node` ( “FG” | “FC” | “CK” ): 需要校验的叶子类型。示例: “FC”
- `min_count` (int, 默认 1): 该叶子类型的最小数量阈值。
- `must_have_prefix` (str, 默认 “FG-API” ): 所属 FG 的前缀规范, 用于一致性检查。
- 3.3 UnityChipCheckerLabelStructure
  - \* 作用: 解析文档并校验检测点设计 (CK), 并缓存 CK 列表用于后续分批实现。
  - \* 参数:
    - `doc_file` (str): 功能/检查点文档路径。示例: {OUT}/  
`allowbreak{}{DUT}_functions_and_checks.`  
`allowbreak{}md`
    - `leaf_node` ( “FG” | “FC” | “CK” ): 需要校验的叶子类型。示例: “CK”
    - `data_key` (str): 共享数据键名, 用于缓存 CK 列表 (供后续分批实现使用)。示例:  
“COVER\_GROUP\_DOC\_CK\_LIST”
    - `min_count` (int, 默认 1): 该叶子类型的最小数量阈值。
    - `must_have_prefix` (str, 默认 “FG-API” ): 所属 FG 的前缀规范, 用于一致性检查。

#### 阶段 4: 测试平台基础架构设计 (fixture/API 框架)

- 目标: 提供统一的 DUT 创建与测试生命周期管理能力。
- 怎么做:
  - 在 <OUT>/  
`allowbreak{}tests/`  
`allowbreak{}{DUT}_api.`  
`allowbreak{}py` 实现 `create_dut()`; 时序电路配置时钟 (InitClock), 组合电路无需时钟。
  - 实现 pytest fixture `dut`, 负责初始化/清理与可选的波形/行覆盖率开关。
- 产出: <OUT>/  
`allowbreak{}tests/`  
`allowbreak{}{DUT}_api.`  
`allowbreak{}py` (含注释与文档字符串)。
- 通过标准: DUT 创建与 fixture 检查通过 (UnityChipCheckerDutCreation / UnityChipCheckerDutFixture)。
- 子阶段检查器:
  - DUT 创建: UnityChipCheckerDutCreation
    - \* 作用: 校验 `{DUT}_api.py` 中的 `create_dut(request)` 是否实现规范 (签名、时钟/复位、覆盖率路径等约定)。
    - \* 参数:
      - `target_file` (str): DUT API 与 fixture 所在文件路径。示例: {OUT}/  
`allowbreak{}tests/`

- ```

    allowbreak{}{DUT}_api.
    allowbreak{}py

```
- dut fixture: UnityChipCheckerDutFixture
    - \* 作用: 校验 pytest fixture dut 的生命周期管理、yield/清理, 以及覆盖率收集调用是否到位。
    - \* 参数:
      - target\_file (str): 包含 dut fixture 的文件路径。示例: {OUT}/
   
allowbreak{}tests/
   
allowbreak{}{DUT}\_api.
   
allowbreak{}py
  - env fixture: UnityChipCheckerEnvFixture
    - \* 作用: 校验 env\* 系列 fixture 的存在、数量与 Bundle 封装是否符合要求。
    - \* 参数:
      - target\_file (str): 包含 env\* 系列 fixture 的文件路径。示例: {OUT}/
   
allowbreak{}tests/
   
allowbreak{}{DUT}\_api.
   
allowbreak{}py
      - min\_env (int, 默认 1): 至少需要存在的 env\* fixture 数量。示例: 1
      - force\_bundle (bool, 当前未使用): 是否强制要求 Bundle 封装。

覆盖率路径规范 (重要):

- 在 create\_dut(request) 中, 必须通过 get\_coverage\_data\_path(request, new\_path=True) 获取新的行覆盖率文件路径, 并传入 dut.
   
allowbreak{}SetCoverage().
   
allowbreak{}.
   
allowbreak{}.
   
allowbreak{}).
- 在 dut fixture 的清理阶段, 必须通过 get\_coverage\_data\_path(request, new\_path=False) 获取已有路径, 并调用 set\_line\_coverage(request, <path>, ignore=...) 写入统计。
- 若缺失上述调用, 检查器会直接报错, 并给出修复提示 (含 tips\_of\_get\_coverage\_data\_pa th 示例)。

阶段 5: 功能覆盖率模型实现

- 目标: 将 FG/FC/CK 转为可统计的覆盖结构, 支撑进度度量与回归。
- 怎么做:
  - 在 <OUT>/
   
allowbreak{}tests/
   
allowbreak{}{DUT}\_function\_coverage\_def.

- allowbreak{}py 实现 get\_coverage\_groups(dut)。
  - 为每个 FG 建立 CovGroup; 为 FC/CK 建 watch\_point 与检查函数 (优先用 lambda, 必要时普通函数)。
- 子阶段:
  - 5.1 覆盖组创建 (FG)
  - 5.2 覆盖点与检查实现 (FC/CK), 支持“分批实现”提示 (COMPLETED\_POINTS/TOTAL\_POINTS)。
- 产出: <OUT>/  
allowbreak{}tests/  
allowbreak{}{DUT}\_function\_coverage\_def.  
allowbreak{}py。
- 通过标准: CoverageGroup 检查 (FG/FC/CK) 与批量实现检查通过。
- 子阶段检查器:
  - 5.1 UnityChipCheckerCoverageGroup
    - \* 作用: 比对覆盖组定义与文档 FG 一致性。
    - \* 参数:
      - test\_dir (str): 测试目录根路径。示例: {OUT}/tests
      - cov\_file (str): 覆盖率模型定义文件路径。示例: {OUT}/  
allowbreak{}tests/  
allowbreak{}{DUT}\_function\_coverage\_def.  
allowbreak{}py
      - doc\_file (str): 功能/检查点文档路径。示例: {OUT}/  
allowbreak{}{DUT}\_functions\_and\_checks.  
allowbreak{}md
      - check\_types (str | List[str]): 检查的类型集合。示例: "FG"
  - 5.2 UnityChipCheckerCoverageGroup
    - \* 作用: 比对覆盖点/检查点实现与文档 FC/CK 一致性。
    - \* 参数:
      - test\_dir (str): 测试目录根路径。示例同上
      - cov\_file (str): 覆盖率模型定义文件路径。示例同上
      - doc\_file (str): 功能/检查点文档路径。示例同上
      - check\_types (List[str]): 检查类型集合。示例: ["FC", "CK"]
  - 5.2 (分批) UnityChipCheckerCoverageGroupBatchImplementation
    - \* 作用: 按 CK 分批推进实现与对齐检查, 维护进度 (TOTAL/COMPLETED)。
    - \* 参数:
      - test\_dir (str): 测试目录根路径。
      - cov\_file (str): 覆盖率模型定义文件路径。
      - doc\_file (str): 功能/检查点文档路径。

- batch\_size (int, 默认 20): 每批实现与校验的 CK 数量上限。示例: 20
- data\_key (str): 共享数据键名, 用于读取 CK 列表。示例: "COVER\_GROUP\_DOC\_C\_K\_LIST"

#### 阶段 6: 基础 API 实现

- 目标: 用 `api_{DUT}_*` 前缀提供可复用的操作封装, 隐藏底层信号细节。
- 怎么做:
  - 在 <OUT>/  
`allowbreak{}tests/`  
`allowbreak{}{DUT}_api.`  
`allowbreak{}py` 实现至少 1 个基础 API; 建议区分“底层功能 API”与“任务功能 API”。
  - 补充详细 docstring: 功能、参数、返回值、异常。
- 产出: <OUT>/  
`allowbreak{}tests/`  
`allowbreak{}{DUT}_api.`  
`allowbreak{}py`。
- 通过标准: UnityChipCheckerDutApi 通过 (前缀必须为 `api_{DUT}_`)。
- 检查器:
  - UnityChipCheckerDutApi
    - \* 作用: 扫描/校验 `api_{DUT}_*` 函数的数量、命名、签名与 docstring 完整度。
    - \* 参数:
      - api\*prefix (str): API 前缀匹配表达式。建议: "`api*{DUT}\_`"
      - target\_file (str): API 定义所在文件。示例: {OUT}/  
`allowbreak{}tests/`  
`allowbreak{}{DUT}_api.`  
`allowbreak{}py`
      - min\_apis (int, 默认 1): 至少需要的 API 数量。

#### 阶段 7: 基础 API 功能正确性测试

- 目标: 为每个已实现 API 编写至少 1 个基础功能用例, 并标注覆盖率。
- 怎么做:
  - 在 <OUT>/  
`allowbreak{}tests/`  
`allowbreak{}test_{DUT}_api_*.`  
`allowbreak{}py` 新建测试; 导入 `from {DUT}_api import *`。
  - 每个测试函数的第一行: `dut.fc_cover['FG-API'].mark_function('FC-API-NAME', test_func, ['CK-XXX'])`。
  - 设计典型/边界/异常数据, 断言预期输出。

- 用工具 RunTestCases 执行与回归。
- 产出: <OUT>/  

```
allowbreak{}tests/
allowbreak{}test_{DUT}_api_*.
```

`allowbreak{}py` 与缺陷记录（若发现 bug）。
- 通过标准: UnityChipCheckerDutApiTest 通过（覆盖、用例质量、文档记录齐备）。
- 检查器:
  - UnityChipCheckerDutApiTest
    - \* 作用: 运行 pytest 并检查每个 API 至少 1 个基础功能用例且正确覆盖标记；核对缺陷记录与文档一致。
    - \* 参数:
      - api\*prefix (str): API 前缀匹配表达式。建议: "api\*{DUT}\\_"
      - target\_file\_api (str): API 文件路径。示例: {OUT}/  
`allowbreak{}tests/
allowbreak{}{DUT}_api.`
      - allowbreak{}py
      - target\*file\_tests (str): 测试文件 Glob。示例: {OUT}/  
`allowbreak{}tests/
allowbreak{}test*{DUT}
_api
*.`
      - allowbreak{}py
      - doc\_func\_check (str): 功能/检查点文档。示例: {OUT}/  
`allowbreak{}{DUT}_functions_and_checks.
allowbreak{}md`
      - doc\_bug\_analysis (str): 缺陷分析文档。示例: {OUT}/  
`allowbreak{}{DUT}_bug_analysis.
allowbreak{}md`
      - min\_tests (int, 默认 1): 单 API 最少测试用例数。
      - timeout (int, 默认 15): 单次测试运行超时（秒）。

