
Kea

Release 2.0.3

梁锡贤、沈祥臣、马搏

Dec 24, 2024

目录

| | |
|--|-----------|
| 1 简介 | 1 |
| 1.1 Kea 的优势 | 1 |
| 1.2 工作原理 | 2 |
| 2 用户手册 | 5 |
| 2.1 环境配置 | 5 |
| 2.1.1 通过 DevEco Studio 配置鸿蒙环境 | 5 |
| 配置 HarmonyOS 环境 | 5 |
| 2.1.2 通过 Android Studio 配置安卓环境 | 8 |
| 使用安卓模拟器配置安卓环境 | 8 |
| 2.1.3 使用命令行配置安卓环境 | 10 |
| 在 Linux 上设置 Android SDK 环境 | 10 |
| 在 Windows 上设置 Android SDK 环境 | 13 |
| 在 MAC OS 上设置 Android SDK 环境 | 14 |
| 创建并运行模拟器 | 16 |
| 2.2 编写第一个性质 | 16 |
| 2.2.1 检查环境 | 16 |
| 2.2.2 安装 | 16 |
| 2.2.3 编写第一个性质（安卓） | 17 |
| 2.2.4 启动 kea 并检查你的性质 | 20 |
| 2.3 为 HarmonyOS 编写性质 | 20 |
| 2.3.1 HarmonyOS 的 UI 控件查看工具: uiviewer 教程 | 20 |
| 2.3.2 HarmonyOS PDL API | 20 |
| 2.3.3 为 HarmonyOS 启动 Kea | 21 |
| 2.4 应用性质描述语言接口 | 21 |
| 2.4.1 用户界面交互事件 | 21 |
| 2.4.2 选择器 | 22 |
| 2.4.3 样例 | 23 |
| 2.5 性质定义教程 | 23 |
| 2.5.1 从应用错误报告中获取应用性质 | 23 |
| 2.5.2 从指定应用程序功能中提取性质 | 27 |
| 2.6 Kea 的参数选项 | 27 |
| 2.6.1 Kea 的参数 | 27 |
| 2.6.2 YAML 配置 | 28 |
| 2.6.3 当运行多个性质时 kea 做了什么? | 29 |
| 2.7 带状态的测试 | 30 |
| 2.8 缺陷报告阅读指南 | 34 |
| 3 设计手册 | 37 |

| | |
|---------------------------------|----|
| 3.1 架构图 | 37 |
| 3.2 目录结构 | 37 |
| 3.3 各功能部分设计文档 | 38 |
| 3.3.1 命令行交互功能 | 38 |
| 功能说明与功能设计 | 38 |
| 命令行参数解析 | 40 |
| YAML 配置文件 | 41 |
| 参数对象 | 41 |
| 启动 Kea | 41 |
| 主要函数设计 | 41 |
| 3.3.2 KeaTest | 43 |
| 功能设计与实现 | 44 |
| 3.3.3 性质装饰器 | 44 |
| 功能说明与功能设计 | 44 |
| 性质的定义 | 44 |
| 初始化函数的定义 | 47 |
| 主路径函数的定义 | 47 |
| 3.3.4 PDL 驱动 | 48 |
| PDL 驱动的功能设计 | 48 |
| 3.3.5 KeaTestElements | 51 |
| 功能设计与实现 | 51 |
| 3.3.6 Kea | 54 |
| 功能设计与实现 | 54 |
| Kea 类中的数据结构实现 | 55 |
| Kea 类中的成员方法实现 | 55 |
| 3.3.7 DroidBot | 59 |
| Droidbot 项目架构 | 59 |
| 3.3.8 HMDroidbot | 60 |
| HMDroidbot 项目架构 | 61 |
| AppHM | 61 |
| DeviceHM | 64 |
| HDC | 67 |
| InputEvent | 69 |
| UTG | 70 |
| DeviceState | 71 |
| 3.3.9 InputManager | 72 |
| 功能设计与实现 | 72 |
| 类属性 | 73 |
| InputManager 类中的数据结构 | 73 |
| InputManager 类中的成员方法 | 74 |
| 使用方法 | 76 |
| 3.3.10 RandomPolicy | 76 |
| 随机事件生成策略的介绍 | 76 |
| 随机事件生成策略的伪代码 | 78 |
| RandomPolicy 类中的数据结构 | 79 |
| RandomPolicy 类中的成员方法 | 79 |
| 3.3.11 GuidedPolicy | 81 |
| 主路径引导策略的介绍 | 81 |
| 默认参数 | 83 |
| GuidedPolicy 类中的数据结构 | 84 |
| 主路径引导策略的伪代码 | 84 |

| | |
|--------------------------------|------------|
| GuidedPolicy 类中的成员方法 | 86 |
| 3.3.12LLMPolicy | 90 |
| LLM 辅助事件生成策略的介绍 | 90 |
| LLM 辅助事件生成策略的伪代码 | 92 |
| LLMPolicy 类中的数据结构 | 93 |
| LLMPolicy 类中的成员方法 | 94 |
| 3.3.13带状态的测试 | 97 |
| 功能说明与功能设计 | 98 |
| 单例模式方法 | 99 |
| 数据状态增删改查的成员方法 | 99 |
| 随机生成状态文本的成员方法 | 100 |
| 随机获取一个状态的成员方法 | 101 |
| 4 实验与结果 | 103 |
| 4.1 Kea 工具应用效果简介 | 103 |
| 4.2 静态代码分析 | 104 |
| 4.3 动态代码测试 | 104 |

简介

保障移动应用质量是移动操作系统生态建设的关键。现有业界普遍采用的移动应用测试与分析技术（如人工/脚本测试、静态分析技术、界面测试技术）存在人力成本高、检错能力弱、功能场景无感知的局限性，很难用于自动化检测移动应用的功能测试中。因此，如何实现移动应用的自动化功能测试一直是一个具有挑战性的问题。

基于性质测试理论（Property-Based Testing, PBT）于 2000 年在函数式编程领域提出。该理论方法以被测系统应满足的性质为测试断言，通过自动生成大量随机输入数据以验证这些性质是否在各种情况下保持正确。与传统测试相比，基于性质的测试能够高效有效地覆盖被测系统输入空间及其边界情况，从而发现深层次的功能缺陷。

Kea 是首个基于性质测试理论设计开发的移动应用自动化功能测试工具，目前支持鸿蒙（OpenHarmony/HarmonyOS）和安卓（Android）应用软件的自动化功能测试。Kea 设计了：(1) 一种面向移动应用的性质描述语言（可支持用户编写以前置条件、交互场景、后置条件为主要形式的应用功能性质），(2) 三种页面探索策略：随机遍历、基于主路径遍历、大模型引导的路径遍历（自动生成事件序列来达到应用更深层的状态，有效覆盖移动应用事件探索空间）。

1.1 Kea 的优势

在传统的应用测试中，我们一般会进行静态分析，动态遍历测试，脚本测试。我们可以从 输入空间、功能相关性两个维度评估这些测试方法对 功能性缺陷的查错能力。

- 静态分析的输入空间小，仅对源代码利用静态分析的算法进行分析。功能相关性低，分析难以贴近真实应用功能。
- 动态遍历测试（Fuzzing）的输入空间大，可以生成大量不同的输入，从而到达不同的应用状态。然而其功能相关性低，一个应用功能可由一条多个事件组合而成的路径表示，随机测试的过程难以完整走出一个完整的应用功能路径。
- 脚本测试的输入空间小，一个脚本测试由初始化，应用功能执行脚本及断言组成，始终为一个单一的路径。功能相关性高，每个脚本都定义了一个功能。

上述三种传统测试方法的能力可以通过以下的坐标图显示，此前没有一种测试方法能同时达成高输入空间和高功能相关性。因此，我们提出将基于性质的测试方法应用于移动应用的功能测试领域，此方法能同时达成高功能相关性和高输入空间，从而对应用的功能正确性进行充分验证。

这里列举了 Kea 的一些优势：

1. 基于性质的测试：Kea 引入了一种通用且实用的测试技术，基于性质的测试（PBT），能够有效验证应用的功能性。

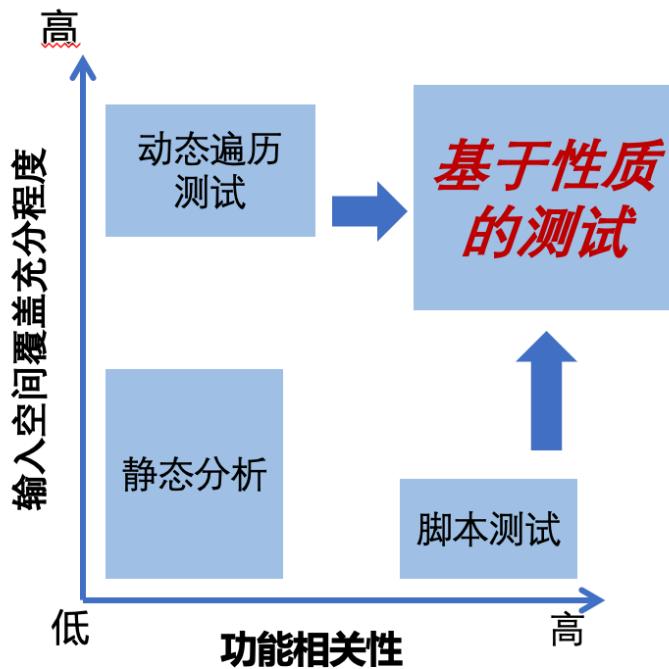


Fig. 1: Kea 与传统测试方法的能力对比示意图

2. 低维护成本：与传统的脚本测试相比，应用性质的维护成本较低，减少了测试人员的工作负担。
3. 强大的功能缺陷检测能力：Kea 在检测移动应用中的功能性缺陷方面非常强大，能够自动探索并验证应用的性质，快速发现潜在问题。

1.2 工作原理

Kea 的核心流程如上图，以下是一些执行流程的细节：

1. 步骤 1：分析被测移动应用功能特性和测试目标，使用软件功能性质描述语言定义被测移动应用的功能性质集合，每个功能性质包括前置条件 P 、交互场景 I 和后置条件 Q ；
2. 将所有定义的功能性质集合的并集存储下来，设置最大测试执行时长、最大执行事件个数，选择探索策略，若选择随机探索策略，则执行步骤 3；若选择主路径引导策略，则根据前置条件 P ，定义一个用户事件序列作为主路径，继续执行步骤 4；若选择大语言模型引导探索策略，则执行步骤 5；
3. 使用随机探索策略生成事件序列，执行性质检测，记录对应的测试结果和用户界面截图，直到达到最大测试执行时间，执行步骤 6；*RandomPolicy*
4. 使用主路径引导策略生成事件序列，执行性质检测，记录对应的测试结果和用户界面截图，直到达到最大测试执行时间，执行步骤 6；*GuidedPolicy*
5. 使用大语言模型引导策略生成事件序列，执行性质检测，记录对应的测试结果和用户界面截图，直到达到最大测试执行时间，执行步骤 6；*LLMPolicy*
6. 基于测试结果和用户界面截图，自动生成被测移动应用的缺陷报告；

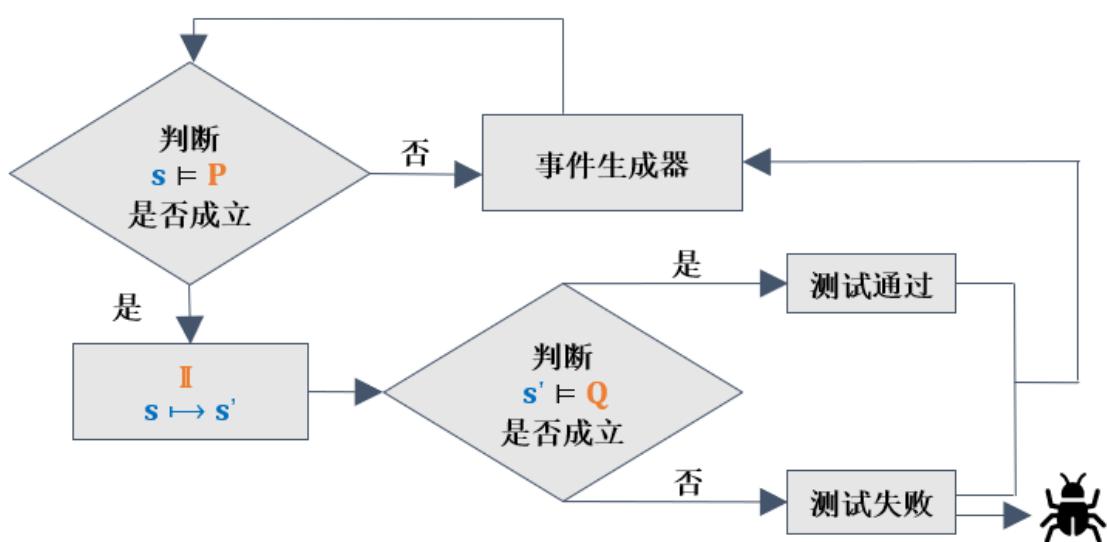


Fig. 2: kea 核心流程

用户手册

2.1 环境配置

跟随下列指示配置鸿蒙或安卓环境。

Tip: 环境配置步骤

1. 确保 hdc 或 adb 命令可用

Kea 依赖 hdc 与鸿蒙设备进行交互。使用 adb 命令与安卓设备进行交互。需要将 hdc 和 adb 工具路径添加到 PATH 环境变量中让其生效。

2. 连接设备或在电脑上启动模拟器
-

2.1.1 通过 DevEco Studio 配置鸿蒙环境

配置 HarmonyOS 环境

Tip: 本节目标

1. 让 `hdc` 命令可用

Kea 依赖 hdc 命令与鸿蒙设备进行交互。关键是将 hdc 添加到 PATH 环境变量中。

2. 连接真机或使用模拟器
-

1. 安装 DevEco Studio

下载并安装 DevEco Studio: [下载 DevEco Studio.](#)

2. 安装并配置 HarmonyOS SDK

打开 DevEco Studio, 安装 HarmonyOS SDK *DevEco Studio -> preferences -> OpenHarmony SDK.*

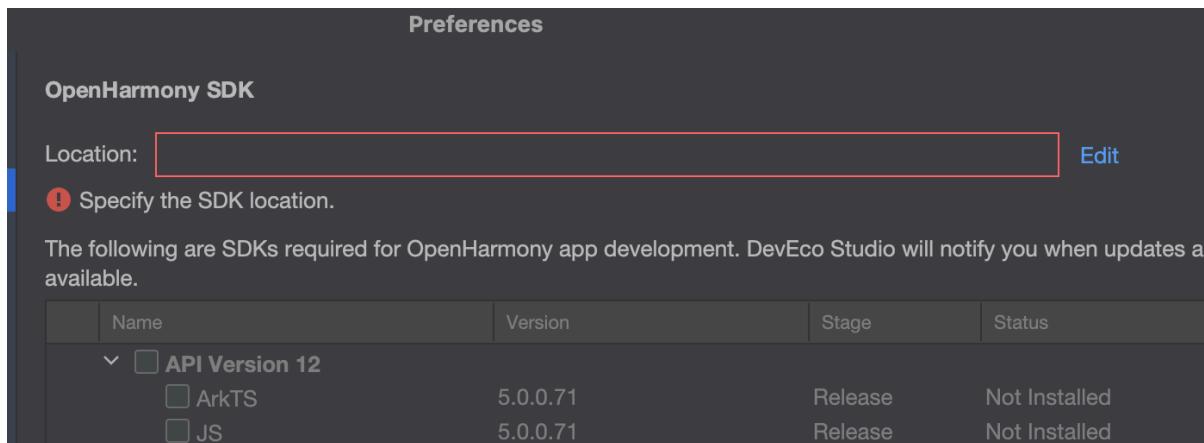


Fig. 1: 在 DevEco Studio 中设置 SDK

点击 edit。设置你的 SDK 路径并安装 OpenHarmony SDK 工具链。API 版本应为 12+ (5.0+)。

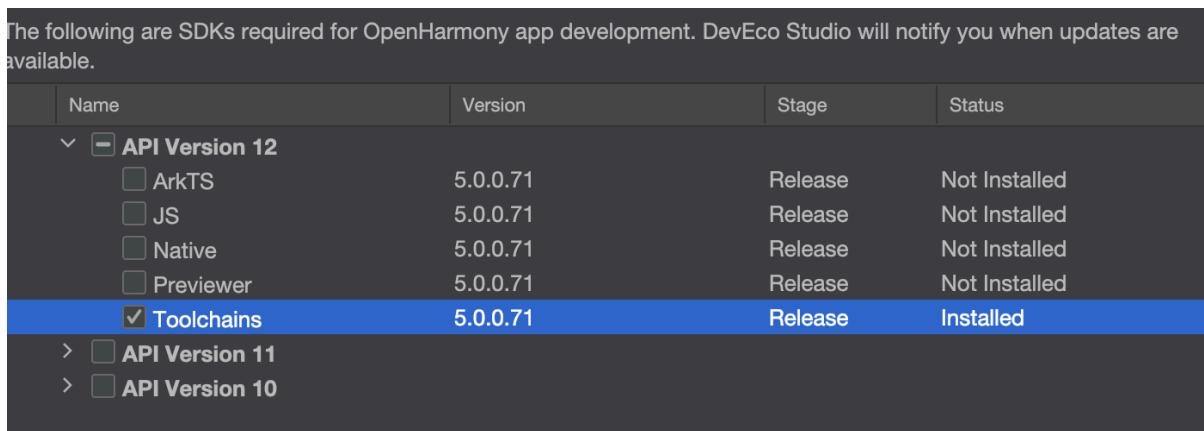


Fig. 2: 下载 toolchains (API 12+)

添加 sdk toolchains 到环境变量

MacOS 和 Linux

```
# macOS and Linux
export HARMONY_SDK_HOME=<Your path to opensdk home>
export PATH="$HARMONY_SDK_HOME/12/toolchains"
```

source shell 的配置文件以激活修改。

Windows 系统

在你的电脑上安装了 DevEco Studio。你需要做的是将其工具目录添加至 PATH 变量，以便令 SDK 命令在终端中生效。

如果你使用的是 zsh 或 bash，请使用 EXPORT 命令设置 HARMONY_SDK_HOME 环境变量。HARMONY_SDK_HOME 环境变量应指向你的 SDK 安装路径。默认路径是 C:\Users\usr_name\AppData\Local\OpenHarmony\Sdk。你可以通过 File -> Settings -> OpenHarmony SDK -> Edit 查看你的安装路径。

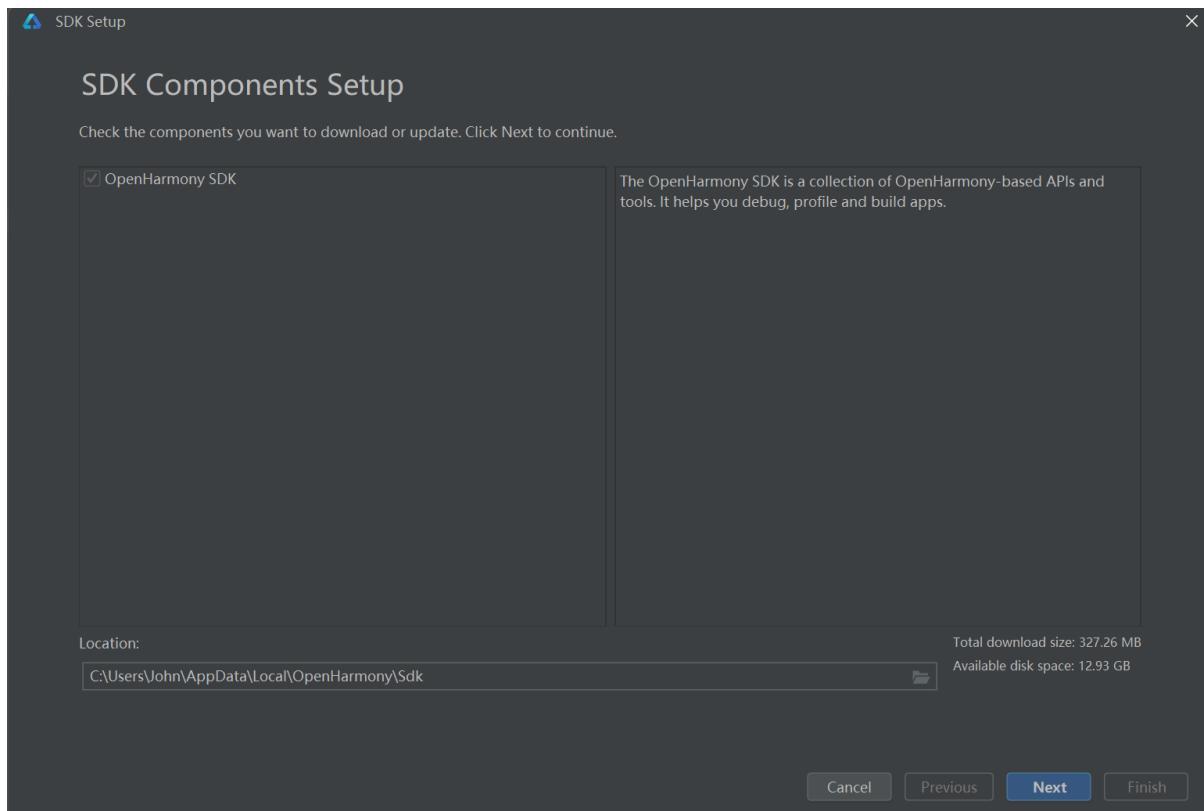


Fig. 3: DevEco Studio 中 HarmonyOS SDK 路径的示例

然后，将以下路径添加到 PATH 变量中。请参见 Windows 系统中如何添加 PATH 环境变量。

```
# Windows
HARMONY_SDK_HOME: "<Your path to opensdk home>"
PATH: %HARMONY_SDK_HOME%\12\toolchains
```

Important: 在终端中运行 hdc，查看命令是否可用。

3. 运行一个模拟器

根据此指南运行一个模拟器：管理及运行模拟器

Important: 在终端中运行 `hdc list targets`。你可以看到模拟器以一个环回地址套接字（`127.0.0.1:port`）的形式提供。

2.1.2 通过 Android Studio 配置安卓环境

使用安卓模拟器配置安卓环境

Important: 前提条件：在你的电脑上安装 Android Studio。

你可以在 <https://developer.android.com/studio> 下载 Android Studio。

Hint: Kea 依赖 adb 命令与 Android 设备进行交互。关键是要确保 ADB (Android 调试桥) 命令可用。以下教程将帮助你设置整个 Android SDK 工具套件。但请记住，将 adb 命令添加到路径中就足够了。

1. 设置命令行工具 (adb)

MacOS 和 Linux

在你的电脑上安装了 Android Studio。你需要做的是将 Android sdk 其添加至 PATH 环境变量，让其在终端中生效。

有关详细信息，请参见 Android Studio 文档：环境变量。

如果你使用的是 zsh 或 bash，请使用 EXPORT 命令设置 ANDROID_HOME 环境变量。ANDROID_HOME 环境变量应指向你的 SDK 安装路径。默认路径是 `/usr/Library/Android/sdk/`。你可以通过 *Android Studio* -> *设置* -> *语言与框架* -> *Android SDK* 查看你的安装路径。

在此窗口中，查看 *SDK* 工具，安装 **Android SDK** 平台工具。

然后，将路径添加到 `.bashrc` 或 `.zshrc` 文件中。

```
export ANDROID_HOME="/usr/.../Library/Android/sdk/"
# export 安卓安装目录中的所有命令。
export PATH="$ANDROID_HOME/emulator:$ANDROID_HOME/tools:$ANDROID_HOME/
↪cmdline-tools/latest/bin:$ANDROID_HOME/tools/bin:$ANDROID_HOME/cmdline-
↪tools/latest:$ANDROID_HOME/platform-tools:$PATH"
```

source shell 的配置文件以激活修改。

Important: text

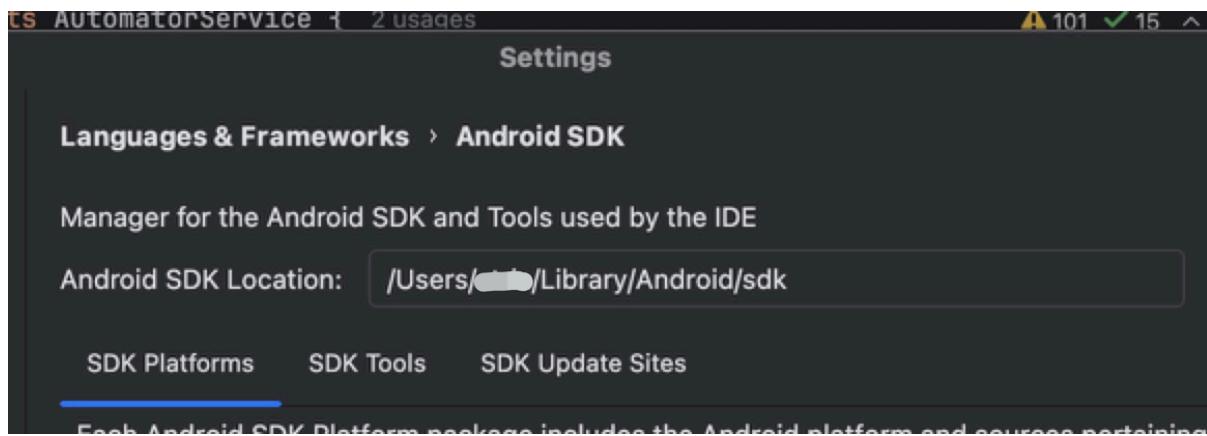


Fig. 4: Android Studio 中 Android SDK 路径的示例

在终端中输入 adb 检查设置是否成功。

Windows

在你的电脑上安装了 Android Studio。你需要做的是将 Android sdk 其添加至 PATH 环境变量，让其在终端中生效。

如果你使用的是 zsh 或 bash，请使用 EXPORT 命令设置 ANDROID_HOME 环境变量。ANDROID_HOME 环境变量应指向你的 SDK 安装路径。默认路径是 C:\Users\usr_name\AppData\Local\Android\Sdk。你可以通过文件 -> 设置 -> 语言与框架 -> Android SDK 查看你的安装路径。

在此窗口中，查看 SDK 工具，安装 **Android SDK** 平台工具。

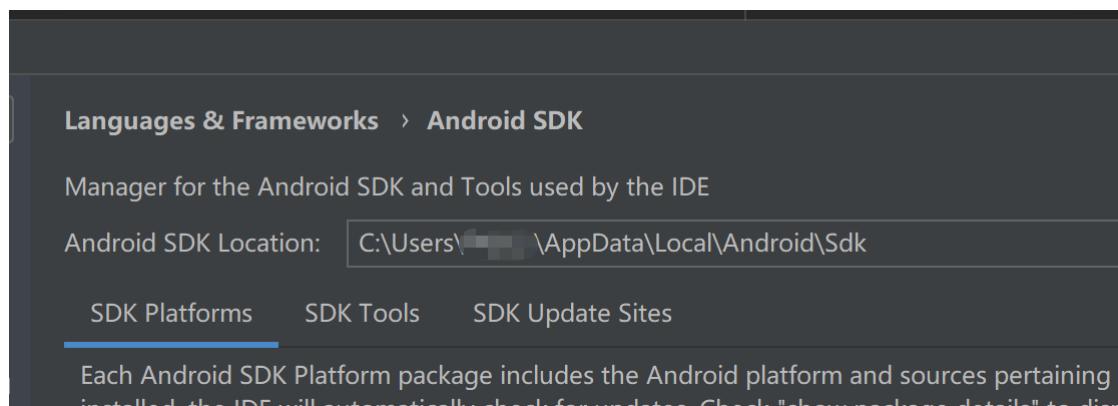


Fig. 5: Android Studio 中 Android SDK 路径的示例

然后，将以下路径添加到 PATH 变量中。请参见 Windows 系统中如何添加 PATH 环境变量。

```
ANDROID_HOME:  
C:\Users\usr_name\AppData\Local\Android\Sdk  
  
PATH:  
%ANDROID_HOME%\platform-tools
```

2.1.3 使用命令行配置安卓环境

步骤 1：在主机上安装并配置 **Android SDK** 环境

在 *Linux* 上设置 *Android SDK* 环境

在 *Windows* 上设置 *Android SDK* 环境

在 *MAC OS* 上设置 *Android SDK* 环境

步骤 2：创建并运行模拟器

创建并运行模拟器

在 *Linux* 上设置 *Android SDK* 环境

1. 安装 *Android* 命令行工具。前往 [Android Developer](#) 并下载与你的操作系统匹配的工具。

仅限命令行工具

| 平台 | SDK 工具包 | 尺寸 | SHA-256 校验和 |
|---------|--|----------|--|
| Windows | commandlinetools-win-11076708_latest.zip | 153.6 MB | 4d6931209eabb1fb7c7e8b240a6a3cb3ab24479ea294f3539429574b1eec862 |
| Mac | commandlinetools-mac-11076708_latest.zip | 153.6 MB | 7bc5c72ba0275c80a8f19684fb92793b83a6b5c94d4d179fc5988930282d7e64 |
| Linux | commandlinetools-linux-11076708_latest.zip | 153.6 MB | 2d2d50857e4eb553af5a6dc3ad507a17adf43d115264b1afc116f95c92e5e258 |

Android Studio 中包含了命令行工具。如果您不需要 Android Studio，可以下载上面的基本 *Android* 命令行工具。您可以使用随附的 [sdkmanager](#) 下载其他 SDK 软件包。

你可以在命令行中使用 `wget` 下载工具下载压缩包，也可以从浏览器中下载并自行解压。

你可以从上述 [Android Developer](#) 网站复制下载链接。然后使用以下命令。

```
 wget https://dl.google.com/android/repository/commandlinetools-linux-11076708_latest.zip?hl=zh-cn
 mkdir -p Android/cmdline-tools
 unzip commandlinetools-linux-11076708_latest.zip?hl=zh-cn -d Android/cmdline-tools
 mv Android/cmdline-tools/cmdline-tools Android/cmdline-tools/latest
```

2. 配置相关环境。

安装 Java。（如果你之前已经安装并配置过 Java，你需要检查你的 JDK 是否与命令行工具适配。如果版本适配，你可以跳过以下关于 Java 环境配置的步骤）

我们使用 `JDK-17` 来运行安卓模拟器。。

```
sudo apt install openjdk-17-jdk
```

打开你的 `.bashrc` 文件。

```
sudo nano ~/.bashrc
```

在文件末尾添加以下内容。

```
export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin
export ANDROID_HOME=$HOME/Android
export PATH="$ANDROID_HOME/emulator:$ANDROID_HOME/tools:$ANDROID_HOME/
↪cmdline-tools/latest/bin:$ANDROID_HOME/tools/bin:$ANDROID_HOME/
↪cmdline-tools/latest:$ANDROID_HOME/platform-tools:$PATH"
```

确保 `PATH` 配置与你存储相关工具的路径匹配。

最后，重新装载 `.bashrc` 文件，使更改立即应用于当前终端会话。

```
source ~/.bashrc
```

3. 验证 `sdkmanager` 是否成功安装。.. code-block:: bash

```
sdkmanager -update sdkmanager -list sdkmanager -licenses
```

如果你获得如下信息，则安装成功。

```
[=====] 100% Computing updates...
All SDK package licenses accepted.
```

你可以从 `sdkmanager` 的常用命令 了解更多。

WSL 常见问题

1. WSL 依赖项

请将你的 Windows 升级到 Win11，并使用 WSL 2。这将解决大多数 Win10 和 WSL 1 中的 WSL 问题。

2. WSL PATH 设置

默认情况下，WSL 将共享 Windows 系统中的环境变量。有时这会导致错误的行为。你可能会发现你使用的实际可执行文件并不是你通过 `which` 命令找到的文件。此问题的根本原因是：WSL 中的 `which` 命令只能找到 WSL 的 `PATH` 中的可执行文件。但是，如果你在 Windows `PATH` 和 WSL `PATH` 中同时配置了一个可执行文件（例如 `python3`），而 Windows `PATH` 在 WSL `PATH` 之前设置（`PATH=$Windows_PATH:$WSL_PATH`）。那么你使用的实际

可执行文件就是 Windows_PATH 中的文件。但你通过 which 找到的则是 WSL_PATH 中的文件。

要解决此问题，你可以按照以下建议操作：

- 在设置 PATH 时将你的环境 PATH 放在前面

使用 PATH>New_PATH:\$PATH 而不是 PATH=\$PATH>New_PATH。这是一个良好的习惯，可以优先考虑你的最新设置并确保其始终有效。

- 禁用环境变量的共享

```
# WSL bash
sudo vim /etc/wsl.conf

# 添加以下内容
[interop]
appendWindowsPath = false

# 在 PowerShell 中重启 WSL
wsl --shutdown
```

3. CPU 硬件加速问题

```
Error: This user doesn't have permissions to use KVM (/dev/kvm), ERROR: x86 emulation currently requires hardware acceleration!
```

```
ProbeKVM: This user doesn't have permissions to use KVM (/dev/kvm).
The KVM line in /etc/group is: [kvm:x:108:yiheng,josephMez]

If the current user has KVM permissions,
the KVM line in /etc/group should end with ":" followed by your username.

If we see LINE_NOT_FOUND, the kvm group may need to be created along with permissions:
sudo groupadd -r kvm
# Then ensure /lib/udev/rules.d/50-udev-default.rules contains something like:
# KERNEL=="kvm", GROUP="kvm", MODE="0660"
# and then run:
sudo gpasswd -a $USER kvm

If we see kvm:... but no username at the end, running the following command may allow KVM access:
sudo gpasswd -a $USER kvm

You may need to log out and back in for changes to take effect.

handleCpuAcceleration: feature check for hvf
emulator: ERROR: x86 emulation currently requires hardware acceleration!
CPU acceleration status: This user doesn't have permissions to use KVM (/dev/kvm).
The KVM line in /etc/group is: [kvm:x:108:yiheng,josephMez]

If the current user has KVM permissions,
the KVM line in /etc/group should end with ":" followed by your username.
```

请遵循 Stack Overflow - Android Studio: /dev/kvm device permission

`denied` 中的第一个解决方案。然后，注销并重新登录 Linux。

```
sudo adduser $USER kvm
sudo chown $USER -R /dev/kvm
```

在 Windows 上设置 Android SDK 环境

1. 安装 Android 命令行工具。前往 [Android Developer](#) 并下载与你的操作系统匹配的工具。

仅限命令行工具

| 平台 | SDK 工具包 | 尺寸 | SHA-256 校验和 |
|---------|--|----------|--|
| Windows | commandlinetools-win-11076708_latest.zip | 153.6 MB | 4d6931209eebb1fb7c7e8b240a6a3cb3ab24479ea294f3539429574b1eec862 |
| Mac | commandlinetools-mac-11076708_latest.zip | 153.6 MB | 7bc5c72ba0275c80a8f19684fb92793b83a6b5c94d4d179fc5988930282d7e64 |
| Linux | commandlinetools-linux-11076708_latest.zip | 153.6 MB | 2d2d50857e4eb553af5a6dc3ad507a17adf43d115264b1afc116f95c92e5e258 |

Android Studio 中包含了命令行工具。如果您不需要 Android Studio，可以下载上面的基本 Android 命令行工具。您可以使用随附的 [sdkmanager](#) 下载其他 SDK 软件包。

将下载的 ZIP 文件解压到你希望安装 SDK 的目录中。

将 cmdline-tools 文件夹中的内容移动到最新文件夹中。最终结构应为：

```
D:\AndroidSDK\cmdline-tools\latest
```

2. 配置相关环境。

安装 Java。（如果你之前已经安装并配置过 Java，你需要检查你的 JDK 是否与命令行工具适配。如果版本适配，你可以跳过以下关于 Java 环境配置的步骤）

访问 [Oracle JDK](#) 的官方网站，选择适合 Windows 的版本并下载。

在这里，你可以使用 JDK-21。

然后运行下载的安装程序，按照提示完成安装。确保记下安装路径以备将来参考。

```
D:\Java\jdk-21
```

3. 设置环境变量。

打开环境变量设置：

- 右键单击 此电脑或 计算机，选择 属性。
- 点击 高级系统设置。

- 在 系统属性窗口中选择 环境变量。

在 系统变量部分，点击 新建添加 JAVA_HOME 变量，值设置为 JDK 安装路径 (D:\Java\jdk-21)。并添加 ANDROID_HOME 变量，值设置为 AndroidSDK 安装路径 (D:\AndroidSDK)。

找到 Path 变量，点击 编辑，然后添加 %JAVA_HOME%\bin、%ANDROID_HOME%\tools、%ANDROID_HOME%\emulator、%ANDROID_HOME%\cmdline-tools\latest\bin、%ANDROID_HOME%\tools\bin、%ANDROID_HOME%\cmdline-tools\latest 和 %ANDROID_HOME%\platform-tools。

4. 验证安装

打开命令提示符：按 Win + R，输入 cmd，然后按 Enter。

在命令提示符窗口中，输入 java -version 和 javac -version，然后按 Enter。

```
java -version  
javac -version  
sdkmanager --version
```

如果显示版本信息，则表示配置成功。

5. 验证 sdkmanager 是否成功安装。

```
sdkmanager --update  
sdkmanager --list  
sdkmanager --licenses
```

在这里，你应该会看到信息显示 All SDK package licenses accepted。

你可以从 [sdkmanager 的常用命令](#) 了解更多。

在 MAC OS 上设置 Android SDK 环境

1. 安装 Android 命令行工具。前往 [Android Developer](#) 并下载与你的操作系统匹配的工具。

仅限命令行工具

| 平台 | SDK 工具包 | 尺寸 | SHA-256 校验和 |
|---------|--|----------|--|
| Windows | commandlinetools-win-11076708_latest.zip | 153.6 MB | 4d6931209eebb1bf7c7e8b240a6a3cb3ab24479ea294f3539429574b1eec862 |
| Mac | commandlinetools-mac-11076708_latest.zip | 153.6 MB | 7bc5c72ba0275c80a8ff9684fb92793b83a6b5c94d4d179fc5988930282d7e64 |
| Linux | commandlinetools-linux-11076708_latest.zip | 153.6 MB | 2d2d50857e4eb553af5a6dc3ad507a17adf43d115264b1afc116f95c92e5e258 |

Android Studio 中包含了命令行工具。如果您不需要 Android Studio，可以下载上面的基本 Android 命令行工具。您可以使用随附的 [sdkmanager](#) 下载其他 SDK 软件包。

你可以在命令行中使用 wget 下载工具下载压缩包，也可以从浏览器中下载并自行解压。

你可以从上述 Android Developer 网站复制下载链接。然后使用以下命令。

```
wget https://dl.google.com/android/repository/commandlinetools-mac-
→11076708_latest.zip?hl=zh-cn
mkdir -p Android/cmdline-tools
unzip commandlinetools-mac-11076708_latest.zip?hl=zh-cn -d Android/
→cmdline-tools
mv Android/cmdline-tools cmdline-tools Android/cmdline-tools/latest
```

2. 配置相关环境。

安装 Java。(如果你之前已经安装并配置过 Java，你需要检查你的 JDK 是否与命令行工具适配。如果版本适配，你可以跳过以下关于 Java 环境配置的步骤)

我们使用 JDK-17 来运行安卓模拟器。。

```
sudo brew install openjdk@17
```

打开你的 .bashrc 文件。

```
sudo nano ~/.zshrc
```

在文件末尾添加以下内容。

```
export PATH="/opt/homebrew/opt/openjdk@17/bin:$PATH"
export ANDROID_HOME="/Users/your_id_name/the_path_you_store_
→commandline_tools/Android"
export PATH="$ANDROID_HOME/emulator:$ANDROID_HOME/tools:$ANDROID_HOME/
→cmdline-tools/latest/bin:$ANDROID_HOME/tools/bin:$ANDROID_HOME/
→cmdline-tools/latest:$ANDROID_HOME/platform-tools:$PATH"
```

确保 PATH 配置与你存储相关工具的路径匹配。

最后，重新装载 .zshrc 文件，使更改立即应用于当前终端会话。

```
source ~/.zshrc
```

3. 验证 sdkmanager 是否成功安装。

```
sdkmanager --update
sdkmanager --list
sdkmanager --licenses
```

如果你获得如下信息，则安装成功。

```
[=====] 100% Computing updates...
All SDK package licenses accepted.
```

你可以从 `sdkmanager` 的常用命令 了解更多。

创建并运行模拟器

在运行 Kea 之前，你需要创建一个模拟器。有关如何使用 `avdmanager` 创建 avd 的信息，请参见 [此链接](#)。以下示例命令将帮助你创建一个模拟器，从而快速开始使用 Kea：

```
sdkmanager "build-tools;29.0.3" "platform-tools" "platforms;android-29"
sdkmanager "system-images;android-29;google_apis;x86"
avdmanager create avd --force --name Android10.0 --package 'system-images;
→ android-29;google_apis;x86' --abi google_apis/x86 --sdcard 1024M --
→ device "pixel_2"
```

接下来，你可以使用以下命令启动一个模拟器并分配其端口号：

```
emulator -avd Android10.0 -read-only -port 5554
```

2.2 编写第一个性质

2.2.1 检查环境

Kea 是一个基于性质的移动应用测试框架，目前支持 Android 和 HarmonyOS。请确保你拥有一台移动设备，并在你的电脑上安装了 Android/HarmonyOS 命令行工具。检查 `adb` (Android) 或 `hdc` (HarmonyOS) 是否可用。

- 如果你没有真机，可以通过模拟器使用 Kea。
- 请确保你已安装 `python 3.9+`。

2.2.2 安装

使用以下命令安装 Kea

```
git clone https://github.com/ecnusse/Kea.git
cd Kea
pip install -e .
```

输入 `kea -h` 以检查 Kea 是否已成功安装。

2.2.3 编写第一个性质 (安卓)

启动你的设备或 Android 模拟器。在终端中输入 `adb devices` 以确保它可用。

我们将使用 `weditor` 来检查 Android 元素并编写性质。

1. 启动 `weditor` 并安装你的应用。

```
pip install weditor==0.7.3
python -m weditor
```

上述命令将在你的电脑上安装 `weditor` 并启动它。它提供了一个主机服务器（默认：`http://localhost:17310`）。你可以在网页浏览器中访问它。

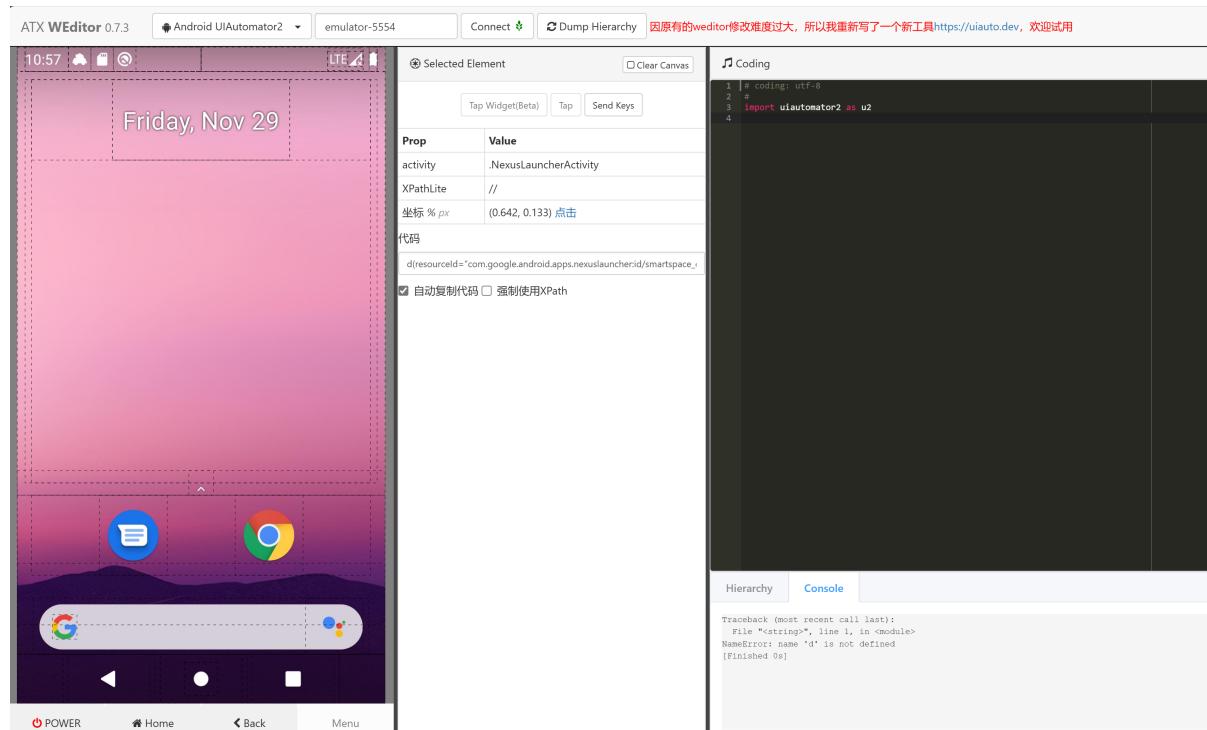


Fig. 6: `weditor` 的主页。

然后，进入 `kea` 工作区并安装应用 `omninoes`。

```
adb install example/omninoes.apk
```

检查应用是否已成功安装。

2. `Dump hierarchy` 并检查 `Android` 元素

在 `weditor` 中 `Dump hierarchy` 以获取 `Android` 元素。

输入设备序列号 -> `Connect` -> `Dump hierarchy`

连接到 `weditor` 后，你可以点击 `Dump hierarchy` 来刷新元素（即 `Dump hierarchy`），每当你屏幕发生变化时。或者，你可以启用自动 `Dump hierarchy`，以避免手动刷新元素。

你可以点击一个元素并检查其属性。

3. 编写你的第一个性质

我们在这个应用中有一个简单的功能需要检查：旋转后搜索输入框不应被清空。



Fig. 7: 从 weditor Dump hierarchy

现在，让我们编写前置条件。这应该是功能开始时的唯一特征。我们想检查搜索输入框，所以让我们先移动到搜索功能。通过点击 搜索按钮，你可以进入搜索编辑页面。显然，这个页面的唯一特征应该是搜索输入框本身。

在 **weditor** 中 **Dump hierarchy**。点击搜索框以检查其属性。

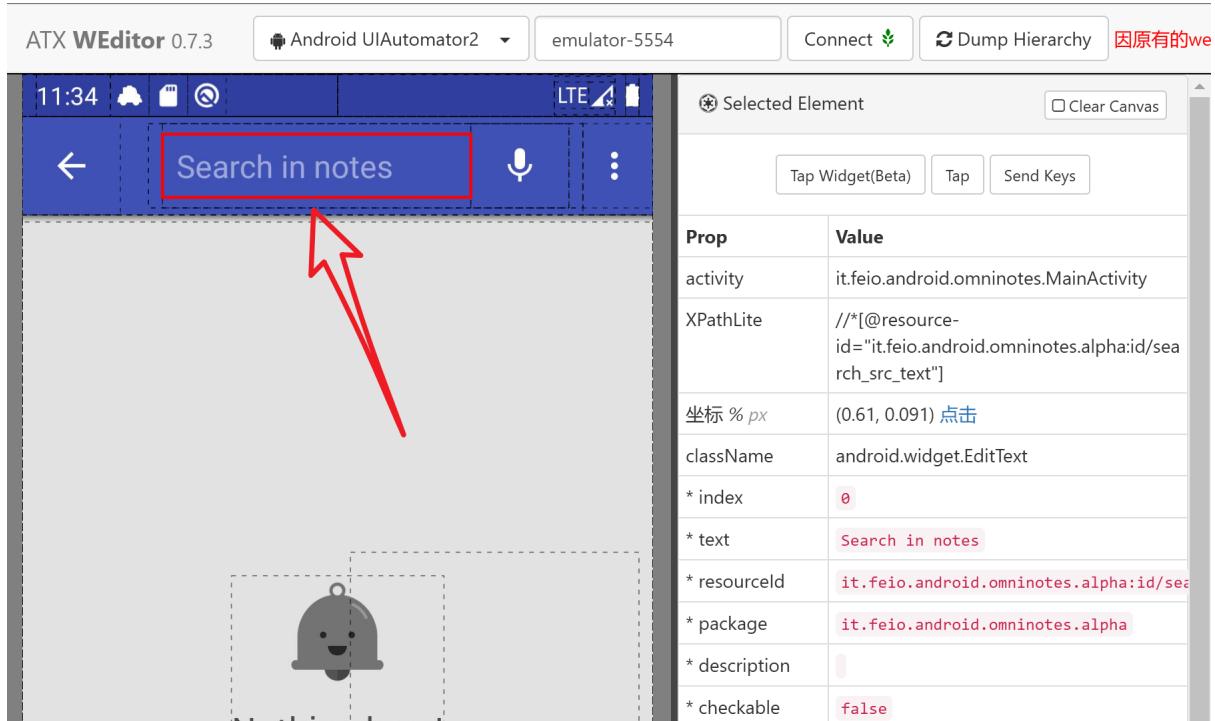


Fig. 8: 在 weditor 中检查一个控件

我们需要特定于控件的属性来定位一个控件。最常用的唯一属性是 **resourceId**。如果没有 **resourceId**, **text** 或 **className** 也可以，但大多数情况下它们不是唯一的，会导致错误。

因此，为了避免 kea 进入错误状态，你可以使用多个属性在选择器中定位一个控件，并使用多个控件定位一个页面。

经过检查，我们知道搜索输入框的 `resourceId`。我们可以用以下命令来定位它。

```
d(resourceId="it.feio.android.omninotes.alpha:id/
search_src_text")
```

Note: 你可能会对 `d(** 选择器)` 脚本感到困惑。这是 kea 的 PDL (Property Description Language, 性质描述语言) 用于与 AUT (被测应用) 交互。你可以阅读[应用性质描述语言接口](#) 以获取详细信息。

要检查这个控件是否存在，我们调用 ```exists```。

```
d(resourceId="it.feio.android.omninotes.alpha:id/
search_src_text").exists()
```

Hint: 双击 weditor 中的控件。这将自动为你生成 `click` 的动作脚本。你可以参考它来编写自己的脚本。

编写交互场景 (即功能的作用)。

我们需要旋转设备。从竖屏状态到横屏，再回到竖屏状态。脚本可以这样编写：`d.rotate('l')` `d.rotate('n')`

编写后置条件。旋转后输入框仍然应该存在。我们使用断言来确认它的存在。

```
assert d(resourceId="it.feio.android.omninotes.alpha:id/
search_src_text").exists()
```

恭喜！你已经编写了你的第一个性质！

4. 使用 Kea API 封装你的性质

在 kea 的根目录下创建一个 Python 文件 `my_prop.py`。

```
#my_prop.py
from kea.core import *

class CheckSearchBox(Kea):
    @precondition(lambda self: d(resourceId="it.feio.android.omninotes.
    ↪alpha:id/search_src_text").exists())
    @rule()
    def search_box_should_exist_after_rotation(self):
        d.rotate('l')
        d.rotate('n')
        assert d(resourceId="it.feio.android.omninotes.alpha:id/search_src_
    ↪text").exists()
```

2.2.4 启动 kea 并检查你的性质

通过以下命令启动 kea。

```
kea -f prop.py -a example/omninoes.apk -o output
```

检查 `output/bug_report.html` 中的错误报告。你可以在本教程中学习如何阅读错误报告：[缺陷报告阅读指南](#)。

2.3 为 HarmonyOS 编写性质

2.3.1 HarmonyOS 的 UI 控件查看工具：uiviewer 教程

我们使用 uiviewer 来支持 HarmonyOS。以下命令将为你安装并启动 uiviewer。

```
pip install -U uiviewer  
python -m uiviewer
```

这将启动一个主机服务器，默认情况下地址为 `http://localhost:8000/`。你可以通过浏览器访问该工具。

2.3.2 HarmonyOS PDL API

我们使用 hmdriver2 作为 PDL API，与 uiautomator2 相似。

在 HarmonyOS 中，应用开发者指定的全局唯一的选择器是 `id`（类似安卓中的 `resourceId`）。当没有 `id` 时，你可以使用 `text` 和 `description` 等属性来定位设备。你可以在选择器中填写多个控件属性来确保尽可能准确地定位至目标控件。

你可以在 [github-hmdriver2](#) 中查看 hmdriver2 的使用手册。

以下是一些 HarmonyOS PDL 的示例。

```
# 点击 id 为 "wifi_entry.icon" 的控件  
d(id="wifi_entry.icon").click()  
  
# 点击 id 为 "display_settings.title" 且 text 为 "Display" 的控件  
d(id="display_settings.title", text="Display").long_click()  
  
# 向 id 为 "url_input_in_search" 的控件输入 "hello"  
d(id="url_input_in_search").input_text("hello")
```

2.3.3 为 HarmonyOS 启动 Kea

你应该在 config.yml 中指定你电脑的系统。你可以查看 YAML 配置 的教程。

以下是一个示例。

```
# config.yml

# 声明电脑的系统为 Linux
env: Linux
```

你可以在终端或 config.yml 中指定其他参数。有关详细信息，请查看提供的 config.yml。

如果你在 config.yml 中指定了所有必要的参数，可以通过 kea -load_config 启动 kea。以下示例是一个完全配置的示例，可以通过 kea -load_config 启动。

```
# env: 你电脑的系统 (例如 windows, macOS, Linux)
env: Linux

# system: 目标 HarmonyOS
system: harmonyOS

device: 127.0.0.1:5555          # 仅连接一个设备时可不填，就自动指定
app_path: example/example.hap   # 应用安装包
policy: random                  # 输入策略
count: 100
properties:
  - example/example_hm_property.py

# package 用于通过包名指定待测应用。此选项和通过 app_path 指定安装包请二选一。
# 当你使用包名指定待测应用时，你一般需要让 keep_app 为 True，这让测试停止时应用不会被卸载。
package: com.youku.next
keep_app: True
```

2.4 应用性质描述语言接口

应用性质描述语言（PDL）是 Kea 与被测试应用交互的方式，用户可通过接口的调用来实现与被测移动应用的交互。

2.4.1 用户界面交互事件

Note: 目前，kea 的性质描述语言底层通过 uiautomator2 作为交互工具，来进行与移动设备的交互。

例如，要向应用程序发送点击事件，你可以使用以下代码：

```
d(resourceId="player_playback_button").click()
```

d 是 uiautomator2 的驱动。

`resourceId` 设置组件的编号，用于选择器定位组件。

`click()` 表示向该组件发送点击事件。

下面是一些常用的交互事件：

- `click`

```
d(text="OK").click()
```

- `long_click`

```
d(text="OK").long_click()
```

- `edit text`

```
d(text="OK").set_text("text")
```

- `rotate device`

```
d.rotate("l") # or left  
d.rotate("r") # or right
```

- `press [key]`

```
d.press("home")  
d.press("back")
```

在定位组件时，可以使用以下选择器。

2.4.2 选择器

选择器用于在用户界面中识别特定的组件，其支持以下参数：

- `text, textContains, textMatches, textStartsWith`
- `className, classNameMatches`
- `description, descriptionContains, descriptionMatches, descriptionStartsWith`
- `checkable, checked, clickable, longClickable`
- `scrollable, enabled, focusable, focused, selected`
- `packageName, packageNameMatches`
- `resourceId, resourceIdMatches`
- `index, instance`

2.4.3 样例

```
# 选择 text 值为 "More Options" 的控件并点击它。
d(text='More Options').click()

# 在一个选择器中使用多个参数。
# 选择具有 text 值为 "Clock" 和类名为 "android.widget.TextView" 的控件并点击它。
d(text='Clock', className='android.widget.TextView').long_click()

# 选择具有资源编号为 "com.example/input_box" 的控件，并将其文本值设置为 "Hello
# world"。
d(resourceId="com.example/input_box").set_text("Hello world")
```