## 阶段 8: 测试框架脚手架构建

- 目标: 为尚未实现的功能点批量生成“占位”测试模板，确保覆盖版图完整。
- 怎么做:
  - 依据 `{DUT}_functions_and_checks.`  
`allowbreak{}md`, 在 `<OUT>/tests/` 创建 `test_*.py`, 文件与用例命名语义化。
  - 每个函数首行标注覆盖率 mark; 补充 TODO 注释说明要测什么; 末尾添加 `assert False, 'Not implemented'` 防误通过。

- 产出：批量测试模板；覆盖率进度指标 (COVERED\_CKS/TOTAL\_CKS)。
- 通过标准：UnityChipCheckerTestTemplate 通过 (结构/标记/说明完整)。
- 检查器：
  - UnityChipCheckerTestTemplate
    - \* 作用：检查模板文件/用例结构、覆盖标记、TODO 注释与防误通过断言；统计覆盖进度。
    - \* 参数：
      - doc\_func\_check (str): 功能/检查点文档路径。示例：{OUT}/allowbreak{}{DUT}\_functions\_and\_checks.  
allowbreak{}md
      - test\_dir (str): 测试目录根路径。示例：{OUT}/tests
      - ignore\*ck\_prefix (str): 统计覆盖时忽略的 CK 前缀 (通常为基础 API 的用例)。示例："test\_api\*{DUT}\\_"
      - data\_key (str): 共享数据键名，用于生成/读取模板实现进度。示例："TEST\_TEMP\_LATE\_IMP\_REPORT"
      - batch\_size (int, 默认 20): 每批模板检查数量。
      - min\_tests (int, 默认 1): 最少要求的模板测试数。
      - timeout (int, 默认 15): 测试运行超时 (秒)。

#### 阶段 9：全面验证执行与缺陷分析

- 目标：将模板填充为真实测试，系统发现并分析 DUT bug。
- 怎么做：
  - 在 `test_*.py` 填充逻辑，优先通过 API 调用，不直接操纵底层信号。
  - 设计充分数据并断言；用 RunTestCases 运行；对 Fail 进行基于源码的缺陷定位与记录。
- 子阶段：
  - 9.1 分批测试用例实现与对应缺陷分析 (COMPLETED\_CASES/TOTAL\_CASES)。
- 产出：成体系的测试集与 /allowbreak{}{DUT}\_bug\_analysis.  
allowbreak{}md。
- 通过标准：UnityChipCheckerTestCase (质量/覆盖/缺陷分析) 通过。
- 检查器：
  - 父阶段：UnityChipCheckerTestCase
    - \* 作用：运行整体测试并对照功能/缺陷文档检查质量、覆盖与记录一致性。
    - \* 参数：
      - doc\_func\_check (str): 功能/检查点文档路径。示例：{OUT}/allowbreak{}{DUT}\_functions\_and\_checks.  
allowbreak{}md
      - doc\_bug\_analysis (str): 缺陷分析文档路径。示例：{OUT}/allowbreak{}{DUT}\_bug\_analysis.

- `allowbreak{}md`
- `test_dir` (str): 测试目录根路径。示例: `{OUT}/tests`
- `min_tests` (int, 默认 1): 最少要求的测试用例数量。
- `timeout` (int, 默认 15): 测试运行超时 (秒)。
- 子阶段 (分批实现): `UnityChipCheckerBatchTestsImplementation`
  - \* 作用: 分批将模板落地为真实用例并回归, 维护实现进度与报告。
  - \* 参数:
    - `doc_func_check` (str): 功能/检查点文档路径。
    - `doc_bug_analysis` (str): 缺陷分析文档路径。
    - `test_dir` (str): 测试目录根路径。
    - `ignore*ck_prefix` (str): 统计覆盖时忽略的 CK 前缀。示例: `"test_api*{DUT}\_"`
    - `batch_size` (int, 默认 10): 每批转化并执行的用例数量。
    - `data_key` (str): 共享数据键名 (必填), 用于保存分批实现进度。示例: `"TEST_TEMPLATE_IMP_REPORT"`
    - `pre_report_file` (str): 历史进度报告路径。示例: `{OUT}/allowbreak{}{DUT}/allowbreak{.}`
    - `allowbreak{}TEST_TEMPLATE_IMP_REPORT.`
    - `allowbreak{}json`
    - `timeout` (int, 默认 15): 测试运行超时 (秒)。

TC bug 标注规范与一致性 (与文档/报告强关联):

- 术语: 统一使用“TC bug”(不再使用“CK bug”)。

- 标注结构:
 

```

<FG-
allowbreak{}*>/
allowbreak{}<FC-
allowbreak{}*>/
allowbreak{}<CK-
allowbreak{}*>/
allowbreak{}<BG-
allowbreak{}NAME-
allowbreak{}XX>/
allowbreak{}<TC-
allowbreak{}test_file.
allowbreak{}py:
allowbreak{}:
allowbreak{}[ClassName]:
allowbreak{}:
```

`allowbreak{}test_case>`; 其中 BG 的置信度 XX 为 0–100 的整数。

- 失败用例与文档关系:

- 文档中出现的 `<TC-*>` 必须能与测试报告中的失败用例一一对应（文件名/类名/用例名匹配）。
- 失败的测试用例必须标注其关联检查点（CK），否则会被判定为“未标记”。
- 若存在失败用例未在 bug 文档中记录，将被提示为“未文档化的失败用例”。

## 阶段 10：代码行覆盖率分析与提升（默认跳过，可启用）

- 目标：回顾未覆盖代码行，定向补齐。

- 怎么做:

- 运行 Check 获取行覆盖率；若未达标，围绕未覆盖行增补测试并回归；循环直至满足阈值。

- 产出：行覆盖率报告与补充测试。

- 通过标准：UnityChipCheckerTestCaseWithLineCoverage 达标（默认阈值 0.9，可在配置中调整）。

- 说明：该阶段在配置中标记 `skip=true`，可用 `--unskip` 指定索引启用。

- 检查器:

- UnityChipCheckerTestCaseWithLineCoverage

- \* 作用：在 TestCase 基础上统计行覆盖率并对比阈值。

- \* 参数：

- doc\_func\_check (str): 功能/检查点文档路径。示例：`{OUT}/allowbreak{}{DUT}_functions_and_checks.`  
`allowbreak{}md`

- doc\_bug\_analysis (str): 缺陷分析文档路径。示例：`{OUT}/allowbreak{}{DUT}_bug_analysis.`  
`allowbreak{}md`

- test\_dir (str): 测试目录根路径。示例：`{OUT}/tests`

- cfg (dict | Config): 必填，用于推导默认路径以及环境配置。

- min\_line\_coverage (float, 默认按配置，未配置则 0.8): 最低行覆盖率阈值。

- coverage\_json (str, 可选): 行覆盖率 JSON 路径。默认：`uc_test_report/allowbreak{}line_dat/allowbreak{}code_coverage.`

- `allowbreak{}json`

- coverage\_analysis (str, 可选): 行覆盖率分析 MD 输出。默认：`unity_test/allowbreak{}{DUT}_line_coverage_analysis.`  
`allowbreak{}md`

- coverage\_ignore (str, 可选): 忽略文件清单。默认：`unity_test/allowbreak{}tests/allowbreak{}{DUT}.`

- `allowbreak{}ignore`

## 阶段 11：验证审查与总结

- 目标：沉淀成果、复盘流程、给出改进建议。
- 怎么做：
  - 完善 /  
`allowbreak{}{DUT}_bug_analysis.`  
allowbreak{}md 的缺陷条目（基于源码分析）。
  - 汇总并撰写 /  
`allowbreak{}{DUT}_test_summary.`  
allowbreak{}md，回看规划是否达成；必要时用 GotoStage 回补。
- 产出：<OUT>/  
`allowbreak{}{DUT}_test_summary.`  
allowbreak{}md 与最终结论。
- 通过标准：UnityChipCheckerTestCase 复核通过。
- 检查器：
  - UnityChipCheckerTestCase
    - \* 作用：复核整体测试结果与文档一致性，形成最终结论。
    - \* 参数: doc\_func\_check: “{OUT}/{DUT}\_functions\_and\_checks.md”; doc\_bug\_analysis: “{OUT}/{DUT}\_bug\_analysis.md”; test\_dir: “{OUT}/tests”。

## 提示与最佳实践

- 随时用工具：Detail/Status 查看 Mission 进度与当前阶段；CurrentTips 获取步骤级指导；Check/Complete 推进阶段。
- TUI 左侧 Mission 会显示阶段序号、跳过状态与失败计数；可结合命令行 -  
`allowbreak{}-`  
`allowbreak{}skip/`  
`allowbreak{}-`  
`allowbreak{}-`  
`allowbreak{}unskip/`  
`allowbreak{}-`  
`allowbreak{}-`  
`allowbreak{}force-`  
`allowbreak{}stage-`  
`allowbreak{}index` 控制推进。

## 14.2 定制工作流（增删阶段/子阶段）

### 14.2.1 原理说明

- 工作流定义在语言配置 ucagent/  
allowbreak{}lang/  
allowbreak{}zh/  
allowbreak{}config/  
allowbreak{}default.  
allowbreak{}yaml 的顶层 stage: 列表。
- 配置加载顺序: setting.yaml → ~/.ucagent/setting.yaml → 语言默认 (含 stage) → 项目根 config.yaml → CLI --override。
- 注意: 列表类型 (如 stage 列表) 在合并时是“整体覆盖”, 不是元素级合并; 因此要“增删改”阶段, 建议把默认的 stage 列表复制到你的项目 config.yaml, 在此基础上编辑。
- 临时不执行某阶段: 优先使用 CLI --skip 跳过该索引; 持久跳过可在你的 config.yaml 中把该阶段条目的 skip: true 写上 (同样需要提供完整的 stage 列表)。

### 14.2.2 增加阶段

- 需求: 在“全面验证执行”之后新增一个“静态检查与 Lint 报告”阶段, 要求生成 <OUT>/allowbreak{}{DUT}\_lint\_report.  
allowbreak{}md 并做格式检查。
- 做法: 在项目根 config.yaml 中提供完整的 stage: 列表, 并在合适位置插入如下条目 (片段示例, 仅展示新增项, 实际需要放入你的完整 stage 列表里)。

```
stage:  
# ... 前面的既有阶段...  
- name: static_lint_and_style_check  
  desc: "静态分析与代码风格检查报告"  
  task:  
    - "目标: 完成 DUT 的静态检查/Lint, 并输出报告"  
    - "第 1 步: 运行 lint 工具 (按项目需要)"  
    - "第 2 步: 将结论整理为 <OUT>/ {DUT} _lint_report.md (中文)"  
    - "第 3 步: 用 Check 校验报告是否存在且格式规范"  
  checker:  
    - name: markdown_file_check  
      class: "UnityChipCheckerMarkdownFormat"  
      args:
```

```

    markdown_file_list: "{OUT}/{DUT}_lint_report.md" # MD 文件路
    ↵ 径或列表
        no_line_break: true # 禁止字面量 "\n" 作为换行
        reference_files: []
        output_files:
            - "{OUT}/{DUT}_lint_report.md"
        skip: false
    # ... 后续既有阶段...