2.5 性质定义教程

在本教程中，你将学习如何使用 Kea 编写应用性质并进行测试。

在移动应用中，性质定义了应用的预期行为。如果应用违反了该性质，则意味着发现了一个错误。

用户所定义的应用功能性质由三个关键组件组成。 $\langle P, I, Q \rangle$ ，(1) P 是一个前置条件，(2) I 是一个交互场景，定义了如何执行应用功能，(3) Q 是一个后置条件，定义了预期的行为。

Kea 给用户提供 `@initializer()` 帮助用户定义初始化函数，让应用能够跳过欢迎页面或登录页面。

在 Kea 中，性质是通过应用 `@rule()` 这样一个性质函数上的装饰器来定义的。

要定义性质的前置条件，用户可以在 `@rule()` 装饰的函数上，使用装饰器 `@ precondition()`。

后置条件则在 `@rule()` 装饰的函数内部使用 `assert` 来完成定义。

对于移动应用，用户可以从多个途径获取应用性质，例如应用的规范、应用的文档、应用的测试用例、应用的错误报告等。

让我们从几个简单的例子开始，介绍如何获取一个性质，如何在 Kea 中编写该性质，以及如何通过 Kea 测试该性质。

2.5.1 从应用错误报告中获取应用性质

以下这个例子将展示如何从应用 `OmniNotes` 中获取一个性质。

`OmniNotes` 是一个用于记录和管理笔记的应用。

本样例来自该应用的错误报告 #634，用户表示，当他删除一个标签时，其他共享相同前缀的标签也被删除。

然后，从这个错误报告中，可以得到一下应用性质：

在删除标签后，标签应该成功移除，笔记内容应保持不变。

根据错误报告，你可以得到一个这样的应用性质：

- P (前置条件)： 应该有标签存在。

- **I (交互场景)**: 从标签列表中移除某个标签。
- **Q (后置条件)**: 指定的标签被删除，并且其余文本内容保持不变。

接下来，让我们在 Kea 中使用性质描述语言定义该性质。

```
@precondition(lambda self: d(resourceId="it.feio.android.omnimotes:id/menu_tag").exists() and
             "#" in d(resourceId="it.feio.android.omnimotes:id/detail_content").info["text"])
)
@rule()
def rule_remove_tag_from_note_shouldnot_affect_content(self):
    # get the text from the note's content
    origin_content = d(resourceId="it.feio.android.omnimotes:id/detail_content").info["text"]
    # click to open the tag list
    d(resourceId="it.feio.android.omnimotes:id/menu_tag").click()
    # select a tag to remove
    selected_tag = random.choice(d(className="android.widget.CheckBox",
                                    checked=True))
    select_tag_name = "#" + selected_tag.right(resourceId="it.feio.android.omnimotes:id/md_title").info["text"].split(" ")[0]
    selected_tag.click()
    # click to uncheck the selected tag
    d(text="OK").click()
    # get the updated content after removing the tag
    new_content = d(resourceId="it.feio.android.omnimotes:id/detail_content").info["text"].strip().replace("Content", "")
    # get the expected content after removing the tag
    origin_content_exlude_tag = origin_content.replace(select_tag_name, "").strip()
    # the tag should be removed in the content and the updated content should be the same as the expected content
    assert not d(textContains=select_tag_name).exists() and new_content == origin_content_exlude_tag
```

`@precondition` 装饰器定义了该性质应当开始被测试的状态节点。代码中，`d(resourceId="it.feio.android.omnimotes:id/menu_tag").exists()` 检查了是否标签按钮存在于界面内，`"#" in d(resourceId="it.feio.android.omnimotes:id/detail_content").info["text"]` 检查了是否笔记内容中存在“#”字符。

`@rule()` 装饰器定义了应用性质函数。在本段代码中，交互场景为执行移除标签的操作。

后置条件则由 `assert` 语句来完成定义。这里，Kea 检查是否指定的标签被删除并且保持其余文本不变。

像这样一条性质就是应该由 `OmniNotes` 应用所遵循的。

此外，用户还可以定义一个初始化函数，在测试性质之前设置应用的初始状态。

为了实现该功能，用户可以使用一个 `@initializer()` 装饰器来定义一个初始化函数并且写一些 UI 操作指令，来引导应用完成初始化操作：

```
@initializer()
def set_up(self):
    for _ in range(5):
```

(continues on next page)

(continued from previous page)

```

d(resourceId="it.feio.android.omnинotes:id/next").click()
d(resourceId="it.feio.android.omnинotes:id/done").click()
if d(text="OK").exists():
    d(text="OK").click()

```

在这里，上述代码可以自动通过 UI 操作来跳过 OmniNotes 的欢迎页面。你可以使用 `@initializer()` 装饰器来定义任意应用的初始化函数。这样，Kea 会在测试应用性质之前执行该初始化函数。这样可以确保在每次测试开始时，应用都处于预期的初始状态。

Tip: 这个功能可以用来在测试应用性质之前设置应用程序的初始状态。例如，可以使用此功能进行登录、向应用程序添加数据等。如果不需要设置应用程序的初始状态，可以跳过此步骤。

此外，如果用户想使用主路径引导探索策略，需要使用 `@mainPath()` 装饰器定义一个函数来设置一个主路径函数。

为了给该应用完成该步骤，可以使用以下代码来定义主路径。

```

@mainPath()
def test_main(self):
    d(resourceId="it.feio.android.omnинotes.alpha:id/fab_expand_menu_button"
      ).long_click()
    d(resourceId="it.feio.android.omнинotes.alpha:id/detail_content").
      click()
    d(resourceId="it.feio.android.omнинotes.alpha:id/detail_content").set_
      text("read a book #Tag1")
    d(description="drawer open").click()
    d(resourceId="it.feio.android.omнинotes.alpha:id/note_content").click()

```

上述代码可以引导 Kea 在 Omninotes 中创建一条内容为 “read a book #Tag1”的笔记。

Tip: 在主路径定义部分，只能使用 UI 操作命令来完成定义；该函数目前不支持其他 Python 语句，例如 `for` 循环。但我们认为这种方法足以实现主路径的功能。

太棒了！到此，你已经学会了如何使用性质描述语言从错误报告中提取并定义一个应用性质。

要测试这个性质，用户需要将其放入定义的一个类中，该类继承自 `KeaTest` 类。

```

from kea.main import *

class Test(KeaTest):

    @initialize()
    def set_up(self):
        for _ in range(5):
            d(resourceId="it.feio.android.omnинotes:id/next").click()
            d(resourceId="it.feio.android.omнинotes:id/done").click()
        if d(text="OK").exists():
            d(text="OK").click()

```

(continues on next page)

(continued from previous page)

```

@mainsPath()
def test_main(self):
    d(resourceId="it.feio.android.omninotes.alpha:id/fab_expand_menu_
↪button").long_click()
    d(resourceId="it.feio.android.omninotes.alpha:id/detail_content") .
↪click()
    d(resourceId="it.feio.android.omninotes.alpha:id/detail_content").
↪set_text("read a book #Tag1")
    d(description="drawer open").click()
    d(resourceId="it.feio.android.omninotes.alpha:id/note_content") .
↪click()

@precondition(lambda self: d(resourceId="it.feio.android.omninotes:id/
↪menu_tag").exists() and
            "#" in d(resourceId="it.feio.android.omninotes:id/detail_
↪content").info["text"])
            )

@rule()
def rule_remove_tag_from_note_shouldnot_affect_content(self):
    # get the text from the note's content
    origin_content = d(resourceId="it.feio.android.omninotes:id/detail_
↪content").info["text"]
    # click to open the tag list
    d(resourceId="it.feio.android.omninotes:id/menu_tag").click()
    # select a tag to remove
    selected_tag = random.choice(d(className="android.widget.CheckBox",
↪checked=True))
    select_tag_name = "#" + selected_tag.right(resourceId="it.feio.
↪android.omninotes:id/md_title").info["text"].split(" ")[0]
    selected_tag.click()
    # click to uncheck the selected tag
    d(text="OK").click()
    # get the updated content after removing the tag
    new_content = d(resourceId="it.feio.android.omninotes:id/detail_
↪content").info["text"].strip().replace("Content", " ")
    # get the expected content after removing the tag
    origin_content_exlude_tag = origin_content.replace(select_tag_name,
↪ "").strip()
    # the tag should be removed in the content and the updated content_
↪should be the same as the expected content
    assert not d(textContains=select_tag_name).exists() and new_
↪content == origin_content_exlude_tag

```

在这里，需要在继承自 KeaTest 类的 Test 类中编写定义该性质。

我们将这个性质脚本文件 example_mainpath_property.py 放在 example 目录中。用户可以通过运行以下命令来测试应用的该性质。

```
kea -f example/example_mainpath_property.py -a example/omninotes.apk
```

当你尝试测试这个性质时，你可能会迅速发现两个新的错误，这些错误违反了该性质。然后，你可以撰写相应的错误报告并提交给应用程序的开发人员。这两个错误目前都已被开发人员修复。

你可以查看这两个错误的报告：

1. Bug Report: Note tag cannot be removed.

2. Bug Report: Deleting One Tag in a Note Affects Another Tag in the Same Note.

2.5.2 从指定应用程序功能中提取性质

接下来是一个完整的示例，展示了如何从应用 Amaze 的功能中提取性质。

Amaze 是一个文件管理应用程序。它提供了简洁直观的用户界面，允许用户轻松浏览、管理和操作文件。

在 Amaze 中，你可以创建一个文件夹，并且在创建后新文件夹应该存在。因此，你可以定义一个性质 `create_folder_should_exist`。这意味着当你想要创建一个文件夹时，它应该能够被成功创建。

你任然需要使用 `@rule()` 和 `@precondition()` 来完成应用性质的定义。在这个样例中，前置条件 P 是创建新文件夹的按钮需要存在，并处于能够创建文件夹的界面上。交互场景 I 是一些创建文件夹的操作事件序列。最后，后置条件 Q 是检查新创建的文件夹是否存在。

```
@precondition(lambda self: d(resourceId="com.amaze.filemanager:id/sd_main_
↪fab").exists() and
            not d(textContains = "SDCARD").exists())
@rule()
def create_folder_should_exist(self):
    d(resourceId="com.amaze.filemanager:id/sd_main_fab").click()
    d(resourceId="com.amaze.filemanager:id/sd_label", text="Folder").
    ↪click()
    file_name = self._files.get_random_value()
    d.send_keys(file_name, clear=True)
    d(resourceId="com.amaze.filemanager:id/md_buttonDefaultPositive").
    ↪click()
    d(scrollable=True).scroll.to(resourceId="com.amaze.filemanager:id/
↪firstline", text=file_name)
    assert d(text=file_name).exists()
```

太好了！你已经学会了如何从应用程序功能中编写应用性质。

Note: 用户可以在一个 `.py` 文件中编写一个应用程序的单个性质或多个性质。也可以将多个性质写在多个 `.py` 文件中。如果选择第一种方法，用户需要确保在一个 `.py` 文件中最多只有一个 `@initializer()` 和一个 `@mainPath()`，同时有多个 `@rule()` 和 `@precondition()` 来对应不同的性质。测试用例的结构如下图所示（请根据需要添加图像或示例代码）。

2.6 Kea 的参数选项

2.6.1 Kea 的参数

Kea 提供了以下选项。使用 `kea -h` 查看详细信息。

以下参数是 `kea` 中最重要的参数。大多数时候，你需要指定它们。

`-f`: 包含性质的测试文件。你可以针对多个文件运行多个性质。参见当运行多个性质时 `kea` 做了什么？。

-a --apk: 被测试应用的安装包文件或包名。

-d --device_serial: 用于测试的设备的序列号。(当只连接了一个设备时, 此参数可省略, kea 将自动指定目标设备。你可以使用 ‘adb devices’ 或 ‘hdc list targets’ 查找你的目标设备)

-o --output: 执行结果的输出目录。(默认: “output”)

-p --policy: 探索策略的名称。(“random”、“guided” 或 “lrm”)

-is_emulator: 声明目标设备为模拟器。

以下是启动 kea 的一些示例。

```
# 快速开始, 默认随机策略, 输出到 "output" 目录
kea -f my_property.py -a myapp.apk

# 自定义策略
kea -f my_property.py -a myapp.apk -p guided

# 使用多个性质
kea -f my_property1.py my_property2.py -a myapp.apk

# 自定义输出目录
kea -f my_property.py -a myapp.apk -o my_output

# 当有多台设备连接到你的电脑时, 指定目标设备
kea -f my_property.py -a myapp.apk -d emulator-5554 -is_emulator
```

以下是用于自定义 kea 的命令。

-t --timeout: 测试时间 (秒)。

-n: 每 n 个事件后, 重新启动应用, 默认为 100。

-debug: 以调试模式运行 (输出调试信息)。

-keep_app: 测试后保留设备上的应用。

-grant_perm: 安装时授予所有权限。对 Android 6.0+ 有用。

-is_harmonyos: 使用 HarmonyOS 设备

-load_config: 从 config.yml 加载配置。config.yml 中的设置将覆盖命令行参数。

-utg: 生成 UTG 图。

2.6.2 YAML 配置

你可以使用 YAML 配置来启动 kea。在你的 kea 根目录中找到 config.yml。

config.yml 文件用于简化通过配置文件指定参数的过程。请注意, config.yml 中的参数值将覆盖通过命令行指定的参数值。

配置参数对应关系

以下是 config.yml 文件中的参数与 Kea 参数对象中的对应关系:

system: 对应 -is_harmonyos, 用于指定是否使用 HarmonyOS 设备。

`app_path, package, package_name`: 这些参数对应 `-a`, 用于指定应用的 apk 文件路径或包名。

`policy`: 对应 `-p`, 用于指定探索策略。

`output_dir`: 对应 `-o`, 用于指定执行结果的输出目录。

`count`: 对应 `-n`, 用于指定测试次数。

`target, device, device_serial`: 这些参数对应 `-d`, 用于指定测试使用的设备序列号。

`property, properties, file, files`: 这些参数对应 `-f`, 用于指定包含性质的测试文件列表。

以下是配置的示例。

```
# env: 你的电脑系统（例如 windows, macOS, Linux）
env: Linux

# system: 目标系统
system: android
# system: harmonyOS

device: emulator-5554
app_path: example/omnинotes.apk
policy: guided
count: 100
properties:
- example/example_property.py
- example/example_mainpath_property.py
# - example/advanced_example_property.py
```

一旦完成配置，你可以简单地使用 `kea -load_config` 启动 `kea`。

Important: 当你在 HarmonyOS 上使用 `kea` 时, `config.yml` 是必需的。

2.6.3 当运行多个性质时 `kea` 做了什么？

默认情况下，随机和主路径引导探索策略在每次运行中验证应用的一个性质。当应用有多个性质可用时，这两种策略可以一起验证这些性质的任何子集。一个好处是 `Kea` 可以提高验证性质的效率。另一个好处是多个性质的交互场景提供了应用的部分模型。这个部分模型使我们更有可能在测试期间达到应用的更深层次状态。

具体来说，要一起验证多个性质，随机策略会检查多个性质的前提条件是否满足，并随机选择一个性质进行检查。主路径引导探索策略会随机选择一个性质作为目标，并沿着其主路径进行引导探索。当这个主路径上的每个状态都被探索后，这种策略会随机选择另一个性质作为新目标。此外，当多个性质的前提条件满足时，这种策略会随机选择一个性质进行检查。

你可以在[工作原理](#) 中查看详细信息。

2.7 带状态的测试

带状态的测试是一种软件测试方法，专注于系统在不同状态下的行为和响应。其原理基于状态管理和状态转移，通过设计测试用例来覆盖各种状态及其转换，以确保系统在不同条件下的正确性和一致性。此方法适合应用于需要保持状态前后一致的应用程序，通过设计带状态的测试用例，确保系统在各种状态下正常运行，从而增强软件的可靠性和用户体验。

在移动应用中，一些功能可以根据特定的输入或操作从一个状态转换到另一个状态。因此，需要额外的数据结构来支持这一点。

在 Kea 中，当你编写需要记录状态信息的性质时，可以使用带状态的测试。如以下代码所示，当你想在设备上进行文件或文件夹的相关操作时，例如创建文件、删除文件或重命名文件。

你可以编写以下代码：

```
_files = Kea.Bundle("files")
```

Bundle 类包含以下函数：

- `add(value: str)`

向当前的 Bundle 对象内添加一个新值。

```
self._files.add(file_name)
```

- `delete(value: str)`

从当前的 Bundle 对象中删除一个值。

```
self._files.delete(selected_file_name)
```

- `update(value: str, new_value: str)`

将当前对象中 `value` 的值更新为 `new_value`

```
self._files.update(file_name, new_name)
```

- `get_all_data()`

该函数会返回当前 Bundle 对象存储的值列表。

```
self._files.get_all_data()
```

- `get_random_value(value_len: int = 10)`

该函数会随机生成一个值并返回。因此，你可以在使用 `add` 和 `update` 函数之前调用它。

```
file_name = self._files.get_random_value()
self._files.add(file_name)
```

- `get_random_data()`

该函数会从当前 Bundle 对象存储的值中随机选择一个值并返回。因此，你可以在使用 `delete` 和 `update` 函数之前调用它。

```
file_name = self._files.get_random_data()
self._files.delete(selected_file_name)
```

接下来是一个完整的示例，展示了如何在定义性质时使用 Kea 的状态测试。这个示例将展示如何在应用程序 Amaze 中使用状态测试，Amaze 是一个文件管理应用，允许用户在设备上操作文件或文件夹。这些性质是为了测试文件系统的数据操作是否存在错误而定义的。在这种情况下，带状态的测试至关重要，你可以使用 Bundle 来存储 Kea 创建的所有文件夹，并在整个测试过程中对它们进行操作。

首先，你可以定义一个 `create_file_should_exist` 性质。该性质的实现步骤如下：

1. 返回到主目录。
2. 创建一个文件。
3. 检查新文件是否存在。这个性质可以确保在创建文件后，文件确实存在于预期的位置。

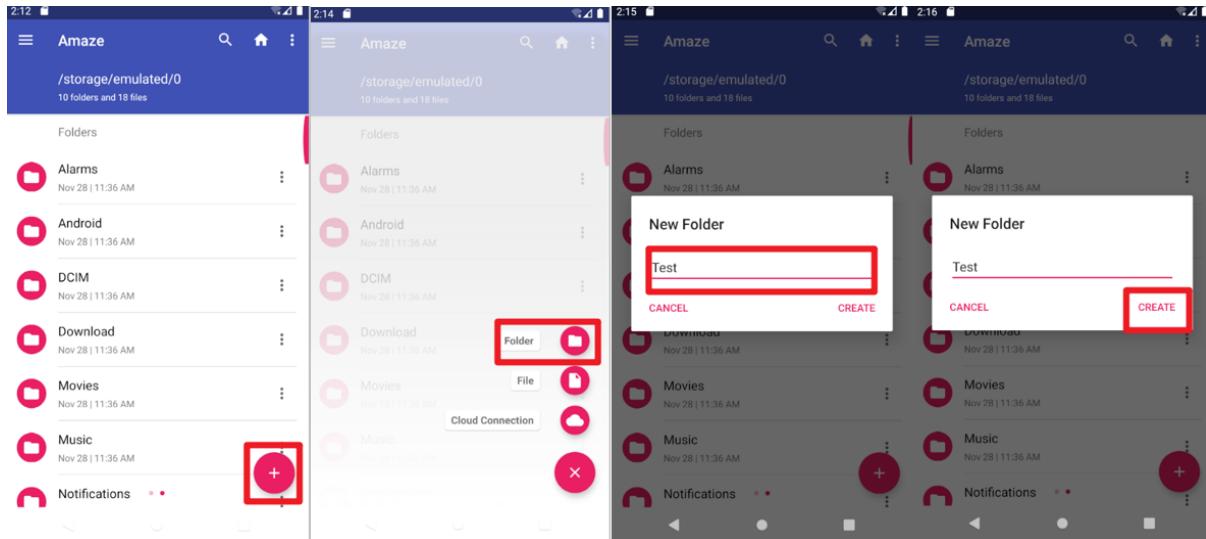


Fig. 9: 创建文件夹截图

```
@precondition(lambda self: d(resourceId="com.amaze.filemanager:id/sd_main_fab").exists() and not d(textContains = "SDCARD").exists())
@rule()
def create_file_should_exist(self):
    d.swipe_ext("down", scale=0.9)
    d(description="Navigate up").click()
    d(resourceId="com.amaze.filemanager:id/design_menu_item_text", textContains="Internal Storage").click()
    d(resourceId="com.amaze.filemanager:id/sd_main_fab").click()
    d(resourceId="com.amaze.filemanager:id/sd_label", text="Folder").click()
    file_name = self._files.get_random_value()
    d.send_keys(file_name, clear=True)
    d(resourceId="com.amaze.filemanager:id/md_buttonDefaultPositive").click()
    self._files.add(file_name)
    d(scrollable=True).scroll.to(resourceId="com.amaze.filemanager:id/firstline", text=file_name)
    assert d(text=file_name).exists()
```

接下来，你可以定义一个 `change_filename_should_follow` 性质。该性质的实现步骤如下：返回到主目录，随机选择一个文件，改变它的名称，并检查原来名称的文件是否消失并且新名称的文件是否存在。

```
@precondition(lambda self: self._files.get_all_data() and
(continues on next page)
```

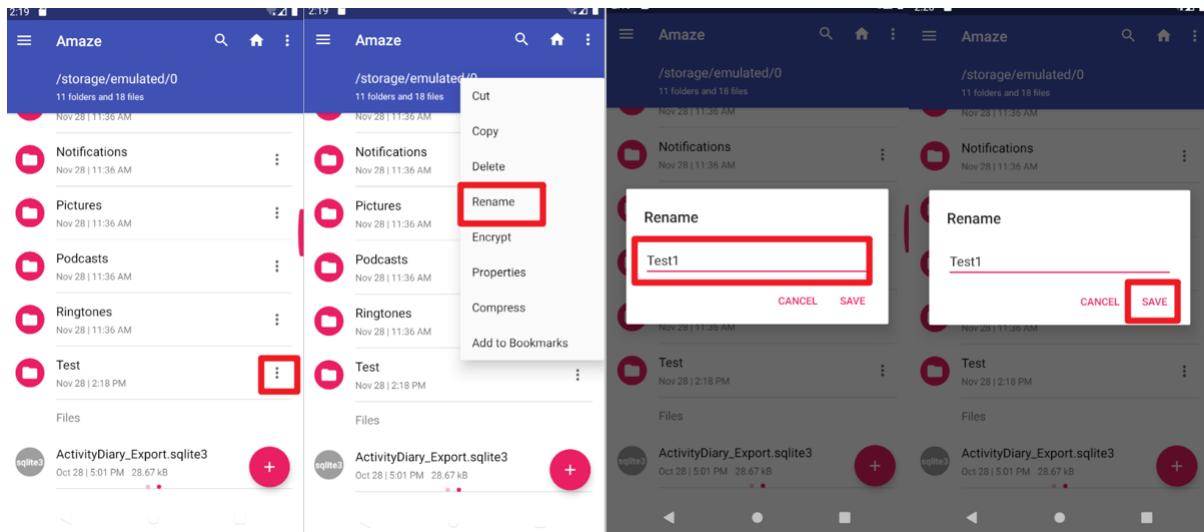


Fig. 10: 重命名文件夹截图

(continued from previous page)

```

d(resourceId="com.amaze.filemanager:id/sd_main_
↪fab").exists() and
    not d(resourceId="com.amaze.filemanager:id/
↪action_mode_close_button").exists())
@rule()
def change_filename_should_follow(self):
    d.swipe_ext("down", scale=0.9)
    d(description="Navigate up").click()
    d(resourceId="com.amaze.filemanager:id/design_menu_item_text",_
↪textContains="Internal Storage").click()
    file_name = self._files.get_random_data()
    new_name = self._files.get_random_value()
    d(scrollable=True).scroll.to(resourceId="com.amaze.filemanager:id/
↪firstline", text=file_name)
    selected_file = d(resourceId="com.amaze.filemanager:id/firstline",_
↪text=file_name)
    selected_file.right(resourceId="com.amaze.filemanager:id/properties").
↪click()
    d(text="Rename").click()
    d.send_keys(new_name, clear=True)
    d(resourceId="com.amaze.filemanager:id/md_buttonDefaultPositive").
↪click()
    self._files.update(file_name, new_name)
    d.swipe_ext("down", scale=0.9)
    d(resourceId="com.amaze.filemanager:id/home").click()
    d(scrollable=True).scroll.to(resourceId="com.amaze.filemanager:id/
↪firstline", text=new_name)
    assert d(text=new_name).exists()
    d.swipe_ext("down", scale=0.9)
    d(resourceId="com.amaze.filemanager:id/home").click()
    d(scrollable=True).scroll.to(resourceId="com.amaze.filemanager:id/
↪firstline", text=file_name)
    assert not d(text=file_name).exists()

```

最后，你可以定义一个 `del_file_should_disappear` 性质。返回到主目录，删除一个文件，并检查该文件是否存在。

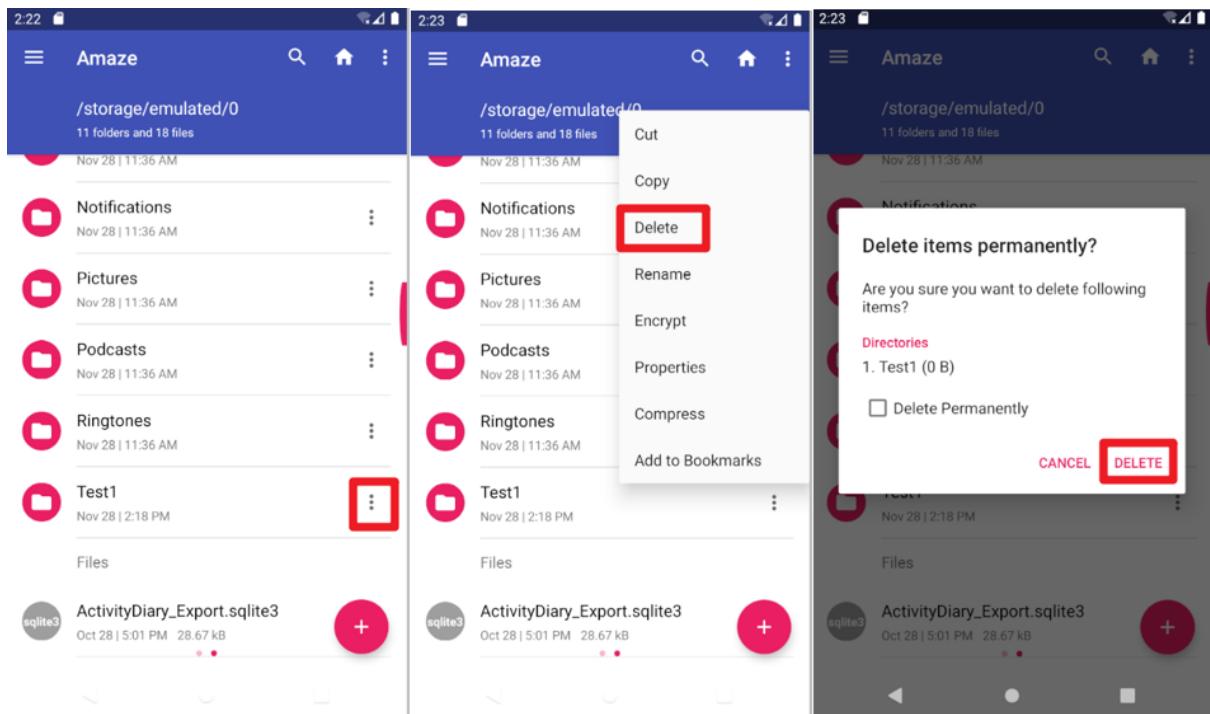


Fig. 11: 删除文件夹截图

```

@precondition(lambda self: self._files.get_all_data() and
              d(resourceId="com.amaze.filemanager:id/sd_main_
              ↪fab").exists() and
              not d(resourceId="com.amaze.filemanager:id/
              ↪action_mode_close_button").exists())
@rule()
def del_file_should_disappear(self):
    d.swipe_ext("down", scale=0.9)
    d(description="Navigate up").click()
    d(resourceId="com.amaze.filemanager:id/design_menu_item_text", □
      ↪textContains="Internal Storage").click()
    file_name = self._files.get_random_data()
    d(scrollable=True).scroll.to(resourceId="com.amaze.filemanager:id/
    ↪firstline", text = file_name)
    selected_file = d(resourceId="com.amaze.filemanager:id/firstline", □
      ↪text = file_name)
    selected_file_name = selected_file.get_text()
    selected_file.right(resourceId="com.amaze.filemanager:id/properties").
    ↪click()
    d(text="Delete").click()
    d(resourceId="com.amaze.filemanager:id/md_buttonDefaultPositive").
    ↪click()
    self._files.delete(selected_file_name)
    d.swipe_ext("down", scale=0.9)
    d(resourceId="com.amaze.filemanager:id/home").click()
    d(scrollable=True).scroll.to(resourceId="com.amaze.filemanager:id/
    ↪firstline", text=file_name)
    assert not d(text=selected_file_name).exists()
  
```

2.8 缺陷报告阅读指南

The figure displays the Kea Test Report interface. At the top, a title bar reads "Kea Tests (Screenshots)". Below it, four screenshots of the "Notes" application are shown, each with a red border around the search bar area. The screenshots are labeled 27, 28, 29, and 30, corresponding to the events "rotate", "rotate", "Click", and "Click". Screenshot 27 shows the search bar at the top. Screenshot 28 shows the search bar has disappeared. Screenshot 29 shows the search bar has reappeared. Screenshot 30 shows the search bar has disappeared again.

Runtime Statistics

- Time needed to found the first bug: 89 seconds
- Time needed to satisfy the precondition at the first time: 22 seconds
- Total Testing Time: 125 seconds

Property Checking Statistics

- Found 1 bugs.
- Satisfied 3 preconditions
- Total Executed Events: 35

Property Violations

| Property Name | Page of Precondition | Pages of Interaction Scenario | Page of Postcondition |
|--|----------------------|-------------------------------|-----------------------|
| search_bar_should_exist_after_rotation | 27 | 27 ~ 28 | 29 |

Fig. 12: 缺陷报告示意图

上面的图像是 Omnimotes 的一个错误报告，用户在旋转屏幕后搜索栏消失了。因此，这违反了 搜索框在旋转屏幕后需要依然存在这一性质。

在测试执行完成之后，你可以在你设置的输出路径下看到 `bug_report.html`。你可以使用浏览器（Google Chrome, Firefox 等）来打开这个 `bug_report.html`。

报告顶部的记录了测试过程中每个用户界面状态的截图，可以帮助你识别和重现该错误。在每个截图下方，你可以看到在该用户界面状态上执行的事件索引和事件类型（例如 `click`, `long click`）。

`Time Consumption Statistics` 模块记录了第一次违反性质的时间，第一次满足前置条件的时间，以及截至目前的测试总用时。

`Statisfaction Quantity Statistics` 模块记录了截至目前违反性质和满足前置条件的总次数，以及总共操作的事件数。

`bug_report.html` 界面下方的表格显示了性质违规列表，包含每个违背性质的 Pre-condition Page (前置条件界面) , Interaction Page (交互场景界面) 和 Postcondition Page (后置条件界面)。点击表格内链接将跳转到相应的截图，以帮助用户了解具体的 bug 触发情况。

设计手册

3.1 架构图

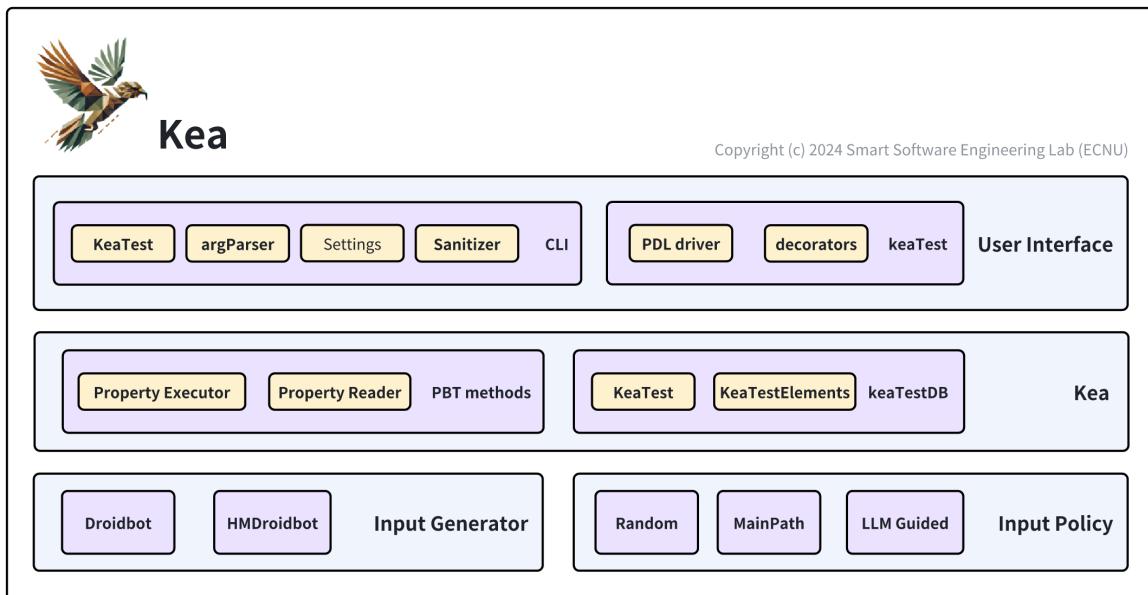


Fig. 1: Kea 架构图

3.2 目录结构

```
.  
├── LICENSE  
├── README.md  
├── config.yml          // kea 的 YAML 配置文件  
├── example/            // kea 示例性质和示例应用  
└── kea/  
    ├── __init__.py  
    ├── resources/        // kea 资源  
    └── adapter/  
        ├── __init__.py   // adapter 抽象类  
        ├── adapter.py    // adapter 实现  
        └── adb.py         // adb 交互操作库
```

(continues on next page)

(continued from previous page)

| | |
|-------------------------|-----------------------|
| cv.py | // cv 库, 用于图像匹配等 |
| hdc.py | // hdc 交互操作库 |
| hilog.py | // 鸿蒙 hilog 操作库 |
| logcat.py | // 安卓 logcat 操作库 |
| minicap.py | // 安卓 minicap 操作库 |
| uiautomator2_helper.py | // uiAutomator2 操作库 |
| app.py | // 安卓应用解析库 |
| app_hm.py | // 鸿蒙应用解析库 |
| Bundle.py | // stateful testing 库 |
| device.py | // 安卓设备库 |
| device_hm.py | // 鸿蒙设备库 |
| device_state.py | // 状态抽象库 |
| droidbot.py | // Droidbot 库 |
| input_event.py | // 输入事件 |
| input_manager.py | // 输入(策略)管理器 |
| input_policy.py | // 输入策略库 |
| intent.py | // intent 操作库 |
| kea.py | // kea 工具 |
| android_pdl_driver.py | // 安卓 PDL 驱动 |
| harmonyos_pdl_driver.py | // 鸿蒙 PDL 驱动 |
| similarity.py | // 组件树结构相似度比对库 |
| start.py | // kea 启动入口 |
| kea_test.py | // kea_test 库 |
| utg.py | // UTG 库 |
| utils.py | // kea 使用的功能函数库 |
| properties/ | // 示例性质 |
| setup.py | // 安装配置 |

3.3 各功能部分设计文档

3.3.1 命令行交互功能

本部分旨在解释 Kea 的命令行界面 (CLI) 是如何设计及实现的，包括如何处理命令行参数、YAML 配置文件以及参数清洗。

功能说明与功能设计

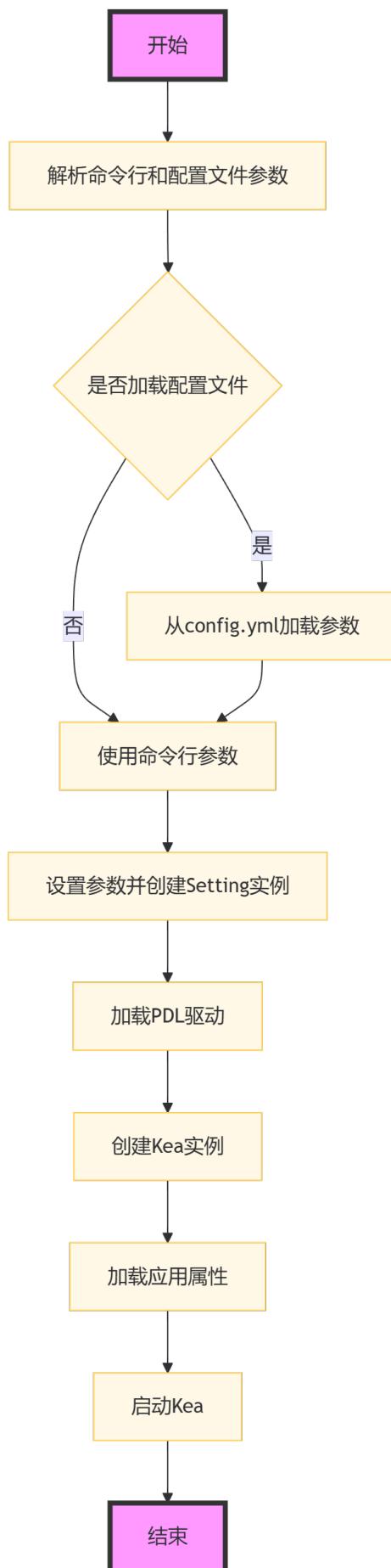
下述流程图的介绍了 Kea 工具 CLI 的启动和配置过程。涵盖了从命令行参数解析、配置文件加载、测试环境设置到自动化测试执行的整个自动化测试准备和启动过程。通过这种方式，用户可以灵活地配置测试环境，并自动化地执行测试用例。

具体执行步骤如下：

1. 解析命令行和配置文件参数

- 使用 argparse 库创建一个参数解析器。
- 定义接受的命令行参数，例如 -f 用于指定性质文件，-d 用于指定设备序列号等。
- 解析命令行输入的参数。

2. 检查是否加载配置文件



- 检查命令行参数中是否包含了 `--load_config` 标志，该标志指示是否从配置文件 `config.yml` 中加载参数。
3. 从 `'config.yml'` 加载参数
- 如果指定了 `--load_config`，则调用 `load_ymal_args` 函数从 `config.yml` 文件中读取参数。
 - 这些参数会覆盖命令行中指定的参数。
4. 使用命令行参数
- 如果没有指定 `--load_config`，则直接使用命令行解析得到的参数。
5. 设置参数并创建 `'Setting'` 实例
- 根据解析得到的参数，创建一个 `Setting` 类的实例，该实例包含了所有需要的配置信息。
6. 加载 `PDL` 驱动
- 根据 `Setting` 实例中的 `is_harmonyos` 属性判断目标设备是 `Android` 还是 `HarmonyOS`。
 - 根据平台加载相应的 `PDL` (`Property Description Language`) 驱动。
7. 创建 `'Kea'` 实例
- 创建 `Kea` 类的实例，`Kea` 可能是一个自动化测试框架的核心类。
8. 加载应用性质
- 使用 `Kea.load_app_properties` 方法加载需要测试的应用性质，这些性质定义了要测试的应用行为。
9. 启动 `'Kea'`
- 调用 `start_keo` 函数，传入 `Kea` 实例和 `Setting` 实例，开始执行自动化测试流程。
 - `start_keo` 函数会初始化 `DroidBot`，它是 `Kea` 的数据生成器，并启动测试。

命令行参数解析

`Kea` 使用 `argparse` 库来解析命令行参数。以下是主要的命令行参数：

- `-f` 或 `--property_files`: 指定要测试的应用性质文件。
- `-d` 或 `--device_serial`: 指定目标设备的序列号。
- `-a` 或 `--apk`: 指定待测应用安装包文件的路径，或待测应用的包名。
- `-o` 或 `--output`: 指定输出目录，默认为“`output`”。
- `-p` 或 `--policy`: 指定输入事件生成策略，默认为“`random`”。
- `-t` 或 `--timeout`: 指定超时时间(秒)，默认为预设值。
- `-debug`: 开启调试模式，输出调试信息。
- `-keep_app`: 测试后保留设备上的应用。
- `-grant_perm`: 安装时授予所有权限，对 `Android 6.0+` 有用。

- `-is_emulator`: 声明目标设备为模拟器。
- `-is_harmonyos`: 使用 HarmonyOS 设备。
- `-load_config`: 从 `config.yml` 加载参数，命令行参数将被忽略。

YAML 配置文件

Kea 支持通过 YAML 配置文件 (`config.yml`) 来指定参数，以简化参数的配置过程。YAML 文件中的参数值将覆盖命令行参数。

参数对象

Kea 使用 `dataclass` 定义了一个名为 `Setting` 的参数对象，用于存储和传递参数。这个对象包含了所有与测试相关的配置。

启动 Kea

以下是 Kea 启动流程的简要说明：

1. 解析命令行参数和 YAML 配置文件。
2. 设置参数对象 `Setting`。
3. 根据目标平台加载相应的 PDL 驱动。
4. 创建 Kea 实例并加载应用性质。
5. 启动 Kea 进行测试。

主要函数设计

以下是 Kea CLI 中每个主要函数的功能介绍：

- **`parse_args` 函数：**

- 负责解析命令行输入的参数。
- 根据用户输入设置相应的命令行参数，并处理 `-load_config` 选项以决定是否从 YAML 配置文件中加载参数。

其简化代码如下：

```
def parse_args():
    parser = argparse.ArgumentParser(...)
    parser.add_argument(...)
    options = parser.parse_args()

    # load the args from the config file `config.yml`
    if options.load_config:
        options = load_ymal_args(options)

    # sanitize these args
    sanitize_args(options)
```

(continues on next page)

(continued from previous page)

```
return options
```

- ***load_yaml_args*** 函数:

- 负责从 `config.yml` YAML 配置文件中读取参数。
- 将配置文件中的参数值应用到参数对象中，覆盖命令行输入的参数。

其简化代码如下：

```
def load_yaml_args(opts):
    config = get_yml_config()
    key_map = {...}
    for key, value in config.items():
        key_lower = key.lower()
        if value and key_lower in key_map:
            key_map[key_lower](value)

    return opts
```

- ***sanitize_args*** 函数:

- 对解析后的参数进行清洗和验证。
- 确保所有参数在传递给 Kea 之前都是有效和一致的。

其简化代码如下：

```
def sanitize_args(options):
    if not options.device_serial:
        identify_device_serial(options)
    if not options.apk_path or not options.property_files:
        raise Error
    if not options.apk_path.endswith('.apk', '.hap'):
        sanitize_app_package_name(options)
```

- ***Setting*** 数据类:

- 定义了 Kea 运行所需的配置参数的数据结构。
- 存储和管理如 APK 路径、设备序列号、输出目录等参数。

其简化代码如下：

```
class Setting:
    apk_path: str
    device_serial: str = None
    output_dir: str = "output"
    is_emulator: bool = True
    policy_name: str = "default_policy"
    event_count: int = 100
    keep_app: bool = None
    keep_env = None
    number_of_events_that_restart_app: int = 100
    is_harmonyos: bool = False
    generate_utg: bool = False
    is_package: bool = False
```

- **`load_pdl_driver`** 函数:

- 根据目标平台 (Android 或 HarmonyOS) 加载相应的 PDL 驱动。
- 确保 Kea 能够与目标设备的操作系统交互。

其简化代码如下:

```
def load_pdl_driver(settings):
    if settings.is_harmonyos:
        from kea.harmonyos_pdl_driver import HarmonyOS_PDL_Driver
        return HarmonyOS_PDL_Driver(serial=settings.device_serial)
    else:
        from kea.android_pdl_driver import Android_PDL_Driver
        return Android_PDL_Driver(serial=settings.device_serial)
```

- **`start_keo`** 函数:

- 初始化 DroidBot 实例，并设置 Kea 的 PDL 驱动。
- 创建 Kea 实例，加载应用性质，并开始执行测试。

其简化代码如下:

```
def start_keo(kea, settings):

    # droidbot is used as the data generator of Kea
    droidbot = DroidBot(...)
    kea._pdl_driver.set_droidbot(droidbot)
    droidbot.start()
```

- **`main`** 函数:

- 作为程序的入口点，串联起整个 Kea 启动流程。
- 调用其他函数完成参数解析、配置加载、PDL 驱动加载和 Kea 启动。

其简化代码如下:

```
def main():
    options = parse_args()
    settings = Setting(...)
    driver = load_pdl_driver(settings)
    Kea.set_pdl_driver(driver)
    Kea.load_app_properties(options.property_files)
    kea = Kea()
    start_keo(kea, settings)
```

3.3.2 KeaTest

本部分旨在解释 Kea 的性质定义类 KeaTest 是如何设计及实现的。

功能设计与实现

KeaTest 是提供给用户编写性质的测试类。继承 KeaTest 就创建了一个测试样例。

在继承的 KeaTest 子类中，我们编写对应的函数，并通过装饰器的方式定义性质中的初始化函数 (Initializer)、前置条件 (Precondition)、交互场景 (Rule) 和主路径函数 (MainPath)。

KeaTest 是一个空类，本质上是为用户提供了一个容器，Kea 会在此容器中寻找用户自定义的性质，并将其加载入 Kea 中。在实现层面，用户继承 KeaTest 编写自己的性质。而 Kea 通过识别并载入 KeaTest 的子类以实现读取用户的性质。

```
class KeaTest:  
    pass
```

3.3.3 性质装饰器

本部分旨在解释 Kea 的性质定义装饰器是如何设计及实现的。

功能说明与功能设计

在 KeaTest 中，使用装饰器定义性质。装饰器的作用是对函数本身进行修改。在 Kea 中，用户的初始化、前置条件、主路径函数都是一个函数，我们使用装饰器获取函数体，并对这个函数体进行标记。由于 python 中函数为一等对象，我们使用装饰器获取函数体后可以动态地往这个函数对象中设置属性，我们根据不同的装饰器，设置不同的 MARKER 属性标记。在 Kea 加载性质的时候，我们读取如下的数据结构，并将如下的数据结构通过 KeaTestElements 类进行读取，并转换为方便 Kea 读取和处理的数据结构：KeaTestElements。

性质的定义

下述的 @rule 和 @precondition 装饰器将用户定义的一条性质封装在数据结构 Rule 中，并对这个性质的函数进行使用 RULE_MARKER 进行标记。

以下是 Rule 数据数据结构的定义。precondition 用于存放一个函数对象，存储一个计算前置条件的函数。function 用于存储这条性质的交互场景 (interaction scenario)。

```
@attr.s(frozen=True)  
class Rule:  
    # `preconditions` denotes the preconditions annotated with  
    # `@precondition`.  
    preconditions:Callable = attr.ib()  
  
    # `function` denotes the function of @Rule.  
    # This function includes the interaction scenario and the assertions  
    # (i.e., the postconditions).  
    function:Callable = attr.ib()
```

@rule 装饰器用于定义一条性质。其中，RULE_MARKER 为一个常量。

参数

- f: Callable[[Any], None] : 一个交互场景函数对象

利用装饰器修改函数对象，为函数对象添加标记

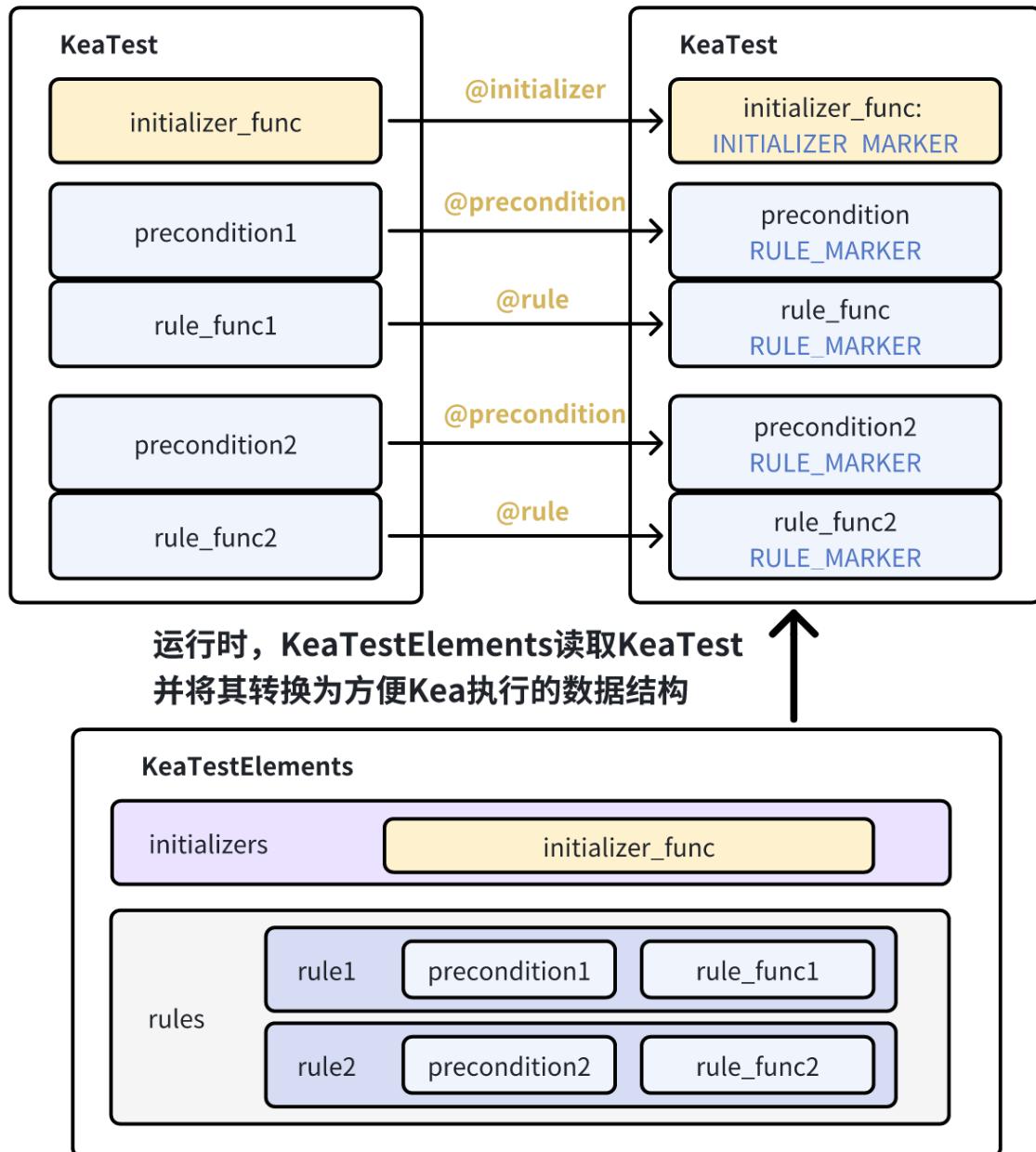


Fig. 3: 从用户自定义 KeaTest 到运行时 KeaTestElements 的转换

返回

- Callable[[Any], None] : 被 RULE_MARKER 标记后已解析 Rule 的函数对象

```
def rule() -> Callable:
    def accept(f):
        precondition = getattr(f, PRECONDITIONS_MARKER, ())
        rule = Rule(function=f, preconditions=precondition)

        def rule_wrapper(*args, **kwargs):
            return f(*args, **kwargs)

        setattr(rule_wrapper, RULE_MARKER, rule)
        return rule_wrapper

    return accept
```

@precondition 前提条件指定了性质何时可以被执行。一个性质可以有多个前提条件，每个前提条件由 @precondition 指定。其中，PRECONDITIONS_MARKER 为一个常量。

参数

- precond: Callable[[Any], bool] : 一个返回布尔值的已经被 @rule 装饰过的函数对象

返回

- Callable[[Any], bool] : 被 RULE_MARKER 标记后已解析前置条件的函数

```
def precondition(precond: Callable[[Any], bool]) -> Callable:
    def accept(f):
        def precondition_wrapper(*args, **kwargs):
            return f(*args, **kwargs)

        rule:"Rule" = getattr(f, RULE_MARKER, None)
        if rule is not None:
            new_rule = rule.evolve(preconditions=rule.preconditions +_
                                   (precond,))
            setattr(precondition_wrapper, RULE_MARKER, new_rule)
        else:
            setattr(
                precondition_wrapper,
                PRECONDITIONS_MARKER,
                getattr(f, PRECONDITIONS_MARKER, ()) + (precond,),
            )
        return precondition_wrapper

    return accept
```

初始化函数的定义

`@initializer` 定义一个初始化函数，用于应用的初始化，如跳过新手教程等。下述的 `@initializer` 装饰器将用户定义的一条性质封装在数据结构 `Initializer` 中，并对这个性质的函数进行使用 `INITIALIZER_MARKER` 进行标记。

以下是 `Initializer` 数据结构的定义。`function` 用于存放一个函数对象，为初始化时要执行的一系列操作。

```
@attr.s()
class Initializer:
    # `function` denotes the function of `@initializer`.
    function: Callable = attr.ib()
```

`@initializer` 装饰器用于定义一个初始化函数，其中，`INITIALIZER_MARKER` 是一个常量。

参数

- `f: Callable[[Any], None]` : 定义了初始化事件的初始化函数对象

返回

- `Callable[[Any], None]` : 被 `INITIALIZER_MARKER` 标记的初始化函数对象

```
def initializer():
    def accept(f):
        def initialize_wrapper(*args, **kwargs):
            return f(*args, **kwargs)

        initializer_func = Initializer(function=f)
        setattr(initialize_wrapper, INITIALIZER_MARKER, initializer_func)
        return initialize_wrapper

    return accept
```

主路径函数的定义

主路径指定了一系列事件，从应用起始页执行这些事件会将应用引到至性质的起始状态（满足前置条件的页面）。下述的 `@mainPath` 装饰器将用户定义的一条性质封装在数据结构 `MainPath` 中，并对这个性质的函数进行使用 `MAINPATH_MARKER` 进行标记。

以下是 `MainPath` 数据结构的定义。`function` 用于存放用户定义的 `mainPath` 函数对象，`path` 为对这个函数进行源代码处理后获取的详细路径步骤，为一个存储了主路径中各个步骤的源代码的列表。

```
@attr.s()
class MainPath:

    # `function` denotes the function of `@mainPath`.
    function: Callable = attr.ib()

    # the interaction steps (events) in the main path
    path: List[str] = attr.ib()
```

@mainPath 装饰器将用户定义的一条性质封装在数据结构 MainPath 中，其中，MAINPATH_MARKER 是一个常量。

参数

- f: Callable[[Any], None] : 定义了主路径事件的函数对象

返回

- Callable[[Any], None] : 被 MAINPATH_MARKER 标记的初始化函数对象

```
def mainPath():
    def accept(f):
        def mainpath_wrapper(*args, **kwargs):
            source_code = inspect.getsource(f)
            code_lines = [line.strip() for line in source_code.
←splitlines() if line.strip()]
            code_lines = [line for line in code_lines if not line.
←startswith('def ') and not line.startswith('@') and not line.startswith('
←#')]
            return code_lines

            main_path = MainPath(function=f, path=mainpath_wrapper())
            setattr(mainpath_wrapper, MAINPATH_MARKER, main_path)
            return mainpath_wrapper

    return accept
```

3.3.4 PDL 驱动

本部分旨在解释 Kea 的性质定义语言驱动 PDL 是如何设计及实现的。

PDL 驱动的功能设计

PDL 驱动是在基于性质的测试中，用户与设备在执行性质时与设备交互的驱动。PDL 驱动有安卓设备的 PDL 驱动（基于 uiautomator2），鸿蒙设备的 PDL 驱动（基于 hmdriver2）