```

### 14.2.3 减少子阶段

- 场景：在“功能规格分析与测试点定义”中，临时不执行“功能点定义（FC）”子阶段。
- 推荐做法：运行时使用 CLI `--skip` 跳过该索引；若需长期配置，复制默认 `stage:` 列表到你的 `config.yaml`，在父阶段 `functional_specification_analysis` 的 `stage:` 子列表里移除对应子阶段条目，或为该子阶段加 `skip: true`。

子阶段移除（片段示例，仅展示父阶段结构与其子阶段列表）：

```

stage:
    - name: functional_specification_analysis
        desc: " 功能规格分析与测试点定义"
        task:
            - " 目标：将芯片功能拆解成可测试的小块，为后续测试做准备"
            # ... 省略父阶段任务...
    stage:
        - name: functional_grouping # 保留 FG 子阶段
            # ... 原有配置...
            # - name: function_point_definition # 原来的 FC 子阶段（此行及其内
    ↵ 容整体删除，或在其中加 skip: true)
            - name: check_point_design # 保留 CK 子阶段
                # ... 原有配置...
            # ... 其他字段...

```

### 小贴士

- 仅需临时跳过：用 `--skip`/`--unskip` 最快，无需改配置文件。
- 需要永久增删：复制默认 `stage:` 列表到项目 `config.yaml`，编辑后提交到仓库；注意列表是整体覆盖，别只贴新增/删减的片段。

- 新增阶段的检查器可复用现有类（如 Markdown/Fixture/API/Coverage/TestCase 等），也可以扩展自定义检查器（放在 `ucagent/allowbreak{}checkers/allowbreak{}并以可导入路径填写到 class`）。

### 14.3 定制校验器 (checker)

#### 原理说明

- 每个（子）阶段下的 `checker:` 是一个列表；执行 `Check` 时会依次运行该列表里的所有检查器。
- 配置字段：
  - `name:` 该检查器在阶段内的标识（便于阅读/日志）
  - `class:` 检查器类名；短名默认从 `ucagent.checkers` 命名空间导入，也可写完整模块路径（如 `mypkg.allowbreak{}mychk.allowbreak{}MyChecker`）
  - `args:` 传给检查器构造函数的参数，支持模板变量（如 `{OUT}`、`{DUT}`）
  - `extra_args:` 可选，部分检查器支持自定义提示/策略（如 `fail_msg`、`batch_size`、`pre_report_file` 等）
- 解析与实例化：`ucagent/allowbreak{}stage/allowbreak{}vstage.allowbreak{}py` 会读取 `checker:`，按 `class/args` 生成实例；运行期由 `ToolStdCheck/allowbreak{}Check` 调用其 `do_check()`。
- 合并语义：配置合并时列表是“整体替换”，要在项目 `config.yaml` 修改某个阶段的 `checker:`，建议复制该阶段条目并完整替换其 `checker:` 列表。

#### 14.3.1 增加 checker

在“功能规格分析与测试点定义”父阶段，新增“文档格式检查”，确保 `{OUT}/allowbreak{}{DUT}_functions_and_checks.allowbreak{}md` 没有把换行写成字面量 `\n`。

```
# 片段示例：需要放入你的完整 stage 列表对应阶段中
- name: functional_specification_analysis
  desc: " 功能规格分析与测试点定义 "
  # ...existing fields...
  output_files:
```

```

    - "{OUT}/{DUT}_functions_and_checks.md"

checker:
    - name: functions_and_checks_doc_format
      clss: "UnityChipCheckerMarkdownFileFormat"
      args:
        markdown_file_list: "{OUT}/{DUT}_functions_and_checks.md" # 功
        ↳ 能/检查点文档
        no_line_break: true # 禁止字面量 "\n"

stage:
# ... 子阶段 FG/FC/CK 原有配置...

```

(可扩展) 自定义检查器 (最小实现, 放在 ucagent/

```

allowbreak{}checkers/
allowbreak{}unity_test.
allowbreak{}py)
```

很多场景下“增加的 checker”并非复用已有检查器, 而是需要自己实现一个新的检查器。最小实现步骤:

1. 新建类并继承基类 ucagent.

```

allowbreak{}checkers.
allowbreak{}base.
allowbreak{}Checker
```

2. 在 `__init__` 里声明你需要的参数 (与 YAML args 对应)

3. 实现 `do_check(self, timeout=0, **kw) -> tuple[bool, object]`, 返回 (是否通过, 结构化消息)

4. 如需读/写工作区文件, 使用 `self.get_path(rel)` 获取绝对路径; 如需跨阶段共享数据, 使用 `self.`

```

allowbreak{}smanager_set_value/
allowbreak{}get_value
```

5. 若想用短名 `clss` 引用, 请在 ucagent/

```

allowbreak{}checkers/
allowbreak{}__init__.
allowbreak{}py 导出该类 (或在 clss 写完整模块路径)
```

最小代码骨架 (示例):

```
# 文件: ucagent/checkers/unity_test.py
from typing import Tuple
```

```

import os
from ucagent.checkers.base import Checker

class UnityChipCheckerMyCustomCheck(Checker):
    def __init__(self, target_file: str, threshold: int = 1, **kw):
        self.target_file = target_file
        self.threshold = threshold

    def do_check(self, timeout=0, **kw) -> Tuple[bool, object]:
        """ 检查 target_file 是否存在并做简单规则校验。 """
        real = self.get_path(self.target_file)
        if not os.path.exists(real):
            return False, {"error": f"file '{self.target_file}' not found"}
        # TODO: 这里写你的具体校验逻辑，例如统计/解析/比对等
        return True, {"message": "MyCustomCheck passed"}

```

在阶段 YAML 中引用（与“增加一个 checker”一致）：

```

checker:
  - name: my_custom_check
    class: "UnityChipCheckerMyCustomCheck" # 若未在 __init__.py 导出，写完整
    ↳ 路径 mypkg.mychk.UnityChipCheckerMyCustomCheck
    args:
      target_file: "{OUT}/{DUT}_something.py"
      threshold: 2
    extra_args:
      fail_msg: " 未满足自定义阈值，请完善实现或调低阈值。" # 可选：通过
    ↳ extra_args 自定义默认失败提示

```

进阶提示（按需）：

- 长时任务/外部进程：在运行子进程时调用 `self.set_check_process(p, timeout)`，即可用工具 `KillCheck/allowbreak{}StdCheck` 管理与查看进程输出。
- 模板渲染：实现 `get_template_data()` 可将进度/统计渲染到阶段标题与任务文案中。
- 初始化钩子：实现 `on_init()` 以在阶段开始时加载缓存/准备批任务（与 Batch 系列 checker 一致）。

### 14.3.2 删除 checker

如临时不要“第 2 阶段基本信息文档格式检查”，将该阶段的 checker：置空或移除该项：

```
- name: dut_function_understanding
  desc: "{DUT} 功能理解"
  # ...existing fields...
  checker: [] # 删除原本的 markdown_file_check
```

### 14.3.3 修改 checker

把“行覆盖率检查”的阈值从 0.9 调整到 0.8，并自定义失败提示：

```
- name: line_coverage_analysis_and_improvement
  desc: "代码行覆盖率分析与提升 {COVERAGE_COMPLETE}"
  # ...existing fields...
  checker:
    - name: line_coverage_check
      class: "UnityChipCheckerTestCaseWithLineCoverage"
      args:
        doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"
        doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"
        test_dir: "{OUT}/tests"
        min_line_coverage: 0.8 # 调低阈值
      extra_args:
        fail_msg: "未达到 80% 的行覆盖率，请补充针对未覆盖行的测试。"
```

可选：自定义检查器类

- 在 ucagent/
 

```
allowbreak{}checkers/
allowbreak{} 新增类，继承 ucagent.
allowbreak{}checkers.
allowbreak{}base.
allowbreak{}Checker 并实现 do_check();
```
- 在 ucagent/
 

```
allowbreak{}checkers/
allowbreak{}__init__.
allowbreak{}py 导出类后，可在 class 用短名；或直接写完整模块路径；
```

- args 中的字符串支持模板变量渲染; extra\_args 可用于自定义提示文案 (具体视检查器实现而定)。

#### 14.3.4 常用 checker 参数 (结构化)

以下参数均来自实际代码实现 (`ucagent/allowbreak{}checkers/allowbreak{}unity_test.allowbreak{}py`)，名称、默认值与类型与代码保持一致；示例片段可直接放入阶段 YAML 的 `checker[] .args`。