Note: PDL 驱动设计参考了 uiautomator2 和 hmdriver2

uiautomator2: <https://github.com/openatx/uiautomator2>

hmdriver2: <https://github.com/codematrixer/hmdriver2>

PDL 驱动的使用语法是 `d(Selector(**kwargs)).attr(args)`。其中 Selector(**kwargs) 是控件选择器，控件选择器通过字典的方式指定控件的属性，如安卓中的 resourceId, className，鸿蒙中的 id, bundlename 等。attr 是对选定控件的操作，包括 click、longClick 等操作。attr(args) 中的 args 为传入方法的参数。如在 `input_text("Hello")` 中传入要输入的字符串 “Hello”。

我们的 PDL 驱动实际上是 kea 与对应自动化测试工具 (uiautomator2、hmdriver2) 的中间层，语法与目标的测试工具一致，主要用于做一些额外的操作，如保存当前事件、截图等，以方便 kea 访问到对应的操作数据，方便生成错误报告等。

安卓设备的 PDL 驱动的实现

安卓设备的 PDL 驱动通过 uiautomator2 实现。主要用于让用户编写的性质和应用进行交互。

安卓的 PDL 驱动继承于 uiautomator2 的 Driver 类，部分安卓 PDL 的 python 风格简化代码实现示意如下：

```
class Android_PDL_Driver(Uiautomator2.Driver):

    def __call__(self, **kwargs: Any) -> Ui:
        return Ui(self, Selector(**kwargs), droidbot=self.droidbot)

    def set_droidbot(self, droidbot:DroidBot):
        self.droidbot = droidbot

    ...

class Ui(Uiautomator2.UiObject):
    def __init__(self, session:Android_PDL_Driver, selector: Selector, ↴
                 droidbot:DroidBot):
        super().__init__(session, selector)
        self.droidbot=droidbot

    def click(self, offset=None):
        self.droidbot.device.save_screenshot_for_report(event_name="click",
        ↴ event = self)
        print(f"Property Action: click({str(self.selector)})")
        super().click(offset)

    ...

    def child(self, **kwargs):
        return Ui(self.session, self.selector.clone().child(**kwargs), ↴
                  self.droidbot)

    def sibling(self, **kwargs):
        return Ui(self.session, self.selector.clone().sibling(**kwargs), ↴
                  self.droidbot)
```

PDL 的核心功能的解析如下：

- 使 PDL 能按 `d(Selectors(**kwargs)).attr(*args)` 的方式调用：

```
def __call__(self, **kwargs: Any) -> Ui:
    return Ui(self, Selector(**kwargs), droidbot=self.droidbot)
```

python 的函数是一等对象，定义 driver 对象的 `__call__` 魔术方法可以让对象可以通过函数的方式调用，完成形如 `d(**kwargs)` 的调用方法。

`UI` 是 `uiautomator2` 中的 `UI` 对象类，可以调用 `.attrs()` 方法。通过定义驱动的 `__call__` 返回一个 `UI` 对象可以完成如此的调用。

- 使 PDL 驱动能和 kea 主体的其他功能进行资源共享：

kea 调用本 PDL 类的 `set_droidbot` 方法设置 `Droidbot`，让本类可以访问 `droidbot`。以此，需要的资源可以通过调用 `droidbot` 的方法返回给 `kea`。

(continues on next page)

(continued from previous page)

3. 发送资源到 kea。

```
def click(self, offset=None):
    self.droidbot.device.save_screenshot_for_report(event_name="click",
→ event = self)
```

在执行点击操作的时候，调用 droidbot 中的对应方法保存截图和当前的事件操作。其他的控件操作方法定义类似。

4. 让 PDL 能使用 uiautomator2 中的 .child 等相对控件获取方法。

定义 child、 sibling 方法内容，根据功能返回对应的相对控件。

鸿蒙设备的 PDL 驱动实现

鸿蒙设备的 PDL 驱动通过 hmdriver2 实现。主要用于让用户编写的性质和应用进行交互。

鸿蒙的 PDL 驱动继承于 hmdriver2 的 Driver 类，部分鸿蒙 PDL 的 python 风格简化代码实现示意如下：

```
class HarmonyOS_PDL_Driver(hmdriver2.Driver):

    def __call__(self, **kwargs: Any) -> Ui:
        return Ui(self, **kwargs)

    def set_droidbot(self, droidbot:Droidbot):
        self.droidbot = droidbot


class Ui(hmdriver2.UiObject):
    def __init__(self, session:HarmonyOS_PDL_Driver, **kwargs) -> None:
        client = session._client
        droidbot = session.droidbot
        self.droidbot = droidbot
        super().__init__(client, **kwargs)

    def click(self, offset=None):
        self.droidbot.device.save_screenshot_for_report(event_name="click",
→ event = self)
        super().click()
```

1. 使 PDL 能按 `d(Selectors(**kwargs)).attr(*args)` 的方式调用：

```
def __call__(self, **kwargs: Any) -> Ui:
    return Ui(self, **kwargs)
```

python 的函数是一等对象，定义 driver 对象的 `__call__` 魔术方法可以让对象可以通过函数的方式调用，完成形如 `d(**kwargs)` 的调用方法。

UI 是 hmdriver2 中的 UI 对象类，可以调用 `.attrs()` 方法。通过定义驱动的 `__call__` 返回一个 UI 对象可以完成如此的调用。

(continues on next page)

(continued from previous page)

2. 使 PDL 驱动能和 kea 主体的其他功能进行资源共享:

kea 调用本 PDL 类的 `set_droidbot` 方法设置 Droidbot, 让本类可以访问 droidbot。以此, 需要的资源可以通过调用 `droidbot` 的方法返回给 kea。

3. 发送资源到 kea。

```
def click(self, offset=None):
    self.droidbot.device.save_screenshot_for_report(event_name="click",
→ event = self)
```

在执行点击操作的时候, 调用 `droidbot` 中的对应方法保存截图和当前的事件操作。其他的控件操作方法定义类似。

3.3.5 KeaTestElements

本部分旨在解释 Kea 运行时数据管理类 `KeaTestElements` 的设计与实现。

功能设计与实现

`KeaTestElements` 是 Kea 运行时存储用户自定义性质的数据结构, 与用户继承并自定义的 `keaTest` 一一对应。在 `kea` 启动时, `KeaTestElements` 会读取每个用户自定义的 `keaTest`, 并重新组织为方便 `kea` 进行读取的数据结构。具体的转换过程可参考装饰器一章: [从用户自定义 KeaTest 到运行时 KeaTestElements 的转换](#)。

`KeaTestElements` 的数据结构图示如下:

其中, `keaTest_name` 是一个字符串, 用于存储用户定义的 `keaTest` 的类名。`Rules` 是一个列表, 用于存储 `Rule` 对象。`Initializers` 是一个列表, 用于存储初始化函数对象 `Initializer`。`MainPaths` 是一个列表, 用于存储主要路径对象 `MainPath`。

其中, `Rule`、`MainPath` 和 `Initializer` 对象的数据结构及定义可参见“装饰器”一章。

`KeaTestElements` 的成员方法定义伪代码如下:

```
class KeaTestElements:
    def load_rules(keaTest)
    def load_initializers(keaTest)
    def load_mainPaths(keaTest)
```

`load_rules` 接收一个用户自定义的 `keaTest` 对象, 读取其中的 `rule` 并将一个 `keaTest` 中的所有 `rule` 存储入 `rules` 列表。`load_initializers` 接收一个用户自定义的 `keaTest` 对象, 读取其中的初始化函数对象 `Initializer` 并将其存储入 `initializers` 列表。`load_mainPaths` 接收一个用户自定义的 `keaTest` 对象, 读取其中的主路径对象 `mainPath` 并将其存储入 `mainPaths` 列表。

具体而言, 在三个 `load` 方法的执行步骤相似, 其执行步骤可描述如下:

1. 传入一个 `keaTest` 对象。
2. 在传入的 `keaTest` 对象中查找含有相对应 MARKER 标记的函数对象。
3. 将其相应的数据结构 (`Rule`, `Initializer` 和 `MainPath`) 以列表的方式组织存储为成员变量。

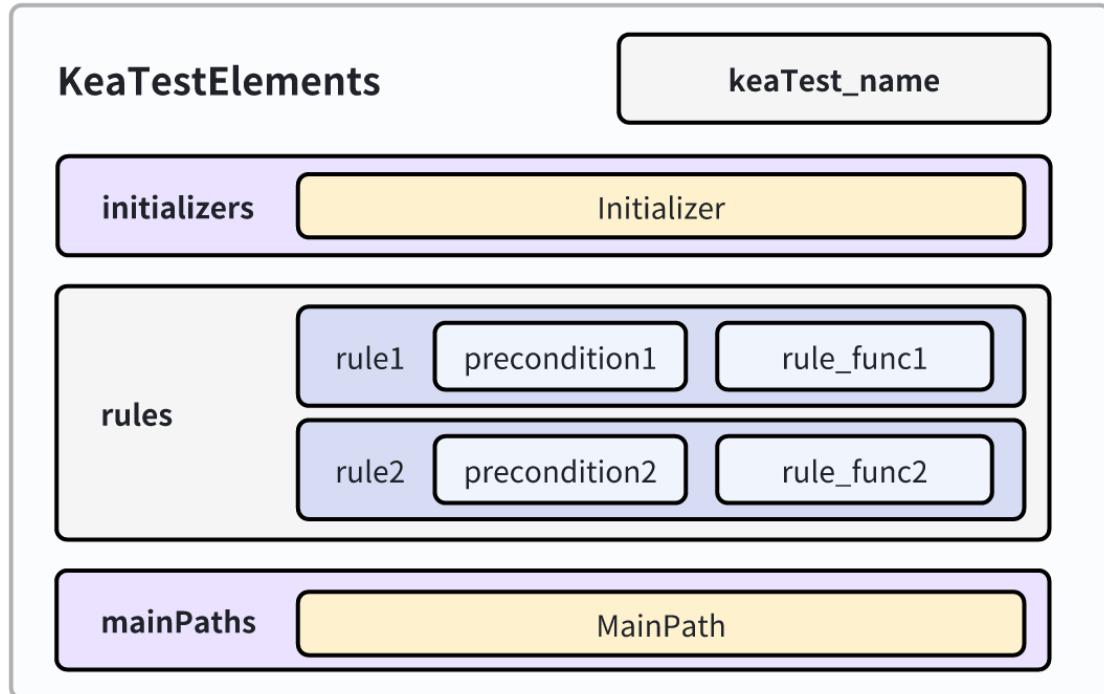


Fig. 4: KeaTestElements 数据结构

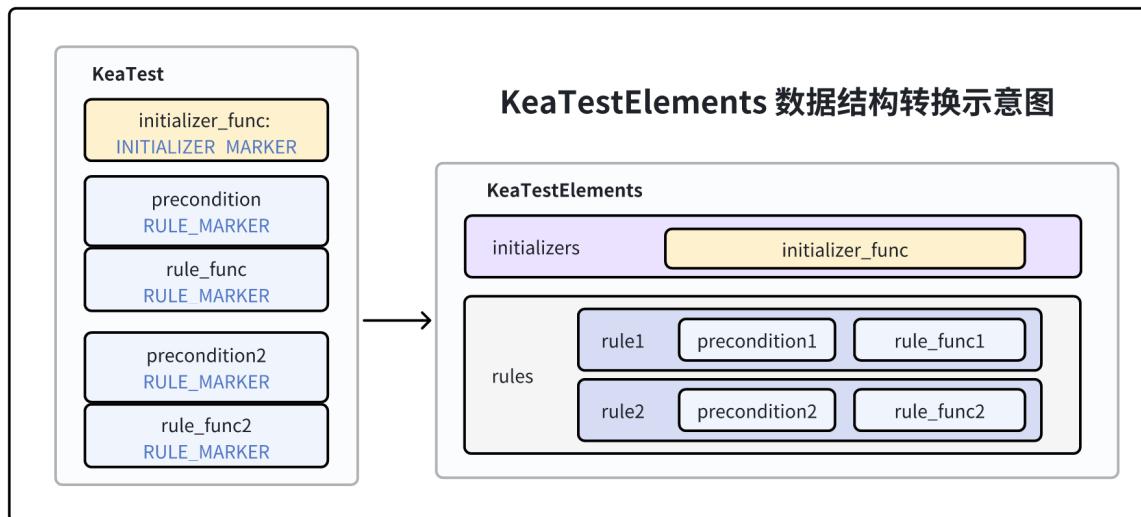


Fig. 5: KeaTestElements 的成员方法读取 KeaTest 中的数据，转换为方便 Kea 使用的数据结构

三个成员方法的具体实现如下：

1. `load_rules`

参数

- `keaTest:KeaTest` : 用户自定义性质的 `keaTest` 对象

返回

- 无

```
def load_rules(self, keaTest:"KeaTest"):
    for _, v in inspect.getmembers(keaTest):
        rule = getattr(v, RULE_MARKER, None)
        if rule is not None:
            self.rules.append(rule)
```

1. `load_initializers`

参数

- `keaTest:KeaTest` : 用户自定义性质的 `keaTest` 对象

返回

- 无

```
def load_initializers(self, keaTest:"KeaTest"):
    for _, v in inspect.getmembers(keaTest):
        initializer = getattr(v, INITIALIZER_MARKER, None)
        if initializer is not None:
            self.initializers.append(initializer)
```

1. `load_mainPaths`

参数

- `keaTest:KeaTest` : 用户自定义性质的 `keaTest` 对象

返回

- 无

```
def load_mainPaths(self, keaTest:"KeaTest"):
    for _, v in inspect.getmembers(keaTest):
        mainPath = getattr(v, MAINPATH_MARKER, None)
        if mainPath is not None:
            self.mainPaths.append(mainPath)
```

3.3.6 Kea

本部分旨在解释 Kea 中的核心控制器类 Kea 的设计与实现。

功能设计与实现

Kea 类是工具中的核心类，为工具中与基于性质测试相关功能的控制器。在 Kea 类中，存储的内容和对外暴露的方法主要有：

- 加载所有 KeaTest 并读取至 KeaTest 至 KeaTestElements 的相关方法。
- 存储用户定义的所有 KeaTestElements。以及访问这些性质相关函数 (initializer, rule, mainPath) 的方法。
- 存储当前运行的 PDL 驱动（含安卓与鸿蒙系统）。以及设置当前要运行的 PDL 驱动的方法。
- 性质的前置条件检查器方法，返回当前应用界面中通过了前置条件的性质。
- 执行一条性质的交互场景的方法。
- 执行主路径步骤的方法。

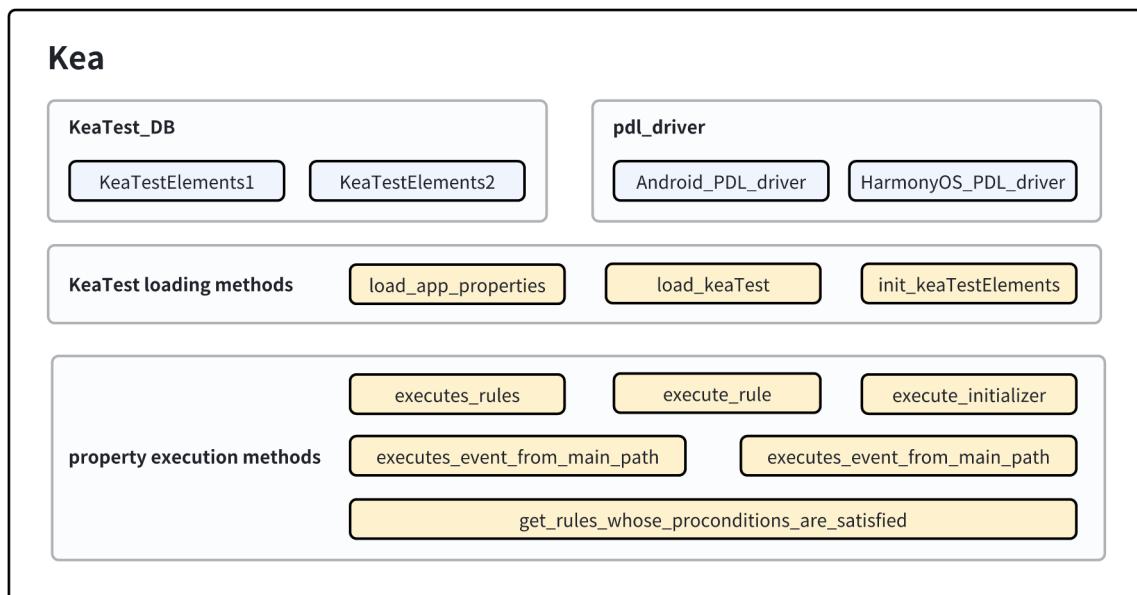


Fig. 6: Kea 类的组成

Kea 类中的数据结构实现

1. KeaTest_DB

KeaTest_DB 是 Kea 运行时的数据库，用于存储所有的用户自定义性质。每个数据项以 <keaTest, keaTestElements> 的键值对方式存储。

2. pdl_driver

pdl_driver 存储的是当前的 PDL (Property Description Language, 性质描述语言) 的驱动。此处存储的是鸿蒙或安卓设备的驱动，主要用于执行主路径中的某一步。

Kea 类中的成员方法实现

读取性质的相关方法

读取性质的相关方法主要对外提供读取用户自定义相关性质的方法。这些性质被读取后会按 keaTestElements 数据结构的方式组织，并存储进入 KeaTest_DB。

1. load_app_properties

load_app_properties 从用户指定的性质文件中读取性质并存储入 Kea，其核心流程如下。

1. 去除重复指定的文件。
2. 对每个文件，检查文件的后缀名是否为 .py 以确认文件有效性。
3. 通过导入模块的方式导入用户定义的性质。
4. 为模块设置对应的系统的 pdl_driver。
5. 检查在步骤 3 中获得的模块的成员，识别其中的用户定义性质 (KeaTest 的子类对象) 并调用 load_KeaTest 方法加载他们。
6. 回到步骤 2 直至所有用户自定义文件被加载完毕。

具体步骤的简化的 python 风格代码如下：

注：为了描述清晰，简化的代码仅对核心流程进行抽象并展示，实际代码与简化的参考代码不完全一致。下文将展示的其他简化代码遵循的规则相同。

参数

- property_files: List[str] : 用户自定义性质的文件路径列表

返回

- 无

```
@classmethod
def load_app_properties(cls, property_files):
    remove_duplicated_files(property_files)

    for file in property_files:
        check_file_basename(file)

        module = get_module_name(file)
```

(continues on next page)

(continued from previous page)

```

try:
    module = importlib.import_module(module_name)

    # set the pdl driver in the modules
    module.d = cls._pdl_driver

    # find all kea tests in the module. Load them.
    for obj in getmembers(module):
        if is_subclass(obj, KeaTest):
            cls.load_KeaTest(obj)

```

1. **load_KeaTest**

`load_KeaTest` 从 `KeaTest` 中取出用户自定义的性质（含初始化函数对象、交互场景、主路径函数对象），转换为 `KeaTestElements` 数据结构并存储入 `KeaTest_DB`。

`load_KeaTest` 的流程如下：

1. 初始化一个 `KeaTestElements`，以 `<KeaTest, KeaTestElements>` 数据项的方式存储进 `KeaTest_DB`
2. 调用 `KeaTestElements` 的方法读取 `KeaTest` 中的性质，并存储进 `KeaTestElements`。

其实现通过简化的 python 代码展示如下：

参数

- `keaTest:KeaTest`： 用户自定义性质的 `keaTest` 对象

返回

- `keaTestElements:KeaTestElements`： 读取了 `keaTest` 对象的 `keaTestElements` 对象

```

@classmethod
def init_KeaTestElements(cls, keaTest):

    keaTest_name = get_keatest_name(keaTest)
    keaTestElements = KeaTestElements(keaTest_name)
    KeaTest_DB.add_item(keaTest, KeaTestElements)
    return keaTestElements


@classmethod
def load_KeaTest(cls, keaTest):

    keaTestElements = cls.init_KeaTestElements(keaTest)
    keaTestElements.load_initializers(keaTest)
    keaTestElements.load_rules(keaTest)
    keaTestElements.load_mainPaths(keaTest)

```

性质执行相关方法

性质执行的相关方法主要对外提供与性质执行相关功能的接口，主要交由输入策略调用。如获取当前页面通过前置条件的性质，执行一条性质的交互场景等。

1. execute_rules

对一组 rules(性质)，随机选择一条性质并调用 execute_rule 方法尝试执行之。

其 python 风格的简化代码定义实现如下。

参数

- rules:List[Rule] : 性质交互场景列表

返回

- 执行结果

```
def execute_rules(rules):

    if rules is empty:
        return CHECK_RESULT.PRECON_NOT_SATISFIED
    rule_to_check = random.choice(rules)
    execute_rule(rule_to_check)
```

1. execute_rule

对于一条 rule(性质)，执行之并返回执行结果。其返回的结果 CHECK_RESULT 是一个结构体常量，如下为它的五种返回情况及其含义：

| | |
|--------------------------|---------------------|
| 1. PRECOND_NOT_SATISFIED | 前置条件不满足，一般由于页面不稳定引起 |
| 2. UI_NOT_FOUND | 找不到执行过程中某一步骤的控件 |
| 3. ASSERTION_FAILURE | 断言（后置条件）失败，找到疑似应用错误 |
| 4. UNKNOWN_EXCEPTION | 未知的错误 |
| 5. PASS | 断言（后置条件）成功，性质通过 |

其 python 风格的简化代码实现如下。

参数

- rules:List[Rule] : 性质交互场景列表

返回

- CHECK_RESULT: 执行结果

```
def execute_rule(rule, keaTest):

    if precondition_satisfied(rule) == False:
        return CHECK_RESULT.PRECON_NOT_SATISFIED
    try:
        execute(rule.function(keaTest))
    except UiObjectNotFoundError:
        return CHECK_RESULT.UI_NOT_FOUND
    except AssertionError:
        return CHECK_RESULT ASSERTION_FAILURE
    except Exception:
        return CHECK_RESULT.UNKNOWN_EXCEPTION
```

(continues on next page)

(continued from previous page)

```
return CHECK_RESULT.PASS
```

1. `get_rules_whose_preconditions_are_satisfied`

对于一组性质，检查他们的前置条件，并获取通过前置条件的性质。

其 python 风格的简化代码如下：

参数

- 无

返回

- `rules_passed_precondition:Dict[Rule, KeaTest]`: 通过了前置条件的性质列表

```
def get_rules_whose_preconditions_are_satisfied():

    for keaTestElements in KeaTest_DB:
        for target_rule in keaTestElements:
            if pass_precondition(target_rule) == True
                rules_passed_precondition.add(target_rule)

    return rules_passed_precondition
```

4. `get_rules_without_precondition`

对于一组性质，若他们的前置条件，并获取无前置条件的性质。

Note: 无前置条件的性质被视为无条件执行，等价于前置条件恒成立。

其 python 风格的简化代码如下：

参数

- 无

返回

- `rules_passed_precondition:Dict[Rule, KeaTest]`: 无前置条件的性质列表

```
def get_rules_without_preconditions(self):

    for eaTestElements in KeaTest_DB:
        for target_rule in eaTestElements.rules:
            if len(target_rule.preconditions) == 0:
                rules_without_precondition.add(target_rule)

    return rules_without_precondition
```

5. `execute_event_from_main_path`

对于给定的一个主路径步骤的源代码，尝试执行之。因为主路径中是用户利用 PDL 驱动编写的步骤，因此需要获取驱动对象，并让其执行相应操作。驱动储存在前述数据结构的 `pdl_driver` 中。

其 python 代码如下。

参数

- executable_script:str : 可执行的主路径步骤源代码

返回

- 无

```
def execute_event_from_main_path(self, executable_script):
    d = self._pdl_driver
    exec(executable_script)
```

3.3.7 DroidBot

DroidBot 是 kea 在应用探索阶段与安卓设备交互的类。主要提供生成事件，截图等方法。DroidBot 同时提供一个 UTG (UI Transition Graph, 事件迁移图)。可以基于事件迁移图，以基于模型的测试 (MBT, Model Based Testing) 编写更高级的应用探索策略。

Note: Droidbot 为本项目的参考项目。

<https://github.com/honeynet/droidbot>

Droidbot 项目架构

Droidbot 项目架构组成部分有：

1. App: 用于解析安卓应用安装包 (.hap) 或安卓包 (package)。提供 EntryActivity 等信息。
2. Device: 一个安卓设备的抽象，提供设备层面的一些操作接口，如发送文件，输入内容，旋转设备，获取前台应用等。
3. ADB: 安卓设备 adb 指令的抽象，提供通过 adb 与设备交互的接口，如 shell, pull_file 等。Device 依赖于本类。
4. uiautomator2: 安卓设备测试工具 uiautomator2，为 Droidbot 拓展提供输入事件等功能。
5. InputManager: 输入控制器，提供策略选择等功能。
6. InputPolicy: 输入策略，提供多种输入策略，如随机策略，大模型指引策略等，用于规定探索应用的规则。
7. EventLog: 事件日志的抽象，执行事件输入前，事件输入后的记录操作，以及发送事件的操作。
8. InputEvent: 输入事件的类，包含点击、长按、输入等事件。
9. UTG: 事件迁移图 (UI Transition Graph) 的类，用于应用建模，可以被输入策略使用进行更复杂的决策。
10. DeviceState: 用于应用界面抽象的类，对应用界面进行不同层次的抽象，提供给 UTG 类进行建模。

因为 InputManager, InputPolicy 与 Kea 的输入策略重点相关, 故另开章节进行介绍。uiautomator2 为参考的框架, 未做开发修改。其他功能基本沿用原来的 Droidbot。故在本节不介绍, 重要的类会在团队参考 Droidbot 开发的鸿蒙版 Droidbot: HMDroidbot 中介绍。

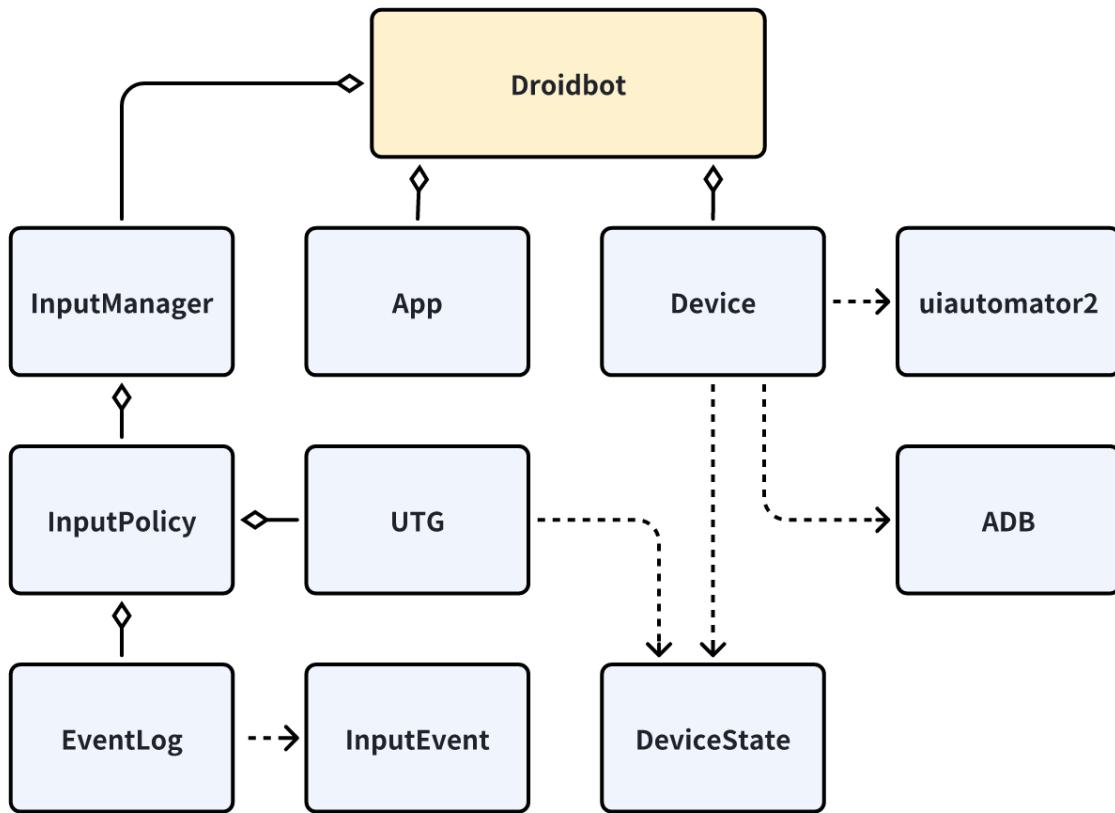


Fig. 7: Droidbot 架构图

3.3.8 HMDroidbot

HMDroidbot 是 kea 在应用探索阶段与安卓设备交互的类。主要提供生成事件，截图等方法。HMDroidbot 同时提供一个 UTG (UI Transition Graph, 事件迁移图)。可以基于事件迁移图，以基于模型的测试 (MBT, Model Based Testing) 编写更高级的应用探索策略。