##### 14.3.4.1 UnityChipCheckerMarkdownFileFormat

- 参数：
  - `markdown_file_list` (str | List[str]): 要检查的 Markdown 文件路径或路径列表。
  - `no_line_break` (bool, 默认 false): 是否禁止把换行写成字面量 \n; true 表示禁止。
- 示例：

```
args:  
  markdown_file_list: "{OUT}/{DUT}_basic_info.md"  
  no_line_break: true
```

##### 14.3.4.2 UnityChipCheckerLabelStructure

- 参数：
  - `doc_file` (str)
  - `leaf_node` ( “FG” | “FC” | “CK” )
  - `min_count` (int, 默认 1)
  - `must_have_prefix` (str, 默认 “FG-API” )
  - `data_key` (str, 可选)
- 示例：

```
args:  
  doc_file: "{OUT}/{DUT}_functions_and_checks.md"  
  leaf_node: "CK"  
  data_key: "COVER_GROUP_DOC_CK_LIST"
```

#### 14.3.4.3 UnityChipCheckerDutCreation

- 参数:

- `target_file` (str)

- 示例:

```
args:  
  target_file: "{OUT}/tests/{DUT}_api.py"
```

#### 14.3.4.4 UnityChipCheckerDutFixture

- 参数:

- `target_file` (str)

- 示例:

```
args:  
  target_file: "{OUT}/tests/{DUT}_api.py"
```

#### 14.3.4.5 UnityChipCheckerEnvFixture

- 参数:

- `target_file` (str)
  - `min_env` (int, 默认 1)
  - `force_bundle` (bool, 当前未使用)

- 示例:

```
args:  
  target_file: "{OUT}/tests/{DUT}_api.py"  
  min_env: 1
```

#### 14.3.4.6 UnityChipCheckerDutApi

- 参数:

- `api_prefix` (str)
- `target_file` (str)
- `min_apis` (int, 默认 1)

- 示例:

```
args:  
  api_prefix: "api_{DUT}_"  
  target_file: "{OUT}/tests/{DUT}_api.py"  
  min_apis: 1
```

#### 14.3.4.7 UnityChipCheckerCoverageGroup

- 参数:

- `test_dir` (str)
- `cov_file` (str)
- `doc_file` (str)
- `check_types` (str|List[str])

- 示例:

```
args:  
  test_dir: "{OUT}/tests"  
  cov_file: "{OUT}/tests/{DUT}_function_coverage_def.py"  
  doc_file: "{OUT}/{DUT}_functions_and_checks.md"  
  check_types: ["FG", "FC", "CK"]
```

#### 14.3.4.8 UnityChipCheckerCoverageGroupBatchImplementation

- 参数:

- `test_dir` (str)
- `cov_file` (str)
- `doc_file` (str)
- `batch_size` (int)
- `data_key` (str)

- 示例:

```
args:  
  test_dir: "{OUT}/tests"  
  cov_file: "{OUT}/tests/{DUT}_function_coverage_def.py"  
  doc_file: "{OUT}/{DUT}_functions_and_checks.md"  
  batch_size: 20  
  data_key: "COVER_GROUP_DOC_CK_LIST"
```

#### 14.3.4.9 UnityChipCheckerTestTemplate

- 基类参数: doc\_func\_check (str), test\_dir (str), min\_tests (int, 默认 1), timeout (int, 默认 15)
- 扩展参数 (extra\_args): batch\_size (默认 20), ignore\_ck\_prefix (str), data\_key (str)
- 示例:

```
args:  
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"  
  test_dir: "{OUT}/tests"  
  ignore_ck_prefix: "test_api_{DUT}_"  
  data_key: "TEST_TEMPLATE_IMP_REPORT"  
  batch_size: 20
```

#### 14.3.4.10 UnityChipCheckerDutApiTest

- 参数:
  - api\_prefix (str)
  - target\_file\_api (str)
  - target\_file\_tests (str)
  - doc\_func\_check (str)
  - doc\_bug\_analysis (str)
  - min\_tests (int, 默认 1)
  - timeout (int, 默认 15)

- 示例:

```
args:  
  api_prefix: "api_{DUT}_"
```

```
target_file_api: "{OUT}/tests/{DUT}_api.py"
target_file_tests: "{OUT}/tests/test_{DUT}_api*.py"
doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"
doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"
```

#### 14.3.4.11 UnityChipCheckerBatchTestsImplementation

- 基类参数: doc\_func\_check (str), test\_dir (str), doc\_bug\_analysis (str), ignore\_ck\_prefix (str), timeout (int, 默认 15)
- 进度参数: data\_key (str, 必填)
- 扩展参数 (extra\_args): batch\_size (默认 5), pre\_report\_file (str)
- 示例:

```
args:
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"
  doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"
  test_dir: "{OUT}/tests"
  ignore_ck_prefix: "test_api_{DUT}_"
  batch_size: 10
  data_key: "TEST_TEMPLATE_IMP_REPORT"
  pre_report_file: "{OUT}/{DUT}/.TEST_TEMPLATE_IMP_REPORT.json"
```

#### 14.3.4.12 UnityChipCheckerTestCase

- 参数:
  - doc\_func\_check (str)
  - doc\_bug\_analysis (str)
  - test\_dir (str)
  - min\_tests (int, 默认 1)
  - timeout (int, 默认 15)
- 示例:

```
args:
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"
  doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"
```

```
test_dir: "{OUT}/tests"
```

#### 14.3.4.13 UnityChipCheckerTestCaseWithLineCoverage

- 基础参数同 UnityChipCheckerTestCase
- 额外必需: cfg (dict|Config)
- 额外可选 (extra\_args):
  - min\_line\_coverage (float, 默认按配置, 未配置则 0.8)
  - coverage\_json (str, 默认 uc\_test\_report/  
allowbreak{}line\_dat/  
allowbreak{}code\_coverage.  
allowbreak{}json)
  - coverage\_analysis (str, 默认 unity\_test/  
allowbreak{}{DUT}\_line\_coverage\_analysis.  
allowbreak{}md)
  - coverage\_ignore (str, 默认 unity\_test/  
allowbreak{}tests/  
allowbreak{}{DUT}.  
allowbreak{}ignore)
- 示例:

```
args:  
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"  
  doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"  
  test_dir: "{OUT}/tests"  
  cfg: "<CONFIG_OBJECT_OR_DICT>"  
  min_line_coverage: 0.9
```

提示: 上面的示例仅展示 args 片段; 实际需置于阶段条目下的 checker[] .args。

## 15 规范生成

### 15.1 工作流介绍

规范生成是 UCAgent 自定义工作流的一个案例，专门用于从分散的设计资料（如源码、文档、注释等）中提取、整理和生成结构化的功能规范文档。通过修改 `config.yaml` 中的流程定义，达到了一个规范生成的效果。

如果有其他的自定义工作流的需求，可以参考[定制工作流（增删阶段/子阶段）](#)来自定义阶段从而满足实际的需要。

#### 15.1.1 应用场景

- **规范文档缺失:**芯片设计项目中常见源码完善但规范文档缺失或过时的情况
- **文档整合需求:**将分散在多处的设计说明、注释、会议记录等整合为统一规范
- **模块理解辅助:**通过自动生成的规范文档快速理解复杂设计模块
- **验证前准备:**在进行单元测试验证前，先生成完整的功能规范作为验证基准

#### 15.1.2 工作流程

本流程采用六阶段流水线式工作流，每个阶段专注于特定的文档生成任务：

收集现有资料 → 源码增强 → 完善子规范 → 人工检查 → 功能规范分析 → 功能行映射

**OUT 可自行配置，默认为 `unity_test`**

**15.1.2.1 1. collect\_existing\_assets (收集现有资料，搭建主 Spec)** 目标:扫描并收集项目中所有可用的设计文档和说明材料

- 检查并读取 README.md、设计文档、注释等
- 将现有资料整理为初始规范框架
- 输出:{OUT}/  
`allowbreak{}{DUT}_spec.`  
`allowbreak{}md` 初始版本

**Checker:** `MarkDownHeadChecker` - 验证生成文档的 Markdown 标题结构是否规范

### 15.1.2.2 2. augment\_with\_code (源码增强补全细节) 目标:逐个分析源码文件,从代码中提取设计意图并补充到规范文档

- 使用 WalkFilesOneByOne 策略逐文件分析
- 提取接口定义、状态机、关键逻辑等信息
- 将源码分析结果融入规范文档

**Checker:** WalkFilesOneByOne - 确保每个源文件都被遍历分析

### 15.1.2.3 3. complete\_subspecs (批量完善子规范) 目标:针对复杂模块,生成更细粒度的子模块规范文档

- 识别需要独立说明的子模块
- 在 {OUT}/{DUT}/ 目录下生成子规范文档
- 批量验证所有子规范的结构完整性

**Checker:** BatchMarkDownHeadChecker - 批量检查所有子规范文档的标题结构

### 15.1.2.4 4. human\_check (人工综合检查) 目标:暂停流程,由人工审核并修正规范文档

- 支持编辑 {DUT}\_spec.md 和子规范文档
- 人工补充 AI 可能遗漏的设计意图
- 修正 AI 理解偏差
- 最后在 TUI 中输入 hmcheck\_pass 继续流程

**Checker:** HumanChecker - 等待用户确认后继续

### 15.1.2.5 5. functional\_specification\_analysis (功能规范分析,提炼功能点与检测点) 目标:从规范文档中提取结构化的功能点标签

- 分析规范文档,生成 FG/FC/CK 功能标签体系
- 标注功能分组、功能点和检查点
- 输出:{OUT}/  
allowbreak{}unity\_test/  
allowbreak{}{DUT}\_functions\_and\_checks.  
allowbreak{}md

**Checker:** UnityChipCheckerLabelStructure - 验证功能标签的结构和命名规范

### 15.1.2.6 6. ref\_function\_line\_map\_generation (功能行映射生成, 参考检查点新检查点差异分析)

目标: 建立功能点与源码行的对应关系

- 分析源码, 将功能点映射到具体代码行
- 生成可追溯的功能-代码映射表
- 用于后续覆盖率分析和测试规划

Checker: FileLineMapChecker - 验证映射文件的差异和完整性

## 15.2 使用规范生成流程

### 15.2.1 前置条件

**workspace** 为工作目录, 可自行选择

#### 1. 准备设计模块

- 将待分析的 RTL 代码放入 `workspace/allowbreak{}{DUT}/allowbreak{}{}` 目录
- 将现有的各种文档放入 `{DUT}/docs/` 或 README 中
- 新生成的 spec 文档会输出到 `workspace/allowbreak{}{OUT}/allowbreak{}{}`

#### 2. 创建 GenSpec 配置

- 在模块目录下创建 `genspec.yaml` 配置文件
- 也可使用默认配置直接启动

### 15.2.2 快速开始

以下命令均在 `examples` 目录。执行为 `ucagent` 是已经安装为命令后。如果未安装命令, 则执行命令(项目内运行)应为:

```
python3 ucagent.py ...
```

以 Adder 模块为例:

```
# 1. 准备环境 (创建 output 目录和现有模块与文档)
mkdir output
```

```
# 2. 移动现有 RTL 和文档
cp -r examples/GenSpec/Adder output/

# 3. 启动 GenSpec 流程 (使用 MCP 集成模式)
ucagent output/ Adder --config examples/GenSpec/genspec.yaml -hm --tui
↪ --mcp-server-no-file-tools --no-embed-tools --guid-doc-path
↪ examples/GenSpec/SpecDoc/dut_spec_template.md

# 或者直接启动 TUI 模式
ucagent output/ Adder --config examples/GenSpec/genspec.yaml -hm --tui -s
↪ --no-embed-tools -l --guid-doc-path
↪ examples/GenSpec/SpecDoc/dut_spec_template.md
```

### 15.2.3 规范生成流程配置说明

规范生成流程使用独立的 `genspec.yaml` 配置文件自定义了一套规范生成流程，主要配置项包括：

```
# 任务使命描述
mission: |
    根据提供的 {DUT} 源码和现有文档，生成完整的功能规范文档

# 禁用默认模板 (GenSpec 不需要测试代码框架)
template: ""

# 写保护目录 (防止误修改源码)
un_write_dirs:
    - "{DUT}/"

# TUI 配置
tui:
    output_lines: 50
    messages_height: 20
    layout_ratio: [3, 2]

# 阶段定义
stages:
```

```
- name: collect_existing_assets
  label: 1-收集现有资料
  checker:
    - MarkdownHeadChecker:
        mode: check
        glob_or_file_path: "{OUT}/{DUT}_spec.md"
      # ... (其他配置)

- name: augment_with_code
  label: 2-源码增强
  checker:
    - WalkFilesOneByOne:
        dir: "{DUT}"
      # ... (其他配置)

# ... (其他阶段)
```

#### 15.2.4 关键配置项说明

- **mission**: 定义 GenSpec 的总体任务目标, 会影响各阶段的执行策略
- **template**: 设为空字符串, 因为 GenSpec 专注于文档生成而非测试代码
- **un\_write\_dirs**: 保护源码目录不被修改, 仅允许在输出目录生成文档
- **stages**: 定义六个阶段的执行顺序、Checker 和提示词模板

#### 15.2.5 GenSpec 输出结构

执行完成后, 输出目录 (`output/`) 结构如下:

```
output/
├── {DUT}_spec.md          # 主规范文档
├── {DUT}/
│   ├── *.v / *.sv / *.scala # RTL 源文件
│   └── ...
└── {DUT}/                  # 子规范目录(可选)
    ├── submodule1_spec.md   # 子模块规范
    └── submodule2_spec.md
└── unity_test/             # 功能标签输出
```

```
|── {DUT}_functions_and_checks.md      # FG/FC/CK 标签清单  
|── {DUT}_line_func_map.md            # 行与功能映射文档  
|── {DUT}_line_map_analysis.md        # 行映射分析报告  
|── {DUT}_spec_summary.md             # 文档检查摘要  
└── {DUT}_spec.md                   # 主规范文档
```

### 15.2.6 与 MCP Client 协作

GenSpec 支持通过 MCP 协议与外部 Code Agent 协作:

#### 1. 配置 MCP Client (以 Qwen Code 为例)

```
{  
  "mcpServers": {  
    "genspec": {  
      "httpUrl": "http://localhost:5000/mcp",  
      "timeout": 10000  
    }  
  }  
}
```

#### 2. 启动 GenSpec Server

```
ucagent output/ Adder --config examples/GenSpec/genspec.yaml -hm --tui  
↳ --mcp-server-no-file-tools --no-embed-tools --guid-doc-path  
↳ examples/GenSpec/SpecDoc/dut_spec_template.md
```

#### 3. 在 MCP Client 中启动协作

在 `output` 文件夹启动 code agent, 然后输入提示词:

> 请通过工具 `RoleInfo` 获取你的角色信息和基本指导, 然后完成任务。使用工具 `ReadTextFile` 读取文

#### 4. 流程监控

在 UCAgent TUI 界面中可以实时查看:

- 当前执行阶段
- 工具调用情况
- 文档生成进度

### 15.3 高级用法

#### 15.3.1 自定义阶段提示词

在 `genspec.yaml` 中可以为每个阶段自定义提示词模板:

```
stages:
  - name: collect_existing_assets
    doc: |
      ** 阶段目标 **: 收集 {DUT} 的所有现有设计文档

      ** 执行步骤 **:
      1. 读取 {DUT}/README.md
      2. 扫描 {DUT}/ 目录下的所有文档文件
      3. 整理为初始规范框架

      ** 输出要求 **:
      - 文件路径: {OUT}/{DUT}_spec.md
      - 格式: Markdown, 包含以下章节:
          - # 概述
          - # 功能描述
          - # 接口说明
          - # 设计细节
```

#### 15.3.2 跳过特定阶段

使用环境变量控制阶段跳过:

```
stages:
  - name: human_check
    skip: ${SKIP_HUMAN_CHECK: false}
```

启动时设置环境变量:

```
SKIP_HUMAN_CHECK=true make spec_Adder
```

## 15.4 常见问题

### 15.4.1 Q1: 生成的规范不完整怎么办?

A: 规范生成流程设计了 `human_check` 阶段专门用于人工补充:

1. 在 `human_check` 阶段暂停时, 手动编辑 `output/allowbreak{}{DUT}_spec.allowbreak{}md`
2. 补充 AI 遗漏的关键信息 (如特殊约束、边界条件)
3. 在 TUI 中输入 `hmcheck_pass` 继续流程

### 15.4.2 Q2: 如何提高规范生成质量?

A: 建议提供更丰富的输入材料:

- 在源码中添加详细注释
- 提供完善的 README.md 说明模块功能
- 如有设计文档 (Word/PDF), 转换为 Markdown 放入模块目录
- 在 `genspec.yaml` 的 `mission` 中明确特殊要求

### 15.4.3 Q3: 支持哪些 HDL 语言?

A: 规范生成流程支持常见的硬件描述语言:

- Verilog (.v)
- SystemVerilog (.sv)
- Chisel/Scala (.scala)
- VHDL (.vhdl, .vhd)

源码分析依赖于 LLM 的代码理解能力, 对语法无严格限制。

### 15.4.4 Q4: 规范生成流程和默认的验证生成流程有什么区别?

A:

都是在 UCAgent 这个大框架下的工作流。只是一个用于文档生成, 一个用于验证生成。可以通过修改 `config.yaml` 自行转换或者同时使用。

| 特性   | 规范生成             | 验证生成                |
|------|------------------|---------------------|
| 目标   | 生成功能规范文档         | 生成并执行测试用例           |
| 输出   | Markdown 规范文档    | Python 测试代码 + 覆盖率报告 |
| 模板   | 无 (template: "") | unity_test 模板       |
| 阶段数  | 6 (文档生成流水线)      | 14+ (验证完整流程)        |
| 适用场景 | 规范缺失/文档整理        | 单元测试验证              |

典型工作流: 先用规范生成流程生成规范 → 再用验证生成流程进行验证

#### 15.4.5 Q5: 如何复用规范生成流程配置?

A:

1. 通用配置模板: examples/  
`allowbreak{}GenSpec/`  
`allowbreak{}genspec.`  
`allowbreak{}yaml` 可作为模板复用
2. 项目级配置: 将 `genspec.yaml` 放在项目根目录, 所有模块共享
3. 模块级定制: 在 examples/  
`allowbreak{}{DUT}/`  
`allowbreak{}genspec.`  
`allowbreak{}yaml` 中覆盖特定配置

优先级: 模块级 > 项目级 > 默认配置

#### 15.4.6 Q6: 规范生成流程能否增量更新规范?

A: 当前规范生成流程采用全量生成模式。增量更新建议流程:

1. 保存旧版本规范: `cp output/{DUT}_spec.md output/{DUT}_spec.md.backup`
2. 重新运行规范生成流程生成新规范
3. 使用 `diff` 或 Copilot Chat 比较变更:  
`diff output/{DUT}_spec.md.backup output/{DUT}_spec.md`
4. 手动合并保留的内容

## 16 多 UCAgent 并发执行

### 16.1 背景

在实际验证工作中，经常需要同时对多个 DUT 进行并发验证以提升效率。UCAgent 支持在同一节点上运行多个实例，实现并发验证。本文档介绍如何在不同模式下实现多 UCAgent 并发执行。

### 16.2 实现方式

UCAgent 支持两种并发执行方式：

1. **API 模式并发**: 基于独立工作区的多实例运行
2. **MCP 模式并发**: 基于端口隔离的 MCP Server + Code Agent 协作