Note: HMDroidbot 为本项目工作的一部份，为队伍中梁锡贤同学开发，开发过程参考项目为 droidbot，应计入本项目工作。

<https://github.com/ecnusse/HMDroidbot>

HMDroidbot 项目架构

HMDroidbot 项目架构组成部分有：

1. AppHM: 用于解析鸿蒙应用安装包 (.hap) 或鸿蒙包 (package)。提供 EntryAbility 等信息。
2. DeviceHM: 一个鸿蒙设备的抽象，提供设备层面的一些操作接口，如发送文件，输入内容，旋转设备，获取前台应用等。
3. HDC: 鸿蒙设备 hdc 指令的抽象，提供通过 hdc 与设备交互的接口，如 shell, pull_file 等。DeviceHM 依赖于本类。
4. Dumper: 鸿蒙设备获取界面布局的抽象类，HDC 依赖于该类
5. hmdriver2: 鸿蒙设备测试工具 hmdriver2，为 DeviceHM 拓展提供输入事件等功能。
6. InputManager: 输入控制器，提供策略选择等功能。
7. InputPolicy: 输入策略，提供多种输入策略，如随机策略，大模型指引策略等，用于规定探索应用的规则。
8. EventLog: 事件日志的抽象，执行事件输入前，事件输入后的记录操作，以及发送事件的操作。
9. InputEvent: 输入事件的类，包含点击、长按、输入等事件。
10. UTG: 事件迁移图 (UI Transition Graph) 的类，用于应用建模，可以被输入策略使用进行更复杂的决策。
11. DeviceState: 用于应用界面抽象的类，对应用界面进行不同层次的抽象，提供给 UTG 类进行建模。

因为 InputManager, InputPolicy 与 Kea 的输入策略重点相关，故另开章节进行介绍。hmdriver2 为参考的框架，未做开发修改。以下介绍除上述三个功能外的其他组成部分。其中 EventLog、InputEvent、UTG 基本复用 droidbot。

AppHM

AppHM 是一个鸿蒙应用的抽象，用于分析一个鸿蒙应用。对外提供对鸿蒙应用的分析方法，获取应用入口 (EntryAbility) 等信息。

AppHM 会根据传入的应用是一个安装包 (.hap) 或一个包名 (如 com.example.app)，选择对应的方法进行初始化。

```
def __init__(self):
    if app_name is a installation package file:
        self._hap_init()
    elif app_name is a package name:
        self._package_init()
```

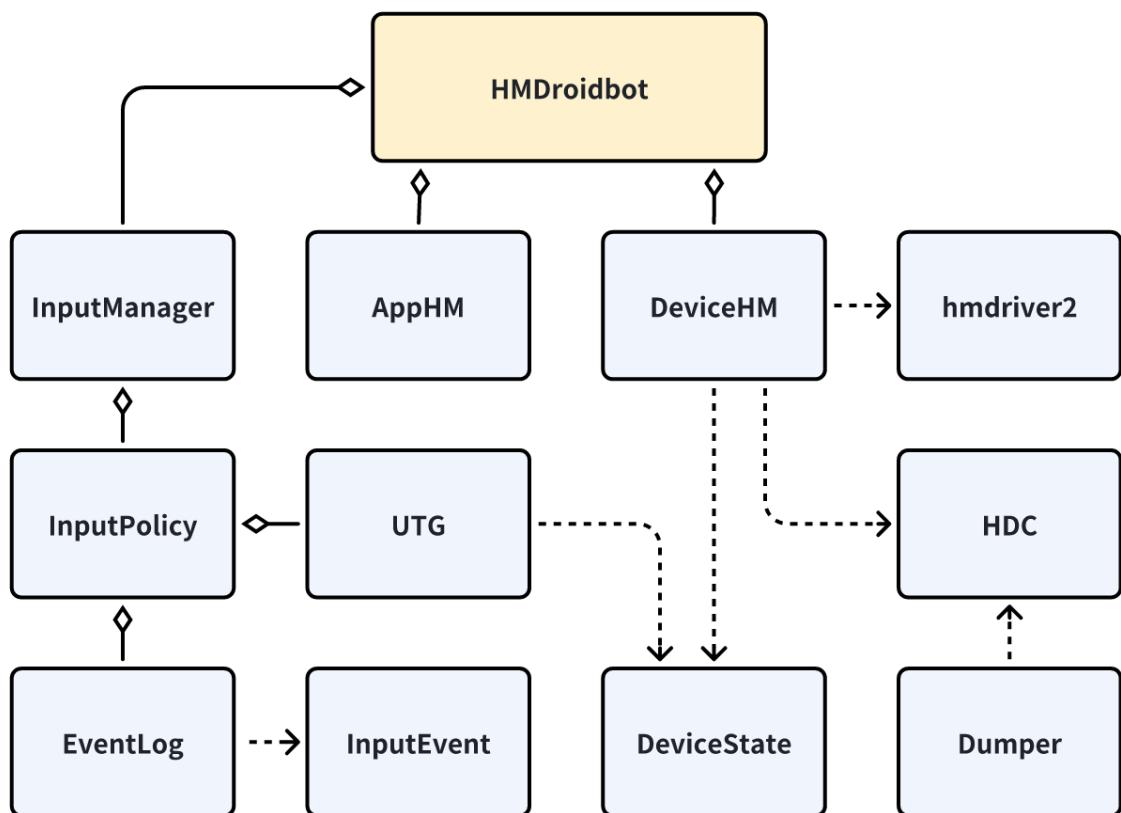


Fig. 8: HMDroidbot 项目架构图

以下是 AppHM 的方法和文档：

| 方法 | 输入 | 输出 | 简介 |
|----------------------|------------------|---------------------|------------------------|
| hap_init | app_path:str | | 给定一个安装包路径，通过安装包初始化 app |
| load_hap_info | pack_info:Dict | | 对安装包进行解压缩，读取其中信息 |
| package_init | package_name:str | | 通过包名初始化 app |
| dumpsys_package_info | | package_info:Dict | 从 dumpsys 指令获取包信息 |
| get_package_name | | package_name:str | 获取 app 包名 |
| get_start_intent | | start_intent:Intent | 获取启动应用的 Intent |
| get_stop_intent | | start_intent:Intent | 获取停止应用的 Intent |
| get_hashes | | hashes:List[str] | 获取应用安装包的哈希 |

DeviceHM

DeviceHM 是一个鸿蒙设备的抽象，提供设备层面的一些操作接口，如发送文件，输入内容，旋转设备，获取前台应用等。

以下是 DeviceHM 的方法和文档。

Note: 设备适配器 (adapters) 是终端与应用进行交互的适配器, DeviceHM 可以通过多种不同的工具与终端交互。如 HDC、hmdriver2 等。

| 方法 | 输入 | 输出 | 简介 |
|----------------------------|----------------------------------|---|--|
| check_connectivity | | | 检查设备和设备适配器是否可用 |
| set_up | | | 初始化设备和设备适配器 |
| connect | | | 连接设备和设备适配器 |
| disconnect | | | 断开设备和设备适配器 |
| is_foreground | app:AppHM | Boolean model_number:str device_name:str sdk_api:str display_info:Dict width:str height:str | 检查某个应用是否在设备中处于前台状态 获取设备的设备号 获取设备名称 获取设备的 SDK API 版本 获取设备的分辨率 获取设备横向分辨率 获取设备纵向分辨率 |
| model_number | | | |
| device_name | | | |
| get_release_version | | | |
| get_display_info | | | |
| get_width | | | |
| get_height | | | |
| unlock | | | 解锁设备 |
| send_intent | intent:Intent | | 向设备中发送一个 Intent |
| send_event | event:InputEvent | | 向设备发送一个事件 |
| start_app | app:AppHM | top_ability_name current_ability_stack | 启动一个应用 获取设备的栈顶 Ability 名称 获取设备的 Ability 栈 |
| get_top_activity_name | | | |
| get_current_activity_stack | | | |
| install_app | app:AppHM | | 安装一个应用 |
| uninstall_app | app:AppHM | | 卸载一个应用 |
| push_file | local_file, remote_dir | | 推送一个文件到设备上 |
| pull_file | remote_file, local_file | | 从设备拉取一个文件 |
| take_screenshot | | | 对设备进行截图 |
| get_current_state | action_count:int x:int, y:int | current_state | 获取当前设备的状态抽象 根据坐标执行点击操作 |
| view_touch | | | |
| | | | continues on next page |

Table 1 - continued from previous page

| 方法 | 输入 | 输出 | 简介 |
|------------------|----------------|----|--------------|
| view_long_touch | x, y, duration | | 根据坐标执行长按操作 |
| view_drag | start_xy | | 根据坐标执行拖动操作 |
| view_append_text | text:str | | 添加一个文本 |
| view_set_text | text:str | | 设置一个文本 |
| key_press | key_code | | 根据事件代码输入一个事件 |
| get_views | get_views | | 获取当前设备上的控件 |
| get_random_port | port:int | | 随机获取一个可用端口 |

HDC

鸿蒙设备 hdc 指令的抽象，提供通过 hdc 与设备交互的接口，如 shell, pull_file 等。DeviceHM 依赖于本类。

鸿蒙设备通过 Dumper 类获取应用的界面。Dumper 类是一个抽象类，共有两种实现：UitestDumper 和 HiDumper。

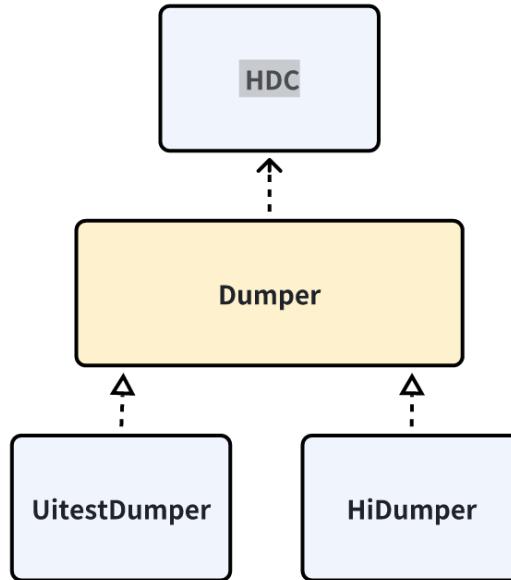


Fig. 9: Dumper 类与 HDC 类的关系示意

以下是 HDC 的方法和文档。

| 方法 | 输入 | 输出 | 简介 |
|----------------------------|-------------------------------------|------------------|----------------------|
| set_up | | | 初始化 HDC |
| run_cmd | | | 执行一个 hdc 命行命令 |
| shell | | | 执行一个 hdc shell 命令行命令 |
| connect | | | 连接 HDC |
| disconnect | | | 断开 HDC |
| check_connec | | | 检查 HDC 是否可用 |
| get_property property_name | | property | 通过 hdc 获取设备的属性 |
| get_model_nu | | model_number:str | 获取设备的型号 |
| get_sdk_vers | | sdk_version:str | 获取 SDK 版本 |
| get_device_n | | | 获取设备名称 |
| get_installe | | in- | 获取已安装的应用 |
| get_stalled_apps | | stalled_apps:Li | |
| get_display_ | | dpi:str | 获取设备显示的 dpi |
| unlock | | | 通过 hdc 解锁设备 |
| touch | x:int, y:int | | 通过 hdc 根据坐标执行点击操作 |
| long_touch | x, y, duration | | 通过 hdc 根据坐标执行长按操作 |
| drag | start_xy | | 通过 hdc 根据坐标执行拖动操作 |
| type | text:str | | 通过 hdc 添加一个文本 |
| press | key_code | | 通过 hdc 根据事件代码输入一个事件 |
| push_file | local_file:Path, remote_dir:Path | | 通过 hdc 推送一个文件到设备上 |
| pull_file | remote_file, lo- | cal_file | 通过 hdc 从设备拉取一个文件 |
| get_views | output_dir | | 获取当前页面的控件 |

Dumper 主要从设备中获取当前页面的布局，并转换为安卓风格使 droidbot 能使用。

以下是 UiTestDumper 的实现：

| 方法 | 输入 | 输出 | 简介 |
|------------------|-------------|-----------|--------------------------------|
| dump_view | | view_path | 通过 uitest 获取 layout json，并返回 |
| preprocess_views | views_path | | 处理 views，转换为可双向查询的树结构 |
| get_adb_view | raw_view:Di | | 处理 views，转换为安卓风格方便 droidbot 使用 |
| get_view | | views:Dic | 获取 views |

以下是 HiDumper 的实现：

| 方法 | 输入 | 输出 | 简介 |
|-------------------------------|---------------------------------|------------------|-----------------------------|
| get_focus_window | | fo- cus_windo | 获取当前前台窗口 |
| dump_target_window | fo- cus_window:int, fp:IO | | 拉取目标窗口的布局入文件 中 |
| dump_layout | fp:IO | | 根据 hidumper 的输出处 理布局为树结构 |
| adapt_hierachy “get_views” | | views:Dic | 处理布局为安卓风格 获取 views |

InputEvent

InputEvent 是一个抽象类，其他所有的事件实现在此抽象接口上。

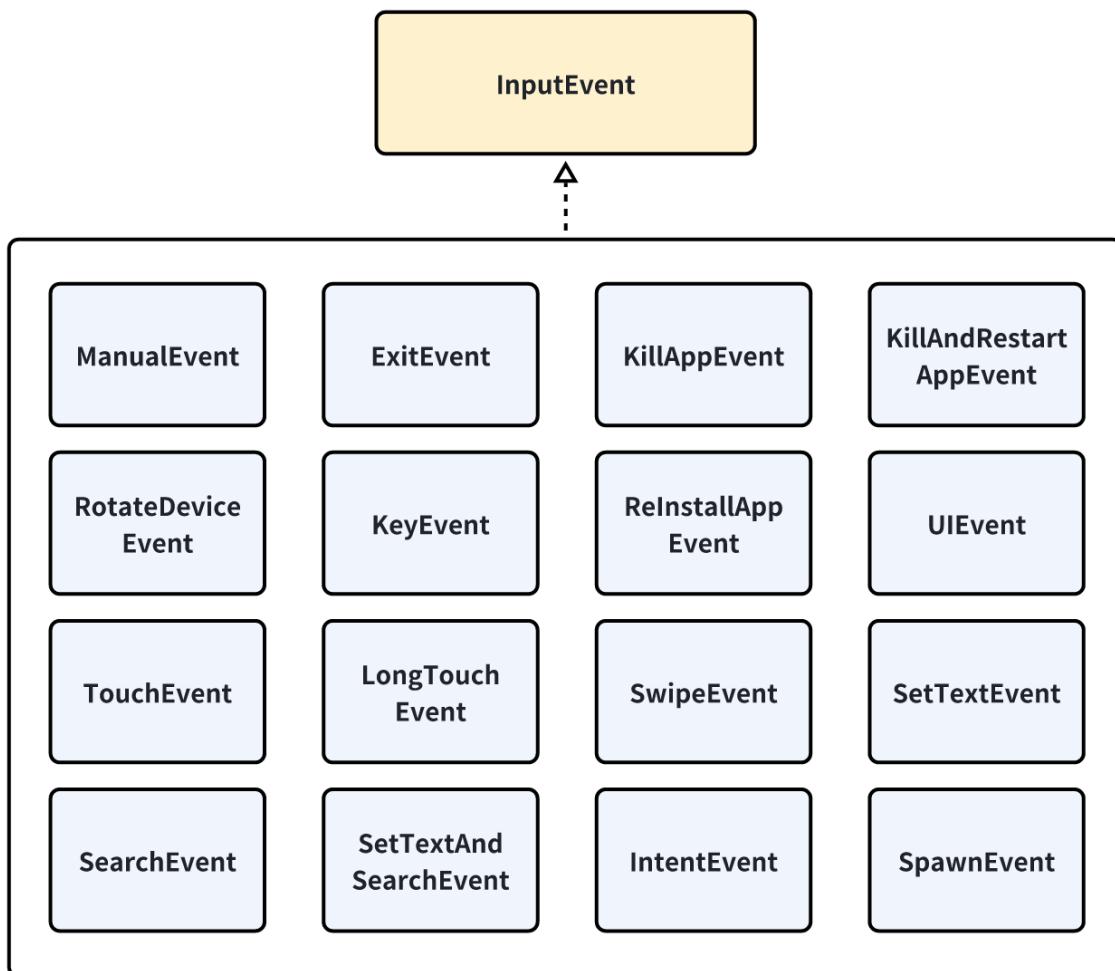


Fig. 10: InputEvent 接口和其实现

以下是 `InputEvent` 抽象类的方法：

| 方法 | 输入 | 输出 | 简介 |
|--------------|-------------------|------------------|-------------------------|
| send | device:HMDDevice | | 向设备发送事件 |
| to_dict | | event_dict:Dict | 以字典形式保存当前事件 |
| from_dict | event_dict:Dict | event:InputEvent | 从字典解析获取一个 InputEvent 实例 |
| get_event_st | state:DeviceState | event_str:str | 根据当前状态获取事件表示 |
| get_views | | views:List[str] | 获取当前事件对应的 views |

UTG

事件迁移图 (UI Transition Graph) 的类，用于应用建模，可以被输入策略使用进行更复杂的决策。

以下是 UTG 的方法和文档。

| 方法 | 输入 | 输出 | 简介 |
|--------------------------|--|-----------------------------------|-----------------------|
| first_state_str | | state_str:str | 第一个事件的状态哈希 |
| last_state_str | | state_str:str | 最后一个事件的状态哈希 |
| effective_event_count | | count:int | 造成迁移的事件数量 |
| nums_transitions | | count:int | 发现的状态迁移数量 |
| clear_graph | | | 清除 UTG |
| add_transition | event:InputEvent, old_state:str, new_state:str | | 添加一个迁移 |
| remove_transition | event:InputEvent, old_state:str, new_state:str | | 删除一个迁移 |
| add_node | state:DeviceState, event:InputEvent | | 添加一个状态节点 |
| is_event_explored | event:InputEvent, state:str | Boolean | 判断一个事件是否已经执行过 |
| is_state_reached | state:str | Boolean | 判断一个状态是否已经到达过 |
| get_reachable_states | current_state:str | reachable_states:List | 获取当前状态可迁移至的状态 |
| reachable_from_one_state | from_state:str, to_state:str | Boolean | 判断两个状态是否可迁移 |
| get_navigation_steps | from_state:str, to_state:str | List[Tuple[state: InputEvent]] | 获取从一个节点导航至另一个节点的步骤 |
| find_activity_acc | state:str | | 根据状态哈希获取状态对应的 Ability |

DeviceState

DeviceState 是用于应用界面抽象的类，对应用界面进行不同层次的抽象，提供给 UTG 类进行建模。

以下是 DeviceState 的方法和文档。

| 方法 | 输入 | 输出 | 简介 |
|---------------|--|---|----------------------------------|
| get_possible_ | | possi- ble_input:List[InputEven | 获取当前状态 上可执行的事件 |
| get_text_rep | | state_desc:str, tivity:str, dexed_views:List[str] | ac- in- 获取当前状态 的描述信息 |
| get_view_by_ | at- tribute_dict:Dict, ran- dom_select:Bool | 根据属性获取可用的控件 | |
| is_view_exis | view_dict:Dict | Boolean | 判断某个控件 是否存在 |
| get_view_des | view:Dict | view_desc:str | 获取控件的描 述 |
| assem- | root_view:Dict, ble_view_tree:views>List[Dict] | | 将 view 组织 为树结构 |
| get_view_str | view:Dict | view_str:str | 获 取 一 个 view 的描述 |
| get_pagePath | | pagePath:str | 获 取 当 前 界 面 对 应 的 pagePath |
| get_state_st | | state_str_raw:str | 获 取 当 前 页 面 的 状 态 描 述 |
| get_state_st | | state_str:str | 获 取 当 前 页 面 的 状 态 哈 希 |
| get_content_ | | content_free_str:str | 获 取 当 前 页 面 的 结 构 哈 希 |
| save_view_im | view_dict:Dict, output_dir:str | | 保 存 控 件 截 图 |

3.3.9 InputManager

本部分旨在解释 Kea 中的策略及输入控制器类 InputManager 的设计与实现。

功能设计与实现

InputManager 类是事件生成器的控制类，负责启动、停止事件的生成，并负责根据指定的输入策略生成和发送事件，支持随机探索策略、主路径引导策略和 LLM 策略。该类提供了灵活的事件管理机制，允许用户自定义事件生成策略，并能够根据应用的运行状态动态调整事件发送。InputManager 所包含的主要方法有：

- 获取当前测试用户所选择的探索策略。
- 添加事件到设备的执行事件列表等待执行。
- 使用当前探索策略开始生成事件进行测试。
- 停止生成事件，结束此次测试。

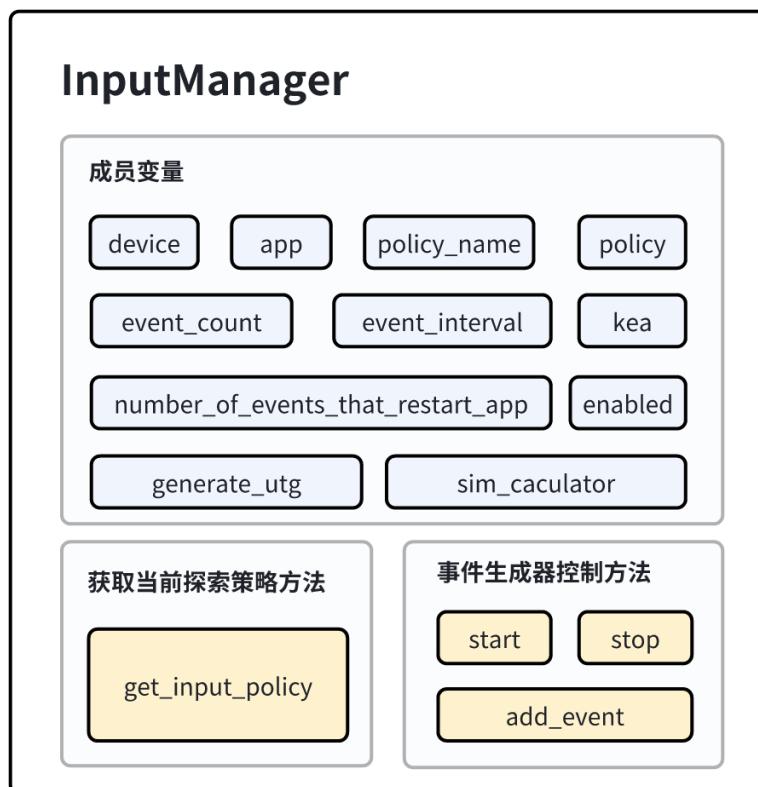


Fig. 11: InputManager 类的组成

Note: 为了便于读者理解，本文中提供的代码段简化版本仅对核心流程进行抽象并展示，实际代码与简化的参考代码不完全一致。

类属性

- DEFAULT_POLICY: 默认的输入策略名称。
- RANDOM_POLICY: 随机输入策略名称。
- DEFAULT_EVENT_INTERVAL: 默认事件间隔时间。
- DEFAULT_EVENT_COUNT: 默认生成事件的数量。
- DEFAULT_TIMEOUT: 默认超时时间。
- DEFAULT_DEVICE_SERIAL: 默认设备序列号。
- DEFAULT_UI_TARPIT_NUM: 默认 UI 陷阱数量。

InputManager 类中的数据结构

1. device

device 是 Device 的对象，用于记录当前测试的设备信息，便于后续对设备的交互操作。

2. app

app 是 App 的对象，用于记录当前所测试的移动应用的信息。

3. policy & policy_name

policy_name 是 string 类型，用于存储用户所选择的探索策略名。policy 是具体探索策略类的对象。

4. event_count & event_interval & number_of_events_that_restart_app

这三个成员变量均为整型。event_count 记录从测试开始到现在生成的事件个数；event_interval 记录了用户设置的两个事件之间停顿的时间；number_of_events_that_restart_app 为多少个事件后需要重启应用程序。

5. kea

kea 为 Kea 类的对象，用于生成事件过程中从 Kea 类中取出记录数据来完成对应用性质的测试。

6. enabled

enabled 为 bool 类型，用于记录当前事件生成器是否需要继续生成事件，默认值为 True。

7. generate_utg

generate_utg 为 bool 类型用于记录用户所设置的是否生成 UI 转移图的参数，便于生成事件的过程中判断是否应该生成 UI 转移图。

8. sim_caculator

sim_caculator 为 Similarity 的对象，用于计算上一个界面状态与当前界面状态之间的相似性。

InputManager 类中的成员方法

构造函数

`__init__` 方法用于初始化 `InputManager` 实例，设置事件发送的基本参数，并根据提供的策略名称初始化对应的输入策略。

参数

- `device`: `Device` 实例，表示目标设备。
- `app`: `App` 实例，表示目标应用。
- `policy_name`: 字符串，指定生成事件的策略名称。
- `random_input`: 布尔值，指示是否使用随机输入。
- `event_interval`: 事件间隔时间。
- `event_count`: 事件生成数量，默认为 ```DEFAULT_EVENT_COUNT```。
- `profiling_method`: 分析方法，用于性能分析。
- `kea`: `Kea` 实例，用于性质测试。
- `number_of_events_that_restart_app`: 重启应用的事件数量。
- `generate_utg`: 布尔值，指示是否生成 UTG。

核心流程

1. 初始化日志记录器。
2. 设置事件发送参数。
3. 根据策略名称初始化输入策略。
4. 设置相似度计算器。

获取探索策略的方法

1. `get_input_policy`

`get_input_policy` 方法根据用户所选择的 `policy_name` 来实例化对应的探索策略对象。实例化的对象存储在 `policy` 成员变量里。支持的策略包括：随机探索策略、主路径引导策略和 LLM 策略。

参数

- `device`: `Device` 实例。
- `app`: `App` 实例。

返回

- 本次测试使用的策略实例。

核心流程

1. 根据策略名称判断使用哪种输入策略。
2. 创建对应的输入策略实例。

```

def get_input_policy(self, device, app):
    if self.policy_name == POLICY_NONE:
        input_policy = None
    elif self.policy_name == POLICY_GUIDED:
        input_policy = GuidedPolicy(device, app, self.kea, self.
                                     generate_utg)
    elif self.policy_name == POLICY_RANDOM:
        input_policy = RandomPolicy(device, app, self.kea,_
                                     self.number_of_events_that_restart_app, True, self.generate_
                                     utg)
    elif self.policy_name == POLICY_LLM:
        input_policy = LLMPolicy(device, app, self.kea, self.
                                 number_of_events_that_restart_app, True, self.generate_utg)
    else:
        input_policy = None
    return input_policy

```

事件生成器的控制方法

1. start

start 方法用于启动所选定的探索策略。

核心流程

1. 记录开始发送事件的日志。
2. 根据输入策略开始发送事件。
3. 处理键盘中断，确保优雅退出。

```

def start(self):
    try:
        if self.policy is not None:
            self.policy.start(self)
    except KeyboardInterrupt:
        pass
    self.stop()

```

2. stop

stop 方法用于结束探索过程。

核心流程

1. 终止事件发送。
2. 清理事件发送相关的资源。
3. 记录停止发送事件的日志。

```

def stop(self):
    self.enabled = False