### 16.3 API 模式并发

#### 16.3.1 原理说明

在 API 模式下，不同 workspace 下的 UCAgent 实例是完全独立的，因此可以在多个终端窗口中同时运行，互不干扰。

#### 16.3.2 基本用法

**16.3.2.1 准备工作** 在运行 UCAgent 之前，需要先准备好 DUT 文件和工作目录。如果你克隆了 UCAgent 仓库，Makefile 会自动完成这些初始化工作 (`init_Adder` 目标)：

1. 创建 RTL 目录并复制设计文件 (.v/.sv/.scala 等)
2. 使用 picker 工具生成波形配置文件
3. 复制相关文档和环境文件

如果你是通过 pip 安装的 ucagent，需要手动准备工作目录：

```
# 创建工作目录
mkdir -p output/Adder output/Adder_RTL

# 复制你的 RTL 设计文件到 RTL 目录
cp /path/to/your/Adder.v output/Adder_RTL/
```

```
# (可选) 如果有 filelist.txt
cp /path/to/your/filelist.txt output/Adder_RTL/

# 使用 picker 生成 DUT 配置 (如果安装了 picker)
picker export output/Adder_RTL/Adder.v --rw 1 --sname Adder --tdir output/ -c
↪ -w output/Adder/Adder.fst

# 复制其他必要文件 (README、环境脚本等)
cp /path/to/your/*.py output/Adder/ 2>/dev/null || true
```

在不同工作区的 config.yaml 中配置好模型、API\_KEY 和 API\_BASE 之后，就可以启动 UCAGent 了。注： config.yaml 文件可从 UCAGent 官方仓库根目录获取。

### 16.3.2.2 多终端窗口方式

准备好工作目录后，最简单的方式是在不同终端窗口中分别启动 UCAGent：

```
# 终端窗口 1
cd workspace_A
ucagent output/ Adder --config config.yaml -s -hm --tui -l

# 终端窗口 2
cd workspace_B
ucagent output/ Mux --config config.yaml -s -hm --tui -l
```

### 16.3.3 使用 Tmux 管理多实例

为了更方便地管理多个并发实例，推荐使用 tmux 工具（Debian / Ubuntu 可直接 apt 安装）。tmux 允许在单个终端中创建和管理多个会话窗口。

```
# 进工作目录
cd your_workspace

# 配置环境变量 (模型、API_KEY 和 API_BASE)，下面是 iflow 的例子
export OPENAI_MODEL=glm-4.6
export OPENAI_API_KEY=<your_key>
```

```
export OPENAI_API_BASE=https://apis.iflow.cn/v1

# 清理旧输出
rm -rf output

# 初始化工作目录 A (Adder)
mkdir -p output/A output/A_RTL
cp /path/to/examples/Adder/*.v output/A_RTL/ 2>/dev/null
cp /path/to/examples/Adder/*.sv output/A_RTL/ 2>/dev/null
cp /path/to/examples/Adder/filelist.txt output/A_RTL/ 2>/dev/null
# 导出 python 包
picker export output/A_RTL/Adder.v --rw 1 --sname Adder --tdir output/ -c -w
↪ output/A/Adder.fst 2>/dev/null
cp /path/to/examples/Adder/*.md output/A/ 2>/dev/null
cp /path/to/examples/Adder/*.py output/A/ 2>/dev/null

# 初始化工作目录 B (Mux)
mkdir -p output/B output/B_RTL
cp /path/to/examples/Mux/*.v output/B_RTL/ 2>/dev/null
cp /path/to/examples/Mux/*.sv output/B_RTL/ 2>/dev/null
cp /path/to/examples/Mux/filelist.txt output/B_RTL/ 2>/dev/null
picker export output/B_RTL/Mux.v --rw 1 --sname Mux --tdir output/ -c -w
↪ output/B/Mux.fst 2>/dev/null
cp /path/to/examples/Mux/*.md output/B/ 2>/dev/null
cp /path/to/examples/Mux/*.py output/B/ 2>/dev/null

# 杀掉可能存在的旧会话
tmux kill-session -t my_multi_api_session 2>/dev/null

# 创建新的 tmux 会话
tmux new-session -d -s my_multi_api_session

# 在第一个窗格启动 Adder 验证（如果使用了虚拟环境，需要在 tmux 窗口里手动激活）
tmux send-keys -t my_multi_api_session:0.0 "ucagent output/A/ Adder --config
↪ config.yaml -s -hm --tui -l --no-embed-tools" C-m

# 水平分割窗口
tmux split-window -h -t my_multi_api_session:0.0
```

```
# 在第二个窗格启动 Mux 验证（延迟 10 秒避免同时启动冲突，如果使用了虚拟环境，需
→ 要在 tmux 窗口里手动激活）
tmux send-keys -t my_multi_api_session:0.1 "sleep 10 && ucagent output/B/ Mux
→ --config config.yaml -s -hm --tui -l --no-embed-tools" C-m

# 附加到 tmux 会话查看
tmux attach-session -t my_multi_api_session
```

**16.3.3.1 完整流程（包含初始化）** !!! info “关于 Makefile 自动化” 如果你克隆了 UCAgent 仓库，Makefile 中的 test\_Adder 目标会自动完成上述初始化步骤（通过 init\_Adder 依赖）：

- 创建 output/allowbreak{}Adder\_RTL 目录并复制 RTL 文件
- 使用 picker 生成波形配置文件
- 复制相关文档和环境文件

在 examples/allowbreak{}MultiRun/allowbreak{}Makefile 中提供了完整的自动化实现：

```
cd examples/MultiRun
make api_mul # 自动初始化并启动 tmux 多实例
```

等效于手动执行： make test\_Adder ARGS='--no-embed-tools' CWD=output/A

tmux 常用快捷键：

- Ctrl+b %: 水平分割窗格
- Ctrl+b -: 垂直分割窗格
- Ctrl+b 方向键: 在窗格间切换
- Ctrl+b d: 分离会话（后台运行）
- Ctrl+b x: 关闭当前窗格

## 16.4 MCP 模式并发

### 16.4.1 原理说明

MCP 模式下通过端口隔离实现并发。核心思路是：

1. 为每个 UCAGENT 实例分配一个独立的可用端口
2. 使用该端口启动 UCAGENT MCP Server
3. 配置 Code Agent 连接到对应端口
4. 多个 “UCAGENT + Code Agent” 组合可以并发运行

#### 16.4.2 端口分配策略

为了避免端口冲突，需要自动获取可用端口。在 `examples/allowbreak{}MultiRun/allowbreak{}Makefile` 中提供了自动获取可用端口的实现：

```
APORT != bash -c '\
for try in $($seq 1 100); do \
    cand=$$(shuf -i 2000-65000 -n 1); \
    if ! ss -ltn | awk '\''{print $$4}\'\' | grep -Eq "[.]:$${cand}$$"; then \
        echo $$cand; \
        exit 0; \
    fi; \
done; \
exit 1'
```

这段代码会：

1. 在 2000-65000 范围内随机选择一个端口号
2. 使用 `ss -ltn` 检查端口是否被占用
3. 返回第一个可用端口
4. 最多尝试 100 次

#### 16.4.3 基本使用流程

##### 16.4.3.1 1. 启动 UCAGENT MCP Server 使用 `--mcp-server-port` 参数指定端口：

```
# 使用端口 5001
ucagent output/Adder_5001/ Adder -s -hm --tui \
--mcp-server-port 5001 \
--mcp-server-no-file-tools \
--no-embed-tools
```

参数说明：

```
· --mcp-server-port 5001: 指定 MCP Server 监听端口为 5001  
· -  
allowbreak{}-  
allowbreak{}mcp-  
allowbreak{}server-  
allowbreak{}no-  
allowbreak{}file-  
allowbreak{}tools: 不暴露文件操作工具（安全考虑）  
· --no-embed-tools: 不启用嵌入式检索工具（节省资源）
```

#### 16.4.3.2 2. 配置 Code Agent 以 iFlow CLI 为例，需要修改 MCP Server 配置指向正确的端口。

创建工作区专属配置：

```
# 创建工作区的 iFlow 配置目录  
mkdir -p output/Adder_5001/.iflow  
  
# 复制默认配置  
cp ~/.iflow/settings.json output/Adder_5001/.iflow/settings.json  
  
# 修改端口配置  
sed -i 's/5000\`/mcp/5001\`/mcp/' output/Adder_5001/.iflow/settings.json
```

配置文件示例 (output/  
allowbreak{}Adder\_5001/  
allowbreak{}.  
allowbreak{}iflow/  
allowbreak{}settings.  
allowbreak{}json):

```
{  
    "mcpServers": {  
        "unitytest": {  
            "httpUrl": "http://localhost:5001/mcp",  
            "timeout": 10000  
        }  
    }  
}
```

#### 16.4.3.3 3. 启动 Code Agent 在对应工作目录启动 Code Agent:

```
# 进入工作目录  
cd output/Adder_5001  
  
# 启动 iFlow CLI (会自动读取当前目录的 .iflow/settings.json)  
npx -y @iflow-ai/iflow-cli@latest -y
```

#### 16.4.3.4 4. 开始验证 在 Code Agent 中输入提示词开始验证:

请通过工具 `RoleInfo` 获取你的角色信息和基本指导，然后完成任务。使用工具 `ReadTextFile` 读取文

#### 16.4.4 使用 Tmux 管理 MCP 并发

推荐使用 tmux 同时管理 UCAgent Server 和 Code Agent:

```
# 创建 tmux 会话  
tmux new-session -d -s mcp_5001  
  
# 在第一个窗格启动 UCAgent MCP Server  
tmux send-keys -t mcp_5001:0.0 "ucagent output/Adder_5001/ Adder  
↪ --mcp-server-port 5001 -hm --tui --mcp-server-no-file-tools  
↪ --no-embed-tools" C-m  
  
# 分割窗口  
tmux split-window -h -t mcp_5001:0.0  
  
# 在第二个窗格启动 Code Agent  
tmux send-keys -t mcp_5001:0.1 "cd output/Adder_5001 && npx -y  
↪ @iflow-ai/iflow-cli@latest -y" C-m  
  
# 附加到会话  
tmux attach-session -t mcp_5001
```