```

3. add_event

add_event 添加一个事件到事件列表，并将该事件发送给移动设备。

参数

- event: 要添加的事件，应为 AppEvent 的子类。

核心流程

1. 将事件添加到事件列表。
2. 创建事件日志记录器。
3. 根据事件间隔时间发送事件到设备。

```
def add_event(self, event):
    if event is None:
        return
    self.events.append(event)
    event_log = EventLog(self.device, self.app, event)
    event_log.start()
    while True:
        time.sleep(self.event_interval)
        if not self.device.pause_sending_event:
            break
    event_log.stop()
```

使用方法

InputManager 类的主要作用是控制事件生成器并管理应用运行期间的事件发送。用户可以通过构造函数初始化 InputManager 实例，并设置相应的参数，如测试设备、被测应用、策略名称等。然后，可以通过 start 方法启动事件生成器。通过 add_event 方法添加单个事件，并发送。通过 stop 方法停止生成事件。

3.3.10 RandomPolicy

RandomPolicy 类是随机事件生成策略的核心类。主要负责基于当前应用状态生成随机事件。该类提供了完整的随机事件生成策略的事件生成过程。RandomPolicy 所包含的主要方法有：

- 根据当前状态生成一个随机事件。
- 根据配置重启或重新安装应用。
- 在满足预条件的情况下，根据随机性决定是否检查性质。

随机事件生成策略的介绍

随机事件生成策略是一种简单有效的策略，它可以在没有明确指导路径的情况下探索应用的状态空间。具体来说，该策略会根据当前应用的状态随机生成事件，以期达到未探索的状态或触发应用中的某些性质。这种策略特别适用于那些没有明确测试路径或需要广泛覆盖应用状态的场景。

具体执行步骤如下：

步骤 1：检查是否满足生成事件的条件，即事件计数是否为首次生成事件或者上一个事件是否为应用重新安装事件。

步骤 2：如果满足条件，则运行初始化器并获取设备当前状态。

步骤 3：判断当前状态是否为空，如果是，则等待 5 秒并返回一个名称为“BACK”的键事件。

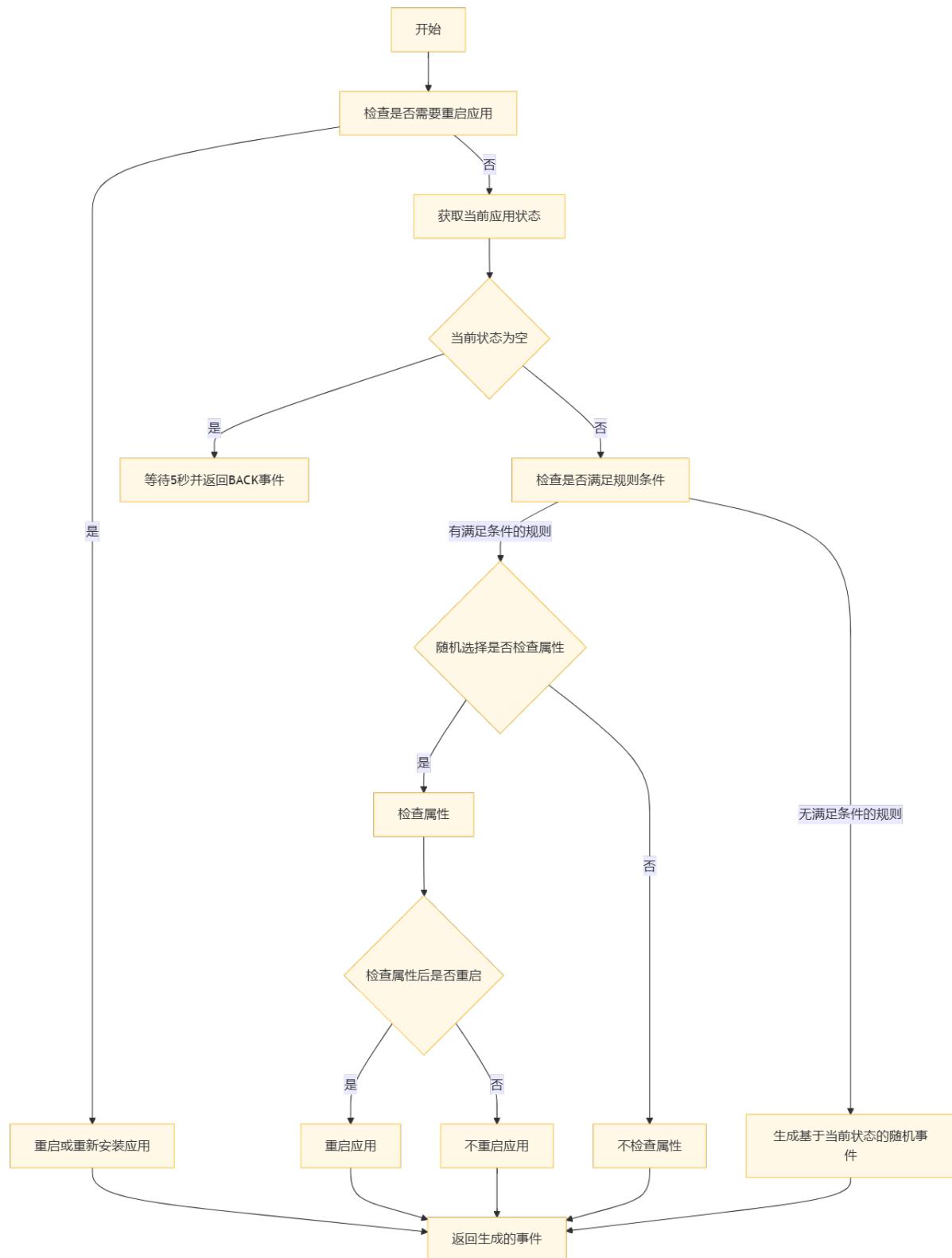


Fig. 12: 随机探索策略的流程图

步骤 4：检查事件计数是否是重启应用事件数量的倍数，如果是，则根据配置决定是清除并重新安装应用还是仅仅重启应用。

步骤 5：获取所有满足预条件的规则，如果存在这样的规则，则记录当前时间，并根据随机性决定是否检查性质。

步骤 6：如果决定检查性质，则执行性质检查。如果检查后需要重启应用，则记录日志并返回应用杀进程事件；否则，不重启应用。

步骤 7：如果因为随机性决定不检查性质，则记录日志并继续。

步骤 8：基于当前应用状态生成一个随机事件。这包括将应用移至前台（如果需要），获取当前状态可能的输入事件，并添加返回键和旋转设备事件。

步骤 9：从可能的事件列表中随机选择一个事件。如果选择的是旋转设备事件，则根据上次旋转事件的方向选择相反方向的旋转事件。

步骤 10：返回生成的随机事件，该事件将被用于与应用的交互。

随机事件生成策略的伪代码

Algorithm: *RandomEventGeneration*

Input: *None*

Output: *GeneratedEvent*

```
1 Function generate_event()
2     current_state ← get_current_state()
3     if current_state is None:
4         wait(5 seconds)
5         return KeyEvent(name="BACK")
6     if event_count % number_of_events_that_restart_app == 0:
7         if clear_and_reinstall_app:
8             return ReInstallAppEvent(app)
9         else:
10            return KillAndRestartAppEvent(app)
11     rules_to_check ← get_rules_whose_preconditions_are_satisfied()
12     if len(rules_to_check) > 0:
13         if random() < 0.5:
14             check_property()
15             if restart_app_after_check_property:
16                 return KillAppEvent(app)
17             return None
18     event ← generate_random_event_based_on_current_state()
19     return event
```

RandomPolicy 类中的数据结构

1. event_count

event_count 整型，记录了已经生成的事件数量。

2. number_of_events_that_restart_app

number_of_events_that_restart_app 整型，记录了在重启应用前需要生成的事件数量。

3. clear_and_reinstall_app

clear_and_reinstall_app 布尔型，指示是否在重启应用前清除并重新安装应用。

4. restart_app_after_check_property

restart_app_after_check_property 布尔型，指示在检查性质后是否重启应用。

RandomPolicy 类中的成员方法

生成随机事件的方法

generate_event

generate_event 方法用于生成一个随机事件。

参数

- 无

返回

- 生成的事件对象。

核心流程

1. 检查是否需要运行初始化器并获取当前应用状态。
2. 根据事件计数和设置决定是否重启应用或清除并重新安装应用。
3. 检查是否有满足前提条件的规则，并根据随机性决定是否检查性质。
4. 生成基于当前状态的随机事件。

```
def generate_event(self):
    current_state = self.from_state
    if current_state is None:
        time.sleep(5)
        return KeyEvent(name="BACK")
    if self.event_count % self.number_of_events_that_restart_
        ↪app == 0:
        if self.clear_and_reinstall_app:
            return ReInstallAppEvent(self.app)
            return KillAndRestartAppEvent(self.app)
            rules_to_check = self.kea.get_rules_whose_preconditions_
                ↪are_satisfied()
            if len(rules_to_check) > 0:
```

(continues on next page)

(continued from previous page)

```

if random.random() < 0.5:
    self.check_rule_whose_precondition_are_satisfied()
    if self.restart_app_after_check_property:
        return KillAppEvent(self.app)
    return None
event = self.generate_random_event_based_on_current_
↪state()
return event

```

生成随机事件的成员方法

`generate_random_event_based_on_current_state`

`generate_random_event_based_on_current_state` 方法用于基于当前状态生成一个随机事件。

参数

- 无

返回

- 生成的事件对象。

核心流程

- 获取当前应用状态。
- 如果需要，将应用移至前台。
- 获取当前状态可能的输入事件。
- 根据随机选择生成一个事件。

```

def generate_random_event_based_on_current_state(self):
    current_state = self.from_state
    event = self.move_the_app_to_foreground_if_needed(current_
↪state)
    if event is not None:
        return event
    possible_events = current_state.get_possible_input()
    possible_events.append(KeyEvent(name="BACK"))
    possible_events.append(RotateDevice())
    self._event_trace += EVENT_FLAG_EXPLORE
    event = random.choice(possible_events)
    return event

```

3.3.11 GuidedPolicy

`GuidedPolicy` 类是主路径引导探索策略的核心类。主要负责获取用户在定义性质时定义的主路径以及根据主路径引导探索策略生成输入事件。该类提供了完整的主路径引导探索策略的事件生成过程引导方法。`GuidedPolicy` 所包含的主要方法有：

- 开始一段测试前从用户定义的主路径中随机选择一条主路径。
- 根据当前状态是否在主路径上决定返回事件种类。
- 在主路径上获取下一个执行的事件。
- 离开主路径探索应用深层状态过程的主要调控。
- 从主路径上获取能引导应用回到主路径上的事件。

主路径引导策略的介绍

据观察，当用户指定应用程序性质时，用户通常沿着应用程序入口的主路径到达目标应用程序功能。这样的主要路径可以很容易地作为获得，并能够用于指导探索。具体来说，当从应用程序入口执行主路径时，我们可以获得一系列 GUI 状态 $S = [s_0, s_1, s_2, \dots, s_n]$ ，其中 $s_n \models P$ ，即 s_n 满足前置条件。此外，探索靠近主路径的状态可能会有更高的机会达到满足前提条件的 GUI 状态。本算法的核心执行流程如图。

具体执行步骤如下：

步骤 1：将主路径中的每个事件 e_i 发送到被测移动应用，得到主路径的状态序列 $S = [s_0, s_1, s_2, \dots, s_n]$ ，从而使被测移动应用到达满足前置条件 P 的状态 s_n ；

步骤 2：判断是否达到最大测试执行时间，若达到则结束测试，否则按照 $S = [s_0, s_1, s_2, \dots, s_n]$ 倒序探索应用程序。若 S 中的所有状态被探索完毕，则初始化被测移动应用；否则，按倒序关系取出一个未被探索的状态 s_i ，发送相应的前序事件序列 $[e_1, e_2, \dots, e_i]$ ，使得被测移动应用到达主路径状态 s_i ，继续步骤 3；

步骤 3：以状态 s_i 作为探索被测移动应用的起始状态，判断当前已执行事件个数是否达到最大执行事件个数，如没有达到，获取当前被测移动应用状态，继续步骤 4；如果达到，则执行 6；

步骤 4：判断当前被测移动应用状态是否满足一个或多个前置条件 P 。若没有任何前置条件满足或者不进行性质检测，则继续执行步骤 5；否则，执行交互场景 I 中定义的事件序列，判断后置条件 Q 是否满足，返回步骤 3；

步骤 5：使用外部用户界面交互工具分析当前界面状态获取可执行事件列表，从可以执行的事件列表中随机选择一项事件，并发送给被测移动应用执行，返回步骤 3；

步骤 6：判断当前状态能否转换为状态 s_n ，若能，则发送相应的事件到被测移动应用，使得移动应用的状态满足前置条件；

为了更清晰的展示该策略的上述过程，在此通过下图进一步详析这一过程。

令 s_0 为被测移动应用的起始探索状态，主路径为 $E = [e_1, e_2, e_3, e_4]$ ，性质为 $\phi = < P, I, Q >$ 。在第一次迭代中，该策略将发送 E 的所有事件并到达 s_4 并且 $s_4 \models P$ 。满足 P 的状态用灰色标记。然后本策略引导从 s_4 开始随机探索。假设它在 s_4 上生成 e_5 并达到 s_5 。假设 $s_5 \models P$ ，策略可能决定执行 I 获得结束状态 s_6 并在该状态下检查后置条件 Q 。假设 $s_6 \models P$ ，则说明没有发现性质错误。此时，假设 e_5 和 I 的执行事件数量超过预先限定的最大事件数量，策略将停止随机探索并尝试导航到满足 P 的主路径。假设 E 中的事件无法在 s_6 上发送，则该策略会放弃引导过程，并开始第二次迭代。在第二次和第三次

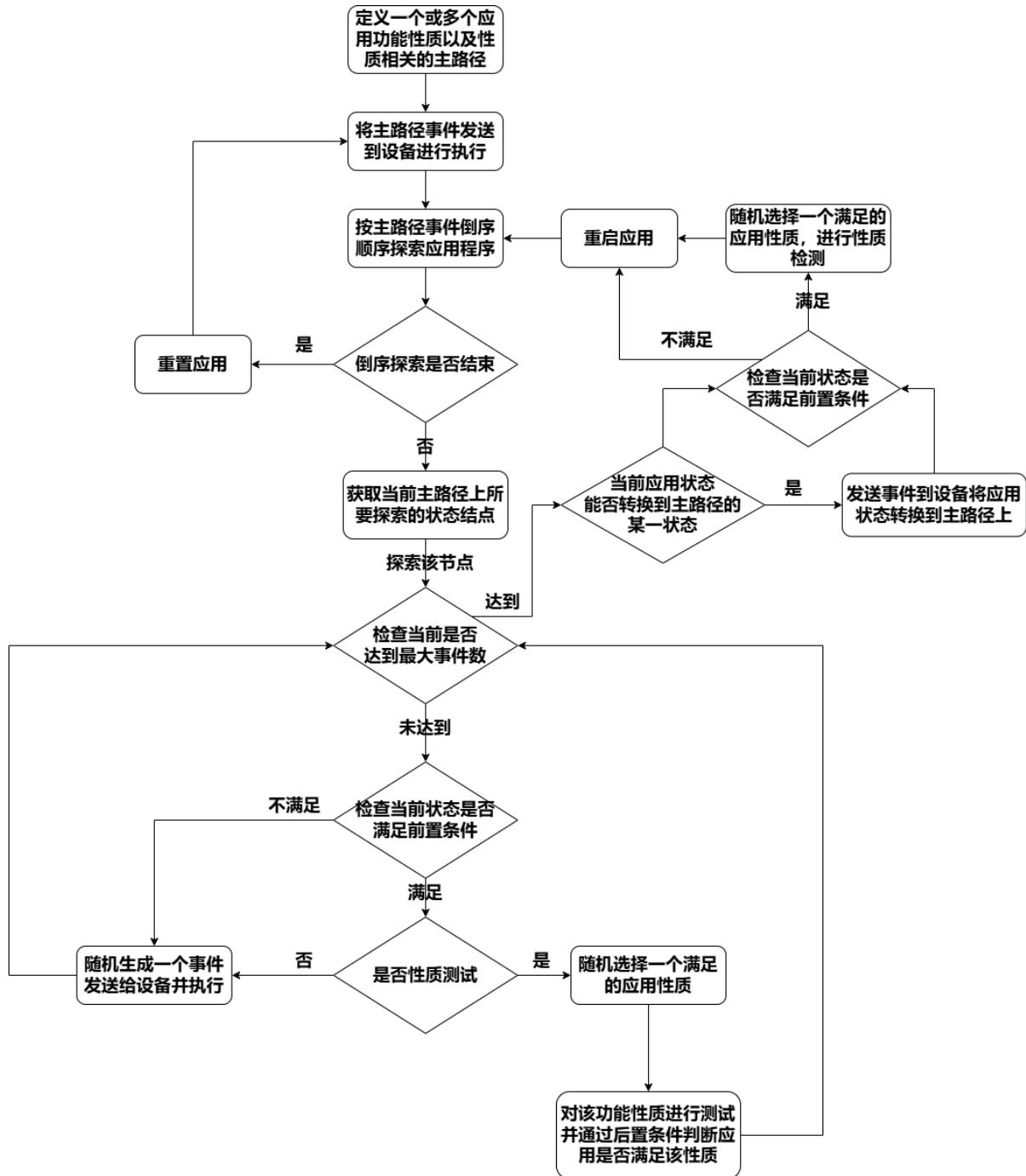


Fig. 13: 主路径引导探索策略的流程图

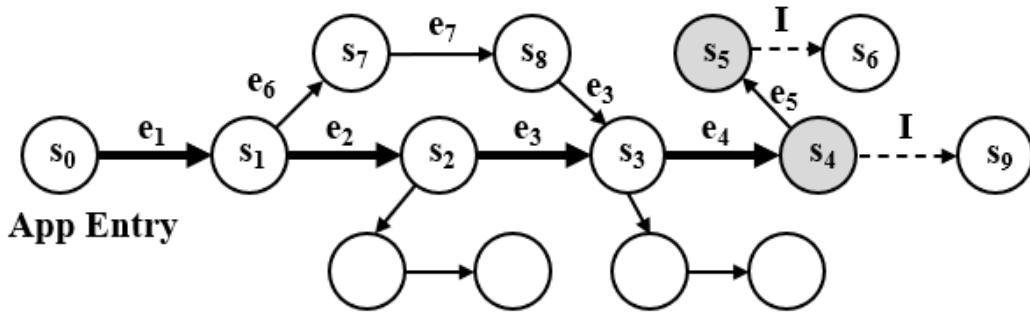


Fig. 14: 主路径引导探索策略的实施例示意图

迭代中，它将分别从 s_3 和 s_2 开始，并执行与第一次迭代类似的过程。在第 4 次迭代中，从 s_1 开始。假设过程中通过生成两个随机事件 e_6 和 e_7 来探索 s_1 , $e_6 \rightarrow s_7$, $e_7 \rightarrow s_8$ ，但 s_7 和 s_8 都不满足前置条件 P ，则本策略尝试导航回遵循满足 P 的主路径。假设过程中发现 e_3 可以在 s_8 上发送。它会依次发送 e_3 和 e_4 以尝试遵循主路径。假设最终到达满足 P 的 s_4 本策略将执行 I 并在结束状态 s_9 上检查后置条件 Q 。如果后置条件不成立，则表明发现一个功能性质错误。

步骤 7：检查当前状态是否满足前置条件，若满足则进行性质测试；

步骤 8：重启移动应用并返回步骤 2。

默认参数

- MAX_NUM_RESTARTS: 最大重启次数。
- MAX_NUM_STEPS_OUTSIDE: 应用外的最大步骤数。
- MAX_NUM_STEPS_OUTSIDE_KILL: 应用外最大步骤数（强制终止）。
- START_TO_GENERATE_EVENT_IN_POLICY: 策略中生成事件的起始时间。
- MAX_NUM_QUERY_LLM: 最大查询 LLM 的次数。
- EVENT_FLAG_STARTED: 事件开始标志。
- EVENT_FLAG_START_APP: 启动应用的事件标志。
- EVENT_FLAG_STOP_APP: 停止应用的事件标志。
- EVENT_FLAG_EXPLORE: 探索事件标志。
- EVENT_FLAG_NAVIGATE: 导航事件标志。
- EVENT_FLAG_TOUCH: 触摸事件标志。
- POLICY_GUIDED: 主路径引导策略名称。
- POLICY_RANDOM: 随机策略名称。
- POLICY_LLM: LLM 策略名称。

- POLICY_NONE: 无策略名称。

GuidedPolicy 类中的数据结构

1. main_path

main_path 是 MainPath 类的对象，是当前测试过程所选用的主路径。

2. execute_main_path

execute_main_path 为 bool 类型，用于记录当前界面状态是否在主路径上。用于根据不同情况生成不同事件。

3. current_index_on_main_path

current_index_on_main_path 整型，为当前所处主路径的事件节点编号。初始值为 0。

4. current_number_of_mutate_steps_on_single_node

current_number_of_mutate_steps_on_single_node 整型，记录了已经在当前状态节点上探索生成事件的次数。初始值为 0。

5. max_number_of_mutate_steps_on_single_node

max_number_of_mutate_steps_on_single_node 整型，记录了对单个状态节点最大探索生成事件的次数。初始值为 20。

6. number_of_events_that_try_to_find_event_on_main_path

number_of_events_that_try_to_find_event_on_main_path 整型，记录了当前尝试回到主路径的次数。初始值为 0。

7. index_on_main_path_after_mutation

index_on_main_path_after_mutation 整型，是对某个状态节点执行一系列事件探索后能够回到的主路径上的事件编号。初始值为 -1。

8. mutate_node_index_on_main_path

mutate_node_index_on_main_path 整型，是记录当前所探索的主路径上的事件节点编号。初始值为所选主路径的长度。

主路径引导策略的伪代码

Algorithm: MainPathGuidedExploration

Input: $\phi = \langle P, I, Q \rangle, E = [e_1, e_2, \dots, e_n]$

Output: BugReport

```

1 Function main ( $\Phi = \langle P, I, Q \rangle, E = [e_1, e_2, \dots, e_n]$ )
2   i  $\leftarrow n$ ;
3   while not timeout do
4     if i > 0 then
5       for e  $\leftarrow e_1$  to  $e_i$  do
6         sendEventToApp (e);
7       explore( $\Phi, E$ )
8       i  $\leftarrow i-1$ 

```

(continues on next page)

(continued from previous page)

```

9      if i = -1 then
10         cleanApp();
11         i ← n;
12         restartApp();
13 Function explore ( $\Phi = \langle P, I, Q \rangle$ ,  $E = [e_1, e_2, \dots, e_n]$ )
14     for t < 1 to MAX_STEP do
15         s ← getCurrentState();
16         if s ⊨ P ∧ random() < 0.5 then
17             checkProperty(I, Q) ;
18         else
19             e ← generateRandomEvent (s);
20             sendEventToApp(e);
21             ej ← canGoToSatisfyPrecondition(E);
22             if ej ≠ null then
23                 goToSatisfyPrecondition (ej, E);
24             s ← getCurrentState();
25             if s ⊨ P then
26                 checkProperty (I, Q) ;
27 Function canGoToSatisfyPrecondition ( $E = [e_1, e_2, \dots, e_n]$ ):
28     s ← getCurrentState();
29     for ej ← en to e1 do
30         if ej ← en to e1 then
31             return ej;
32         return null;
33 Function goToSatisfyPrecondition(ej,  $E = [e_1, e_2, \dots, e_n]$ ):
34     for e ← ej to en do
35         s ← getCurrentState();
36         if e.widget ∈ s then
37             sendEventToApp(e);

```

此策略采用输入一个性质 $\phi = \langle P, I, Q \rangle$ 和事件序列 $E = [e_1, e_2, \dots, e_n]$ 形式的主路径。该策略沿主路径向后遍历，并在接近主路径的范围内进行 UI 探索（第 3-12 行）。具体来说，它从 e_n 向后迭代到 e_1 （第 8 行）。对于 $0 < i \leq n$ 的每个事件 e_i ，它会将前缀 $[e_1, e_2, \dots, e_i]$ 发送到应用程序，以达到主路径的 GUI 状态 s_i （第 5-6 行）。接下来，它探索接近 s_i 的 GUI 状态，尝试找到满足前置条件 P 的状态（第 7 行）。请注意，在 s_1 之后，本系统还通过不从 E 发送任何事件来探索接近 s_0 的 GUI 状态（第 4 行）。如果时间预算允许，可以沿主路径进行多次遍历，我们在探索主路径上的每个状态（第 9-11 行）后清理应用程序数据。

从主路径的 GUI 状态开始的探索类似于前述随机探索策略（第 14-20 行）。具体来说，在每个 GUI 状态中，本算法检查是否满足某个前置条件 P （第 15-16 行）。如果有前置条件满足，本算法会有 50% 的概率来测试 s 处的性质（第 16 行）。否则，将生成随机事件并发送给设备以达到另一个状态（第 19-20 行）。上述过程迭代 $MAXSTEP$ 次（第 14 行）。

探索完之后，本算法尝试回到满足前提条件的状态（第 21-23 行）。因为从主路径的 GUI 状态开始的随机探索可能会改变应用程序的内部状态，在这种情况下，达到满足前置条件的状态可能会进一步表现出不同的行为。为此，本算法搜索可在当前 GUI 状态（第 28-32 行）发送的最接近事件 e_j 。如果 e_j 存在，我们尝试给设备发送发送 E 的后缀 $[e_j, e_j + 1, \dots, e_n]$ （第 34-37 行）。最后尝试再次测试应用性质（第 24-26 行）。

GuidedPolicy 类中的成员方法

Note: 为了便于读者理解，本文中提供的代码段简化版本仅对核心流程进行抽象并展示，实际代码与简化的参考代码不完全一致。

获取主路径的方法

1. select_main_path

select_main_path 从用户定义的所有主路径中随机选择一条作为本轮测试的引导路径。

```
def select_main_path(self):
    if len(self.kea.all_mainPaths) == 0:
        self.logger.error("No mainPath")
        return
    self.main_path = random.choice(self.kea.all_mainPaths)
    self.path_func, self.main_path = self.main_path.function, self.
    ↪main_path.path
    self.main_path_list = copy.deepcopy(self.main_path)
    self.max_number_of_events_that_try_to_find_event_on_main_path =_
    ↪min(10, len(self.main_path))
    self.mutate_node_index_on_main_path = len(self.main_path)
```

事件生成管理的成员方法

1. generate_event

根据当前应用状态判断应当生成事件的方法。如果在应用外则返回应用；如果在初始阶段则运行初始化函数；如果在执行主路径则返回主路径事件；如果在探索应用则返回继续探索所需事件。

参数

- 无

返回

- 一个生成的事件。

```
def generate_event(self):
    current_state = self.from_state
    event = self.move_the_app_to_foreground_if_needed(current_state)
    if event is not None:
        return event
    if (self.event_count == START_TO_GENERATE_EVENT_IN_POLICY and_
    ↪self.current_index_on_main_path == 0)
        or isinstance(self.last_event, ReInstallAppEvent):
            self.select_main_path()
            self.run_initializer()
            self.from_state = self.device.get_current_state()
    if self.execute_main_path:
```

(continues on next page)

(continued from previous page)

```

event_str = self.get_next_event_from_main_path()
if event_str:
    self.kea.execute_event_from_main_path(event_str)
    return None
if event is None:
    event = self.mutate_the_main_path()
return event

```

在主路径上获取事件的成员方法

1. `get_next_event_from_main_path`

获取主路径上应该执行的下一个事件。

参数

- 无

返回

- 主路径上的下一个事件。

```

def get_next_event_from_main_path(self):
    if self.current_index_on_main_path == self.mutate_node_
        ↪index_on_main_path:
        self.execute_main_path = False
        return None
    u2_event_str = self.main_path_list[self.current_index_
        ↪on_main_path]
    if u2_event_str is None:
        self.current_index_on_main_path += 1
        return self.get_next_event_from_main_path()
    self.current_index_on_main_path += 1
    return u2_event_str