### 16.4.5 Makefile 自动化示例

在 examples/  
allowbreak{}MultiRun/  
allowbreak{}Makefile 中提供了完整的自动化实现：

```
mcp_mul: clean
    # 获取可用端口
    @echo "Selected MCP port: $(APORT)"
    tmux kill-session -t my_multi_mcp_session_$(APORT) || true

    # 在 tmux 中启动 MCP Server
    tmux new-session -d -s my_multi_mcp_session_$(APORT)
    tmux send-keys -t my_multi_mcp_session_$(APORT):0.0 "$(mcp_cmd)"
    ↳ ARGS='--mcp-server-port=$(APORT)' CWD=$(mcp_cwd) C-m

    # 配置 iFlow 并启动
    mkdir -p ../../$(mcp_cwd)/.iflow
    cp $(ifw_cfg) ../../$(mcp_cwd)/.iflow/settings.json
    sed -i "s/$(ifw_port)\/mcp\/$(APORT)\/mcp/"
    ↳ ../../$(mcp_cwd)/.iflow/settings.json

    tmux split-window -h -t my_multi_mcp_session_$(APORT):0.0
    tmux send-keys -t my_multi_mcp_session_$(APORT):0.1 "cd
    ../../$(mcp_cwd);$(bash_iflow_wait)" C-m
    tmux attach-session -t my_multi_mcp_session_$(APORT)
```

使用方法：

```
# 进入 examples/MultiRun 目录
cd examples/MultiRun

# 配置 API Key (编辑 IFLOW_env.bash)
vim IFLOW_env.bash

# 执行 MCP 模式并发验证
make mcp_mul
```

每次执行 make mcp\_mul 会：

1. 自动分配一个可用端口
2. 创建独立的工作目录（如 `output/allowbreak{}Adder_5001`）
3. 启动 UCAgent MCP Server
4. 配置并启动 iFlow CLI
5. 使用 tmux 进行会话管理

可以在多个终端中运行多次 `make mcp_mul`，每次会获得不同的端口和独立的工作环境。

## 16.5 配置说明

### 16.5.1 环境变量配置文件

在 `examples/allowbreak{}MultiRun/allowbreak{}IFLOW_env.allowbreak{}bash` 中配置 API 相关信息：

```
export OPENAI_MODEL=glm-4.6
export OPENAI_API_KEY=<your_key>
export OPENAI_API_BASE=https://apis.iflow.cn/v1
```

使用前需要替换 `<your_key>` 为实际的 API Key。

### 16.5.2 Makefile 变量说明

```
# MCP Server 启动命令
mcp_cmd = make -C ../../ mcp_Adder

# 工作目录（包含端口号以区分）
mcp_cwd = output/Adder_${APORT}

# iFlow 配置文件路径
ifw_cfg = ~/.iflow/settings.json

# iFlow 默认端口（用于替换）
ifw_prt = 5000
```

可根据实际项目需求修改这些变量。

## 16.6 其他 Code Agent 支持

除了 iFlow CLI，其他支持 MCP 的 Code Agent 也可以使用相同的方式：

### 16.6.1 Qwen Code

配置文件路径： /

```
allowbreak{}.  
allowbreak{}qwen/  
allowbreak{}settings.  
allowbreak{}json
```

```
{  
    "mcpServers": {  
        "unitytest": {  
            "httpUrl": "http://localhost:5001/mcp",  
            "timeout": 10000  
        }  
    }  
}
```

启动方式：

```
cd output/Adder_5001  
qwen
```

### 16.6.2 Claude Code

参考：[Claude Code MCP 配置文档](#)

### 16.6.3 Gemini CLI

参考：[Gemini CLI MCP 配置文档](#)

### 16.6.4 VS Code Copilot

参考: [VS Code MCP 扩展配置](#)

## 16.7 故障排查

### 16.7.1 端口被占用

现象: 启动 UCAgent MCP Server 时提示端口已被占用

解决方法:

```
# 查找占用端口的进程
lsof -i :5001

# 或使用 ss 命令
ss -tlnp | grep 5001

# 终止占用端口的进程
kill -9 <PID>
```

### 16.7.2 Code Agent 连接失败

现象: Code Agent 无法连接到 MCP Server

排查步骤:

1. 确认 UCAgent MCP Server 已正常启动

```
curl http://localhost:5001/mcp
```

2. 检查配置文件中的端口号是否正确

```
cat output/Adder_5001/.iflow/settings.json
```

3. 检查防火墙设置

```
sudo ufw status
```

### 16.7.3 tmux 会话丢失

**现象：**退出终端后 tmux 会话消失

**说明：**tmux 会话在系统重启或用户登出后可能丢失。

**解决方法：**

1. 使用 `systemd` 或 `screen` 等工具保持会话
2. 使用 tmux 插件 `tmux-resurrect` 保存和恢复会话
3. 编写启动脚本并添加到系统服务

### 16.7.4 内存不足

**现象：**系统响应变慢，出现 OOM (Out of Memory) 错误

**解决方法：**

1. 减少并发实例数量
2. 使用 `--no-embed-tools` 禁用嵌入式工具节省内存
3. 为系统配置 swap 空间
4. 使用 `ulimit` 限制单个进程内存使用

## 16.8 相关文档

- [API 直接使用模式](#)
- [MCP 集成模式](#)
- [批处理执行](#)
- [TUI 使用指南](#)

# 17 批处理执行模式

## 17.1 概述

批处理模式 (Batch Run) 允许 UCAgent 在无人值守的情况下自动完成一系列验证任务。

## 17.2 核心特性

### 17.2.1 1. 自动退出机制

UCAgent 正常运行时，完成任务后会保持运行状态以便查看结果。批处理模式通过 -

```
allowbreak{}-
allowbreak{}exit-
allowbreak{}on-
allowbreak{}completion
```

参数实现任务完成后自动退出，便于串联执行多个任务。

### 17.2.2 2. 自动继续机制

在 MCP 模式下，通过 Code Agent 的 Hooks 功能，可以在 LLM 停止时自动发送“继续”指令，实现任务的自动推进。

### 17.2.3 3. 任务编排

支持通过 Makefile 或脚本编排多个验证任务的执行顺序，实现复杂的批处理流程。

## 17.3 使用方式

### 17.3.1 API 模式批处理

API 模式下的批处理最为简单，通过 -

```
allowbreak{}-
allowbreak{}exit-
allowbreak{}on-
allowbreak{}completion
```

参数让 UCAgent 在完成任务后自动退出。

```
# 单个 DUT 的批处理验证
ucagent output/ Adder \
-s -l \
--exit-on-completion \
--no-embed-tools \
--config config.yaml
```

### 17.3.1.1 基本用法 参数说明:

- `-s, --stream-output`: 实时输出日志
- `-l, --loop`: 启动后立即进入循环执行
- `--exit-on-completion, -eoc`: 任务完成后自动退出
- `--no-embed-tools`: 禁用嵌入式检索工具（节省资源）

### 17.3.1.2 串联多个任务 使用 shell 脚本串联执行:

```
#!/bin/bash
# batch_verify.sh

DUTS=("Adder" "Mux" "FSM" "ALU")
FAILED=()

for dut in "${DUTS[@]}"; do
    echo "====="
    echo " 开始验证: $dut"
    echo "====="

    if ucagent output/ $dut -s -l --exit-on-completion --no-embed-tools; then
        echo "✓ $dut 验证成功"
    else
        echo "✗ $dut 验证失败"
        FAILED+=("$dut")
    fi

    # 保存结果到独立目录
    if [ -d "output" ]; then
        mkdir -p results/$dut
        cp -r output/* results/$dut/
        rm -rf output
    fi
done

# 输出汇总
echo ""
echo "====="
echo " 批处理验证完成"
```

```
echo "====="
echo " 成功: $((#${DUTS[@]} - ${#FAILED[@]}))/${#DUTS[@]}%"
if [ ${#FAILED[@]} -gt 0 ]; then
    echo " 失败的 DUT: ${FAILED[*]}"
    exit 1
fi
```

### 17.3.1.3 Makefile 方式 在 examples/

allowbreak{}BatchRun/  
allowbreak{}Makefile 中提供了完整的批处理示例：

```
APORT=5001
OPORT=5000

clean:
    rm output -rf

init: clean
    mkdir -p output

api_batch: init
    make -C ../../ clean
    make -C ../../ test_Adder ARGS="-eoc --no-embed-tools"
    cp -r ../../output output/Adder
    make -C ../../ clean
    make -C ../../ test_Mux ARGS="-eoc --no-embed-tools"
    cp -r ../../output output/Mux
    make -C ../../ clean
    make -C ../../ test_FSM ARGS="-eoc --no-embed-tools"
    cp -r ../../output output/FSM
```

使用方法：

```
cd examples/BatchRun
make api_batch
```

这个命令会依次验证 Adder、Mux、FSM 三个模块，并将结果保存到 `output/` 目录的对应子目录中。

### 17.3.2 MCP 模式批处理

MCP 模式下需要配合 Code Agent 的 Hooks 功能实现自动继续。

#### 17.3.2.1 工作原理

1. **UCAgent MCP Server**: 以 -

```
allowbreak{}-
allowbreak{}exit-
allowbreak{}on-
allowbreak{}completion 参数启动
```

2. **Code Agent Hooks**: 在 LLM 停止时触发自动继续

3. **自动推进**: 通过 tmux 发送命令实现无人值守

#### 17.3.2.2 Hook 消息机制

UCAgent 提供了 `--hook-message` 参数，用于获取配置中定义的提示词：

```
# 查看继续提示词
ucagent --hook-message continue

# 查看完成提示词
ucagent --hook-message quit

# 从配置文件读取 (格式: [config.yaml::]key[|stop_key])
ucagent --hook-message config.yaml::continue_key|complete_key
```

工作机制：

- 检查 UCAgent 状态（通过 `.ucagent_info.json`）
- 如果任务未完成，返回 `continue_key` 对应的提示词
- 如果任务已完成，返回 `stop_key` 对应的提示词（如果提供）
- 通过退出码 0 表示成功，非 0 表示失败

#### 17.3.2.3 iFlow CLI 配置

在 /

```
allowbreak{}.
allowbreak{}iflow/
allowbreak{}settings.
allowbreak{}json 中配置 Hooks:
```

```
{  
  "mcpServers": {  
    "unitytest": {  
      "httpUrl": "http://localhost:5001/mcp",  
      "timeout": 10000  
    }  
  },  
  "hooks": {  
    "Stop": [  
      {  
        "hooks": [  
          {  
            "type": "command",  
            "command": "M=`ucagent --hook-message 'continue|quit'`  
            ↳ && (tmux send-keys $M; sleep 1; tmux send-keys  
            ↳ Enter)",  
            "timeout": 3  
          }  
        ]  
      }  
    ]  
  }  
}
```

配置说明：

- **Stop 触发器**: 在 LLM 每次停止工作时触发
- **command 执行**:
  - `ucagent --hook-message 'continue|quit'`: 获取提示词
  - `tmux send-keys $M`: 将提示词发送到当前 tmux 窗口
  - `sleep 1; tmux send-keys Enter`: 按下回车键
- **timeout**: 命令执行超时时间 (秒)

!!! tip “提示词配置” `continue` 和 `quit` 可以是：

- 配置文件中的键名 (如 `config.yaml::continue\_prompt`)
- 环境变量名 (如 `\$CONTINUE\_MSG`)
- 直接的提示词文本

### 17.3.2.4 单次 MCP 批处理 单个 DUT 的 MCP 批处理流程:

```
#!/bin/bash
# mcp_batch_one.sh

DUT=$1
PORT=${2:-5001}

# 1. 启动 UCAgent MCP Server
tmux new-session -d -s ucagent_${DUT}
tmux send-keys -t ucagent_${DUT} "ucagent output/ ${DUT} --mcp-server-port $PORT
↪ -eoc --tui --no-embed-tools" C-m

# 2. 等待 Server 就绪
while [ ! -f "output/Guide_Doc/dut_fixture.md" ]; do
    sleep 5
done

# 3. 配置 iFlow
mkdir -p output/.iflow
cp ~/.iflow/settings.json output/.iflow/settings.json
sed -i "s/5000\mcp/$PORT\mcp/" output/.iflow/settings.json

# 4. 启动 iFlow CLI (自动继续)
cd output
npx -y @iflow-ai/iflow-cli@latest -y

# 5. 等待完成
while [ ! -f "exited.txt" ]; do
    sleep 5
done

echo "✓ $DUT 验证完成"
```

### 17.3.2.5 批量 MCP 执行 在 examples/ allowbreak{}BatchRun/ allowbreak{}Makefile 中提供了完整的批处理实现:

```
mcp_one%:
    @echo "Selected MCP port: $(APORT)"
    make -C ../../ clean
    make -C ../../ mcp_* ARGS="-eoc --mcp-server-port=$(APORT)"
    cp -r ../../output output/$*
    @while [ ! -f "../../output/exited.txt" ]; do \
        sleep 5; \
    done
    make -C ../../ clean

mcp_batch: init
    @$(MAKE) mcp_one_Adder
    @$(MAKE) mcp_one_Mux
    @$(MAKE) mcp_one_FSM

iflow_once:
    @while [ -f "../../output/.ucagent_info.json" ]; do \
        sleep 5; \
    done
    @while [ ! -f "../../output/Guide_Doc/dut_fixture.md" ]; do \
        sleep 5; \
    done
    mkdir -p ../../output/.iflow
    cp ~/.iflow/settings.json ../../output/.iflow/settings.json
    sed -i "s/$(OPORT)/mcp/$(APORT)/mcp/" ../../output/.iflow/settings.json
    (sleep 10; tmux send-keys `ucagent --hook-message cagent_init`; sleep 1;
    ↵ tmux send-keys Enter)&
    cd ../../output && npx -y @iflow-ai/iflow-cli@latest -y && (echo true >
    ↵ exited.txt)
    sleep 30

iflow_batch:
    @$(MAKE) iflow_once # Adder
    @$(MAKE) iflow_once # Mux
    @$(MAKE) iflow_once # FSM

iflow_batch_auto_tmux:
    tmux kill-session -t my_batch_iflow_session_$(APORT) || true
```

```
tmux new-session -d -s my_batch_iflow_session_${APORT}
tmux send-keys -t my_batch_iflow_session_${APORT}:0.0 "make mcp_batch" C-m
tmux split-window -h -t my_batch_iflow_session_${APORT}:0.0
tmux send-keys -t my_batch_iflow_session_${APORT}:0.1 "make iflow_batch" C-m
tmux attach-session -t my_batch_iflow_session_${APORT}
```

使用方法：

```
cd examples/BatchRun

# MCP 模式批处理 (需要先登录 iFlow)
make iflow_batch_auto_tmux
```

工作流程：

1. 左侧窗格：依次启动 UCAgent MCP Server (Adder → Mux → FSM)
2. 右侧窗格：依次启动 iFlow CLI 并自动继续
3. 同步机制：通过 `exited.txt` 标记文件确保顺序执行
4. 结果收集：每个 DUT 的结果保存到独立目录

## 17.4 其他 Code Agent 支持

### 17.4.1 Claude Code

Claude Code 也支持 Hooks 功能，配置方式类似：

参考文档：[Claude Code Hooks Guide](#)

配置示例：

```
{
  "hooks": {
    "onStop": {
      "command": "ucagent --hook-message continue",
      "type": "shell"
    }
  }
}
```

### 17.4.2 Gemini CLI

Gemini CLI 同样支持 Hooks:

参考文档: [Gemini CLI Configuration - Hooks](#)

配置示例:

```
hooks:  
  stop:  
    - type: command  
      command: "M=$(ucagent --hook-message continue) && echo $M"
```

### 17.4.3 Qwen Code

Qwen Code 的 Hooks 配置:

```
{  
  "hooks": {  
    "onModelStop": [  
      {  
        "type": "command",  
        "command": "ucagent --hook-message continue"  
      }  
    ]  
  }  
}
```

## 17.5 配置提示词

### 17.5.1 默认提示词

UCAgent 内置了默认的继续提示词, 可以通过 `--hook-message` 查看:

```
# 查看默认的继续提示词  
ucagent --hook-message continue  
# 输出: 继续完成任务
```

```
# 查看默认的退出提示词
ucagent --hook-message quit
# 输出: 任务已完成, 可以退出
```

### 17.5.2 自定义提示词

在项目的 config.yaml 中自定义提示词：

```
# 批处理相关提示词
batch_prompts:
  continue_key: |
    请继续执行验证任务。如果遇到问题，先尝试解决，然后继续。

  complete_key: |
    所有验证任务已完成，可以退出了。感谢使用 UCAgent！

  init_key: |
    请通过工具 RoleInfo 获取你的角色信息和基本指导，然后完成任务。
    使用工具 ReadTextFile 读取文件。你需要在当前工作目录进行文件操作，不要超出该
    目录。
```

使用自定义提示词：

```
# 从配置文件读取
ucagent --hook-message config.yaml::batch_prompts.continue_key

# 指定继续和退出的键
ucagent --hook-message "batch_prompts.continue_key|batch_prompts.complete_key"
```

### 17.5.3 环境变量方式

也可以通过环境变量定义提示词：

```
export CONTINUE_MSG="继续执行下一个验证任务"
export COMPLETE_MSG="批处理验证全部完成"

# 使用环境变量
```

```
ucagent --hook-message "\$CONTINUE_MSG|\$COMPLETE_MSG"
```

## 17.6 故障排查

### 17.6.1 问题 1：任务未自动退出

现象：使用 -

```
allowbreak{}-  
allowbreak{}exit-  
allowbreak{}on-  
allowbreak{}completion
```

后，UCAgent 仍未退出

可能原因：

1. Exit 工具未被成功调用
2. 任务未真正完成
3. 存在后台线程阻止退出

解决方法：

```
# 查看 UCAgent 状态  
cat output/.ucagent_info.json  
  
# 检查是否有残留进程  
ps aux | grep ucagent  
  
# 强制清理  
pkill -9 -f ucagent
```

### 17.6.2 问题 2：Hook 未触发

现象：配置了 Hooks 但自动继续功能不工作

排查步骤：

1. 检查 Code Agent 配置文件语法

```
# iFlow  
cat ~/.iflow/settings.json | jq .
```

## 2. 验证 Hook 命令

```
# 手动测试  
M=$(ucagent --hook-message continue) && echo $M
```

## 3. 查看 Code Agent 日志

```
# iFlow 日志通常在  
cat ~/.iflow/logs/latest.log
```

### 17.6.3 问题 3：批处理中断

现象：批处理执行到一半停止

可能原因：

1. 某个任务耗时过长
2. 系统资源耗尽
3. 网络问题 (API 调用失败)

解决方法：

```
# 添加超时和重试  
timeout 7200 ucagent output/ DUT -s -l -eoc --no-embed-tools || {  
    echo "任务超时或失败，等待 30 秒后重试..."  
    sleep 30  
    ucagent output/ DUT -s -l -eoc --no-embed-tools  
}
```

### 17.6.4 问题 4：结果文件冲突

现象：多个任务的结果相互覆盖

解决方法：为每个任务使用独立的工作目录

```
#!/bin/bash  
for dut in Adder Mux FSM; do  
    # 创建独立工作目录
```

```

WORK_DIR="batch_work_${dut}$$"
mkdir -p $WORK_DIR

# 在独立目录中执行
(cd $WORK_DIR && uagent ../output/ $dut -s -l -eoc --no-embed-tools)

# 收集结果
mkdir -p results/$dut
cp -r $WORK_DIR/output/* results/$dut/
rm -rf $WORK_DIR
done

```

## 17.7 示例项目

完整的批处理示例位于 examples/

allowbreak{}BatchRun/

allowbreak{} 目录:

```

examples/BatchRun/
├── Makefile          # 批处理任务定义
├── README.md         # 说明文档
└── (生成的输出)
    └── output/
        ├── Adder/      # Adder 验证结果
        ├── Mux/        # Mux 验证结果
        └── FSM/        # FSM 验证结果

```

快速开始:

```

# 进入示例目录
cd examples/BatchRun

# API 模式批处理
make api_batch

# MCP 模式批处理 (需要 iFlow 认证)
make iflow_batch_auto_tmux

```

## 17.8 相关文档

- 多实例并发执行 - 同时运行多个 UCAgent 实例
- API 直接使用模式 - UCAgent 基础使用方法
- MCP 集成模式 - 与 Code Agent 集成
- TUI 使用指南 - 终端界面使用