```

应用状态探索过程的成员方法

1. `mutate_the_main_path`

根据当前事件执行次数判断是继续生成随机事件还是回到主路径上。

参数

- 无

返回

- 探索过后生成的事件。

```

def mutate_the_main_path(self):
    event = None
    self.current_number_of_mutate_steps_on_single_node += 1
    if (self.current_number_of_mutate_steps_on_single_node >= self.
        ↪max_number_of_mutate_steps_on_single_node):
        if (self.number_of_events_that_try_to_find_event_on_main_path
            (continues on next page)

```

(continued from previous page)

```

→<= self.max_number_of_events_that_try_to_find_event_on_main_path):
    self.number_of_events_that_try_to_find_event_on_main_path_
→+= 1
    if self.index_on_main_path_after_mutation == len(self.
→main_path_list):
        rules_to_check = (self.kea.get_rules_whose_
→preconditions_are_satisfied())
        if len(rules_to_check) > 0:
            t = self.time_recoder.get_time_duration()
            self.time_needed_to_satisfy_precondition.append(t)
            self.check_rule_whose_precondition_are_satisfied()
        return self.stop_mutation()
event_str = self.get_event_from_main_path()
try:
    self.kea.execute_event_from_main_path(event_str)
    return None
except Exception:
    return self.stop_mutation()
return self.stop_mutation()
self.index_on_main_path_after_mutation = -1
if len(self.kea.get_rules_whose_preconditions_are_satisfied()) >_
→0:
    self.check_rule_whose_precondition_are_satisfied()
    return None
event = self.generate_random_event_based_on_current_state()
return event

```

2. generate_random_event_based_on_current_state

生成随机事件探索应用。

参数

- 无

返回

- 随机生成的事件。

```

def generate_random_event_based_on_current_state(self):
    current_state = self.from_state
    event = self.move_the_app_to_foreground_if_
→needed(current_state)
    if event is not None:
        return event
    possible_events = current_state.get_possible_input()
    possible_events.append(KeyEvent(name="BACK"))
    possible_events.append(RotateDevice())
    self._event_trace += EVENT_FLAG_EXPLORE
    event = random.choice(possible_events)
    return event

```

3. stop_mutation

停止探索过程，并重置参数。

参数

- 无

返回

- 重启或者重装应用事件。

```
def stop_mutation(self):
    self.index_on_main_path_after_mutation = -1
    self.number_of_events_that_try_to_find_event_on_main_
    ↪path = 0
    self.execute_main_path = True
    self.current_number_of_mutate_steps_on_single_node = 0
    self.current_index_on_main_path = 0
    self.mutate_node_index_on_main_path -= 1
    if self.mutate_node_index_on_main_path == -1:
        self.mutate_node_index_on_main_path = len(self.
    ↪main_path)
    return ReInstallAppEvent(app=self.app)
    return KillAndRestartAppEvent(app=self.app)
```

从探索过程返回主路径的成员方法

1. `get_event_from_main_path`

根据当前是否已经回到主路径上，如果已经回到则执行完后续主路径事件，如果没回到主路径上则根据主路径事件序列倒叙尝试返回主路径。

参数

- 无

返回

- 主路径事件字符串。

```
def get_event_from_main_path(self):
    if self.index_on_main_path_after_mutation == -1:
        for i in range(len(self.main_path_list) - 1, -1, -1):
            event_str = self.main_path_list[i]
            ui_elements_dict = self.get_ui_element_dict(event_str)
            current_state = self.from_state
            view = current_state.get_view_by_attribute(ui_elements_
    ↪dict)
            if view is None:
                continue
            self.index_on_main_path_after_mutation = i + 1
            return event_str
    else:
        event_str = self.main_path_list[self.index_on_main_path_after_
    ↪mutation]
        ui_elements_dict = self.get_ui_element_dict(event_str)
        current_state = self.from_state
        view = current_state.get_view_by_attribute(ui_elements_dict)
        if view is None:
            return None
        self.index_on_main_path_after_mutation += 1
        return event_str
    return None
```

2. get_ui_element_dict

获取主路径上单个事件所操作的组件的相关信息。

参数

- ui_element_str: 组件信息字符串

返回

- 字典形式的该组件相关信息。

```
def get_ui_element_dict(self, ui_element_str: str) -> Dict[str, str]:
    start_index = ui_element_str.find("(") + 1
    end_index = ui_element_str.find(")", start_index)
    if start_index != -1 and end_index != -1:
        ui_element_str = ui_element_str[start_index:end_index]
    ui_elements = ui_element_str.split(",")
    ui_elements_dict = {}
    for ui_element in ui_elements:
        attribute_name, attribute_value = ui_element.split(
            "=")
        attribute_name = attribute_name.strip()
        attribute_value = attribute_value.strip()
        attribute_value = attribute_value.strip('\'')
        ui_elements_dict[attribute_name] = attribute_value
    return ui_elements_dict
```

3.3.12 LLMPolicy

LLMPolicy 类是使用 LLM (Large Language Model) 在检测到 UI 陷阱时生成输入事件的核心类。主要负责在应用状态空间中遇到难以探索的 UI 状态时，利用 LLM 生成输入事件以增强功能场景覆盖。该类提供了完整的 LLM 辅助事件生成策略的事件生成过程。LLMPolicy 所包含的主要方法有：

- 根据当前状态生成一个 LLM 辅助的随机事件。
- 根据配置重启或重新安装应用。
- 在满足预条件的情况下，根据随机性决定是否检查性质。

LLM 辅助事件生成策略的介绍

LLM 辅助事件生成策略是一种结合了大型语言模型的策略，它可以在应用的 GUI 测试中遇到难以通过传统随机策略探索的状态时，利用 LLM 生成更有效的输入事件。这种策略特别适用于那些需要深入探索应用状态空间或需要优化测试覆盖率的场景。

具体执行步骤如下：

- 步骤 1：开始执行 LLMPolicy 类的事件生成过程。
- 步骤 2：初始化 LLMPolicy 实例，设置日志记录器、动作历史记录等。
- 步骤 3：启动事件生成循环，直到输入管理器的事件计数结束或条件不再满足。
- 步骤 4：检查事件计数器是否小于输入管理器设定的事件计数。

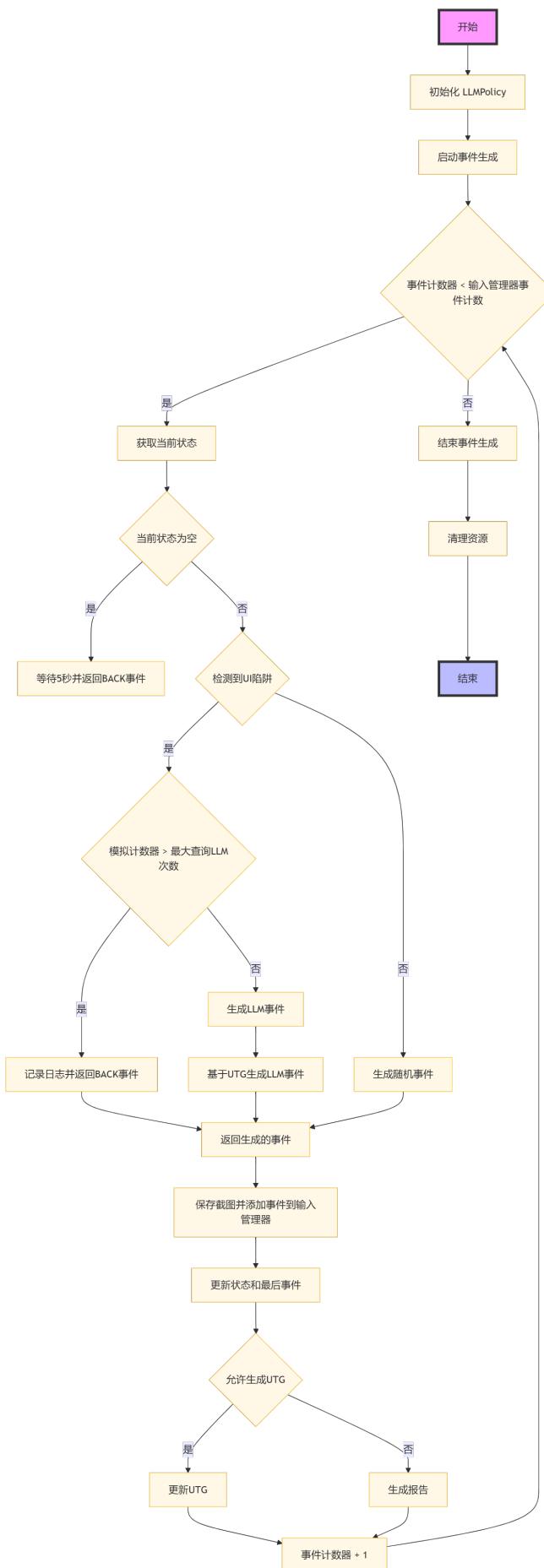


Fig. 15: LLM 辅助事件生成策略的流程图

- 步骤 5: 如果事件计数器小于输入管理器设定的事件计数, 获取当前应用状态。
- 步骤 6: 判断当前状态是否为空。
- 步骤 7: 如果当前状态为空, 等待 5 秒并返回一个名称为” BACK” 的键事件。
- 步骤 8: 如果当前状态不为空, 检查是否检测到 UI 陷阱。
- 步骤 9: 如果检测到 UI 陷阱, 检查模拟计数器是否超过了最大查询 LLM 次数。
- 步骤 10: 如果模拟计数器超过了最大查询 LLM 次数, 记录日志并返回一个名称为” BACK” 的键事件。
- 步骤 11: 如果模拟计数器未超过最大查询 LLM 次数, 生成一个 LLM 事件。
- 步骤 12: 如果未检测到 UI 陷阱, 生成一个随机事件。
- 步骤 13: 基于当前用户触发的事件 (UTG) 生成 LLM 事件。
- 步骤 14: 返回生成的事件, 该事件将被用于与应用的交互。
- 步骤 15: 保存截图并将生成的事件添加到输入管理器。
- 步骤 16: 更新当前状态和最后事件。
- 步骤 17: 检查是否允许生成 UTG。
- 步骤 18: 如果允许生成 UTG, 更新 UTG。
- 步骤 19: 生成报告, 包括所有状态和触发的 bug 信息。
- 步骤 20: 事件计数器加 1, 回到步骤 4 继续循环。
- 步骤 21: 如果事件计数器达到输入管理器设定的事件计数, 结束事件生成循环。
- 步骤 22: 清理资源, 结束 LLMPolicy 类的事件生成过程。

LLM 辅助事件生成策略的伪代码

Algorithm: *LLM – AssistedEventGeneration*

Input: *None*

Output: *BugReport*

```

1 Function LLM_Assisted_Event_Generation(policy_params)
2     Initialize policy with device, app, kea, and other parameters
3     Set event_count to 0
4
5     While input_manager is enabled and event_count < max_events
6         If device needs initialization
7             Perform device initialization
8
9         Determine current application state
10        If app is not running
11            Start the app
12        ElseIf app is in background
13            Bring app to foreground
14        Else
15            Generate LLM prompt based on current state and history
16            Query LLM for next action

```

(continues on next page)

(continued from previous page)

```

17     If LLM response indicates action
18         Perform action and update state
19     Else if LLM indicates text input
20         Get text from LLM and perform text input action
21     End If
22 End If

23
24     If event_count is a multiple of restart_threshold
25         Restart the app if necessary
26
27     Save screenshot and add event to input_manager
28     Increment event_count
29 End While

30
31     Generate final bug report if any issues found
32     Clean up policy resources
33 End Function

```

LLMPolicy 类中的数据结构

1. **event_count**

- event_count 整型，记录了已经生成的事件数量。

2. **number_of_events_that_restart_app**

- number_of_events_that_restart_app 整型，记录了在重启应用前需要生成的事件数量。

3. **clear_and_restart_app_data_after_100_events**

- clear_and_restart_app_data_after_100_events 布尔型，指示是否在 100 次事件后清除并重启应用数据。

4. **restart_app_after_check_property**

- restart_app_after_check_property 布尔型，指示在检查性质后是否重启应用。

5. **_action_history**

- _action_history 列表，记录了动作历史。

6. **_all_action_history**

- _all_action_history 集合，记录了所有动作历史记录。

7. **_activity_history**

- _activity_history 集合，记录了活动历史记录。

8. **from_state**

- from_state 对象，记录了起始状态。

9. **task**

- task 字符串，记录了 LLM 的任务描述。

LLMPolicy 类中的成员方法

启动事件生成的方法

start

start 方法用于启动事件生成过程。

参数

- input_manager: InputManager 的实例。

核心流程

1. 初始化事件计数器和输入管理器。
2. 循环生成事件直到达到输入管理器设定的事件计数或条件不再满足。
3. 根据当前状态和 LLM 的指导生成事件。
4. 将生成的事件添加到输入管理器中并更新设备状态。
5. 处理异常情况并在每次事件后增加事件计数器。

```

1  def start(self, input_manager):
2      self.event_count = 0
3      self.input_manager = input_manager
4      while self.event_count < input_manager.event_count:
5          event = self.generate_event()
6          self.input_manager.add_event(event)
7          self.event_count += 1
8
9  def generate_event(self):
10     if not self.from_state:
11         self.from_state = self.device.get_current_state()
12     if self.event_count == 0:
13         event = KillAppEvent(app=self.app)
14     elif self.event_count == 1:
15         event = IntentEvent(self.app.get_start_intent())
16     else:
17         event = (self.generate_llm_event()
18                  if input_manager.sim_calculator.detected_ui_
19                  ↳tarpit(input_manager)
20                  else self.generate_random_event_based_on_
21                  ↳current_state())
22     return event

```

生成 LLM 事件的方法

generate_llm_event

generate_llm_event 方法用于生成一个 LLM 辅助的事件。

参数

- 无

返回

- 生成的事件对象。

核心流程

1. 检查是否需要运行初始化器并获取当前应用状态。
2. 根据事件计数和设置决定是否重启应用或清除并重新安装应用。
3. 检查是否有满足前提条件的规则，并根据随机性决定是否检查性质。
4. 生成基于 LLM 的事件。

```

1  def generate_llm_event(self):
2      if self.event_count == START_TO_GENERATE_EVENT_IN_POLICY_-
→or isinstance(self.last_event, ReInstallAppEvent):
3          self.run_initializer()
4          self.from_state = self.device.get_current_state()
5      if not self.from_state:
6          time.sleep(5)
7          return KeyEvent(name="BACK")
8
9      if self.event_count % self.number_of_events_that_restart_-
→app == 0 and self.clear_and_reinstall_app:
10         return ReInstallAppEvent(self.app)
11
12     rules_to_check = self.kea.get_rules_whose_preconditions_
→are_satisfied()
13     if rules_to_check and random.random() < 0.5:
14         self.check_rule_whose_precondition_are_satisfied()
15     if self.restart_app_after_check_property:
16         return KillAppEvent(self.app)
17
18     event = self.generate_llm_event_based_on_utg()
19
20     if isinstance(event, RotateDevice):
21         event = RotateDeviceToLandscapeEvent() if self.last_-
→rotate_events == RotateDeviceToPortraitEvent() else_
→RotateDeviceToPortraitEvent()
22         self.last_rotate_events = event
23
24     return event

```

生成基于 UTG 的 LLM 事件的方法

`generate_llm_event_based_on_utg`

`generate_llm_event_based_on_utg` 方法用于基于当前 UTG 生成一个 LLM 辅助的事件。

参数

- 无

返回

- 生成的事件对象。

核心流程

1. 获取当前应用状态。
2. 如果应用不在活动堆栈中，尝试启动应用。
3. 如果应用在活动堆栈中但不在前台，尝试返回前台。
4. 如果应用在前台，根据 LLM 的指导选择下一步操作。

```

1  def generate_llm_event_based_on_utg(self):
2      current_state = self.from_state
3      if current_state.get_app_activity_depth(self.app) < 0:
4          start_app_intent = self.app.get_start_intent()
5          return IntentEvent(intent=start_app_intent) if not_
6          ↪self._event_trace.endswith(EVENT_FLAG_START_APP) else None
7
8      elif current_state.get_app_activity_depth(self.app) > 0_
9          ↪and self.__num_steps_outside > MAX_NUM_STEPS_OUTSIDE:
10         go_back_event = KeyEvent(name="BACK") if self.__num_
11         ↪steps_outside <= MAX_NUM_STEPS_OUTSIDE_KILL else_
12         ↪IntentEvent(self.app.get_stop_intent())
13         return go_back_event
14
15     action, _ = self._get_action_with_LLM(current_state,_
16     ↪self.__action_history, self.__activity_history)
17     return action if action else self.__random_explore_
18     ↪action()
19
20     def __random_explore_action(self):
21         if self.__random_explore:
22             return random.choice(self.__all_action_history)
23             # If couldn't find an exploration target, stop the app
24             stop_app_intent = self.app.get_stop_intent()
25             return IntentEvent(intent=stop_app_intent)

```

查询 LLM 的方法

`_query_llm`

`_query_llm` 方法用于向 LLM 查询以生成事件。

参数

- `prompt`: 提供给 LLM 的提示文本。
- `model_name`: 使用的 LLM 模型名称，默认为”gpt-3.5-turbo”。

返回

- LLM 的响应文本。

核心流程

1. 设置 LLM 客户端。
2. 发送提示文本到 LLM。
3. 接收并返回 LLM 的响应。

```

1 def _query_llm(self, prompt, model_name):
2
3     client = OpenAI()
4     response = client.chat.completions.create(messages=[{
5         "role": "user", "content": prompt}],
6         model=model_name, timeout=30)
7     return response.choices[0].message.content

```

获取动作与 LLM 交互的方法

`_get_action_with_LLM`

`_get_action_with_LLM` 方法用于获取基于 LLM 的下一个动作。

参数

- `current_state`: 当前应用状态。
- `action_history`: 动作历史记录。
- `activity_history`: 活动历史记录。

返回

- 选中的动作和候选动作列表。

核心流程

1. 构建包含任务描述、当前状态和历史记录的提示文本。
2. 向 LLM 查询并接收响应。
3. 解析响应以获取动作索引。
4. 根据索引选择动作并更新历史记录。

```

1 def _get_action_with_LLM(self, current_state, action_history,_
2                           activity_history):
3
4     prompt = self._build_prompt(current_state, action_
5                                  history, activity_history)
6     response = self._query_llm(prompt)
7     action_idx = self._parse_response(response)
8     return self._select_action(action_idx, current_state,_
9                                action_history, activity_history)

```

3.3.13 带状态的测试

本部分旨在解释 Kea 的带状态的测试是如何设计及实现的

功能说明与功能设计

`Bundle` 类是带状态测试的核心类。主要负责记录多组不同类型数据的相关状态和操作的组合，用于测试系统在不同状态下的行为。该类提供了完整的成员方法用于状态的增删改查操作。`Bundle` 所包含的主要方法有：

- 根据当前类型数据的状态情况判断是否需要新增状态（单例模式）。
- 对某类型数据状态的增删改查。
- 随机生成指定长度的状态文本值。
- 随机获取某类型数据的一个状态。

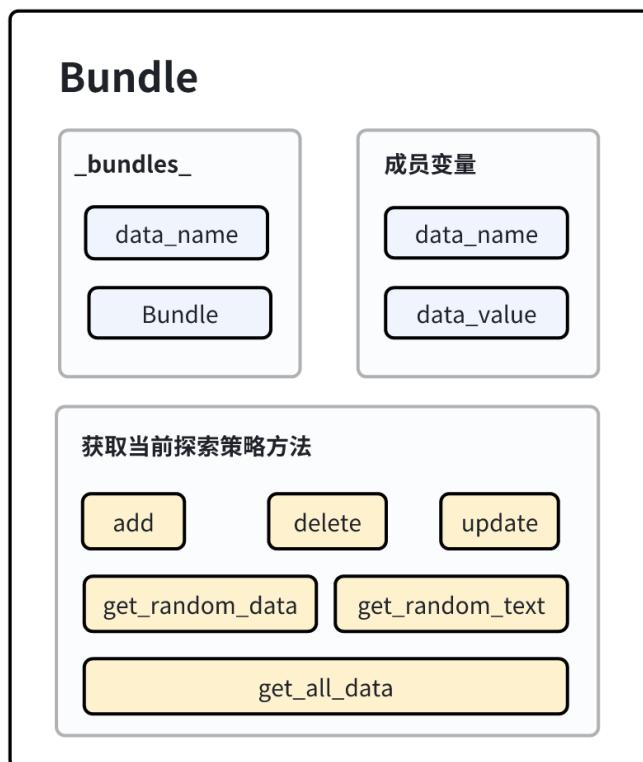


Fig. 16: `Bundle` 类的组成

Bundle 类中的数据结构的实现

1. `_bundle_`

`_bundle_` 为 `Bundle` 的类变量，是带状态测试中各种类型数据的状态记录库，用于存储应用状态便于对状态的操作。每个数据项以 `<string, Bundle>` 的键值对方式存储。

2. `data_name`

`data_name` 为 `string` 类型，存储了一个类型的数据的名称。

3. `data_value`

`data_value` 是 `list` 类型，存储了该类型的数据的所有状态。

Note: 为了便于读者理解，本文中提供的代码段简化版本仅对核心流程进行抽象并展示，实际代码与简化的参考代码不完全一致。

Bundle 类中功能方法的实现

单例模式方法

1. __new__

根据当前类型数据的名称判断是否已经实例化过该类型数据的 `Bundle` 对象，若没有实例化则实例化该类型 `Bundle` 对象并返回。否则，不实例化并返回之前实例化的对象。

参数

- `data_name`: 想要实例化的某个类型数据名称。

返回

- 该类型数据的实例。

```
def __new__(cls, data_name: str = None):
    if data_name in cls._bundles_:
        return cls._bundles_[data_name]
    else:
        instance = super().__new__(cls)
        cls._bundles_[data_name] = instance
    return instance
```

数据状态增删改查的成员方法

1. add

为当前类型数据增加状态。

参数

- `value`: 新增的状态值。

```
def add(self, value = None):
    if value is None:
        raise ValueError("the value of " + self.data_name + " "
                         "cannot be None")
    self.data_value.append(value)
```

2. delete

为当前类型数据删除状态。

参数

- `value`: 要删除的状态值。

```
def delete(self, value = None):
    if value is None:
        raise ValueError("the value of " + self.data_name + " cannot be None")
    self.data_value.remove(value)
```

3. update

为当前类型数据更新状态。

参数

- value: 要修改的旧状态值。
- new_value: 新的状态值。

```
def update(self, value = None, new_value = None):
    if new_value is None:
        raise ValueError("the new name of " + self.data_name + " cannot be None")
    if value is None:
        raise ValueError("the old name of " + self.data_name + " cannot be None")
    try:
        self.data_value.remove(value)
        self.data_value.append(new_value)
    except KeyError:
        print(f"'{value}' is not a object of Bundle.")
```

4. get_all_data

获取该类型数据的所有状态。

返回

1. 状态列表

```
def get_all_data(self):
    return self.data_value
```

随机生成状态文本的成员方法

1. get_random_text

随机生成指定长度的状态文本值。

参数

- value_max_len: 需要状态文本值的最大长度。

返回

1. 合法的状态文本值

```
def get_random_text(self, value_max_len = 10):
    text = st.text(alphabet=string.ascii_letters, min_size=1, max_size=value_max_len).example()
    return text
```

随机获取一个状态的成员方法

1. `get_random_data`

随机获取某类型数据的一个状态。

返回

1. 该类型数据的一个状态值

```
def get_random_data(self):
    random_item = random.choice(self.data_value)
    return random_item
```


实验与结果

4.1 Kea 工具应用效果简介

Kea 在 8 个开源应用中找到了 25 个此前未知的功能缺陷。下面是这些功能缺陷清单：

- **OmniNotes:**

```
#942 https://github.com/federicoiosue/Omni-Notes/issues/942
#946 https://github.com/federicoiosue/Omni-Notes/issues/946
#948 https://github.com/federicoiosue/Omni-Notes/issues/948
#949 https://github.com/federicoiosue/Omni-Notes/issues/949
#950 https://github.com/federicoiosue/Omni-Notes/issues/950
#951 https://github.com/federicoiosue/Omni-Notes/issues/951
#954 https://github.com/federicoiosue/Omni-Notes/issues/954
#956 https://github.com/federicoiosue/Omni-Notes/issues/956
#981 https://github.com/federicoiosue/Omni-Notes/issues/981
#937 https://github.com/federicoiosue/Omni-Notes/issues/937
#938 https://github.com/federicoiosue/Omni-Notes/issues/938
#939 https://github.com/federicoiosue/Omni-Notes/issues/937
#940 https://github.com/federicoiosue/Omni-Notes/issues/940
#941 https://github.com/federicoiosue/Omni-Notes/issues/941
#945 https://github.com/federicoiosue/Omni-Notes/issues/945
```

- **Markor:**

```
#2153 https://github.com/gsantner/markor/issues/2153
#2196 https://github.com/gsantner/markor/issues/2196
#2197 https://github.com/gsantner/markor/issues/2197
#2198 https://github.com/gsantner/markor/issues/2198
#2199 https://github.com/gsantner/markor/issues/2199
#2250 https://github.com/gsantner/markor/issues/2250
```

- **AmazeFileManager:**
#3991 <https://github.com/TeamAmaze/AmazeFileManager/issues/3991>
#4016 <https://github.com/TeamAmaze/AmazeFileManager/issues/4016>
#4130 <https://github.com/TeamAmaze/AmazeFileManager/issues/4130>
- **AnkiDroid:**
#15993 <https://github.com/ankidroid/Anki-Android/issues/15993>
#15995 <https://github.com/ankidroid/Anki-Android/issues/15995>
- **transistor:**
#488 <https://codeberg.org/y20k/transistor/issues/488>
#489 <https://codeberg.org/y20k/transistor/issues/489>
#495 <https://codeberg.org/y20k/transistor/issues/495>
- **Simpletask:**
#1230 <https://github.com/mpcjanssen/simpletask-android/issues/1230>

4.2 静态代码分析

- 使用 Flake8 对 Kea 的代码进行静态分析，检查错误。
- 使用 black 对 Kea 的代码进行重构，以确保代码符合 PEP8 标准。

4.3 动态代码测试

我们使用 `python coverage` 运行 `kea` 进行 覆盖率分析。

其结果如下

| File | statements | missing | excluded | coverage ▼ |
|--|-------------|-------------|----------|------------|
| example/example_hm_property.py | 7 | 0 | 0 | 100% |
| kea/__init__.py | 2 | 0 | 0 | 100% |
| kea/start.py | 102 | 2 | 0 | 98% |
| example/stateful_testing_example_property.py | 66 | 2 | 0 | 97% |
| example/example_property.py | 15 | 1 | 0 | 93% |
| kea/android_pdl_driver.py | 44 | 3 | 0 | 93% |
| kea/kea_test.py | 50 | 7 | 0 | 86% |
| kea/Bundle.py | 44 | 7 | 0 | 84% |
| kea/kea.py | 202 | 33 | 0 | 84% |
| kea/input_manager.py | 77 | 14 | 0 | 82% |
| kea/droidbot.py | 152 | 34 | 0 | 78% |
| kea/app_hm.py | 96 | 23 | 0 | 76% |
| kea/device_hm.py | 393 | 94 | 0 | 76% |
| kea/app.py | 102 | 27 | 0 | 74% |
| kea/harmonyos_pdl_driver.py | 42 | 11 | 0 | 74% |
| kea/device_state.py | 569 | 179 | 0 | 69% |
| example/example_mainpath_property.py | 30 | 10 | 0 | 67% |
| kea/intent.py | 87 | 29 | 0 | 67% |
| example/advanced_example_property.py | 49 | 17 | 0 | 65% |
| kea/device.py | 647 | 231 | 0 | 64% |
| kea/input_event.py | 652 | 232 | 0 | 64% |
| kea/input_policy.py | 568 | 207 | 0 | 64% |
| Total | 3996 | 1163 | 0 | 71% |

Fig. 1: kea 覆盖率分析整体数据