项目文档

191250106　倪梦雅

目录

1. 项目简介

当为 Android 应用程序生成 GUI 测试时，通常是一个独立的测试计算机，生成交互，然后在实际的 Android 设备上执行。这种方法虽然在应用程序和交互快速执行方面是有效的，但测试计算机和设备间的通信开销大大降低了测试的速度。

该项目（DD-2）在一个用于 Android 的测试生成器（DM-2）的基础上进行改进，通过 Android 的 Accessibility Service 服务访问待测试应用并与之交互。因为直接在测试的设备上运行，所以大大提高了测试生成器和被测应用之间的通信效率，同时简化了测试设置。

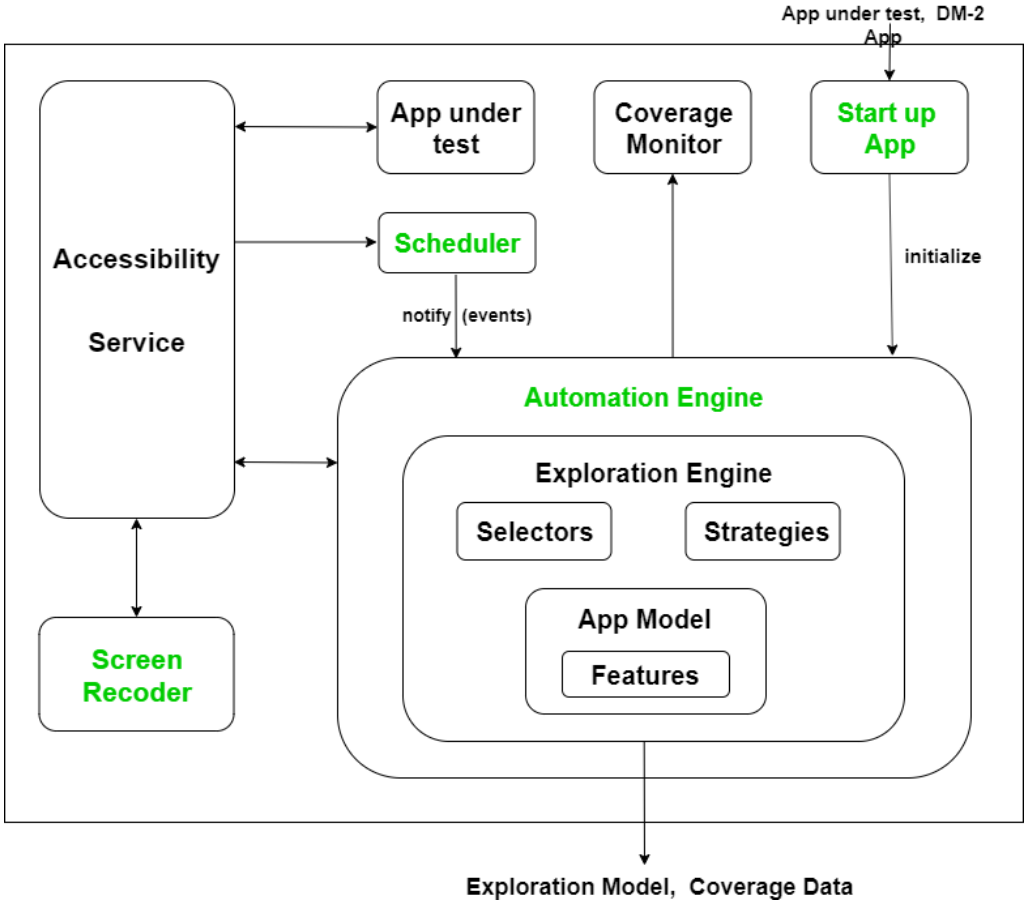

图 1.DD-2 的整体结构（绿色部分为 DD-2 的核心组件）

2. 功能模块

2.1 Start up App

start up App 用来请求必要的权限，并让用户选择一个应用程序进行测试。出于安全原因，Android 不允许后台服务在没有事先请求用户许可的情况下直接录制屏幕。

如下图，Start up App 激活 Screen Recorder,并将设备设置转发给用户，用户必须激活服务（Automation Engine）。

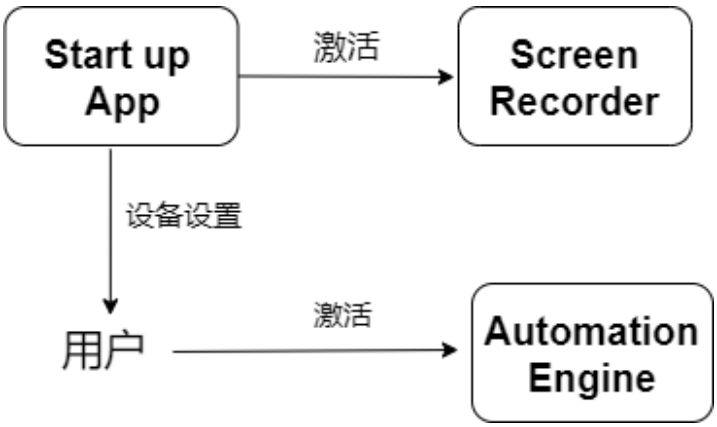

图 2.start up app 与其他功能模块的交互

2.2 Exploration Engine

Exploration Engine 通过一个探测循环来确定每个执行的设备动作后接下来应该做出什么交互。

DD-2 基于 App Model 的当前状态，通过 Selectors 和 Strategies 扩展这一机制。Selectors 根据当前和以前的 model 状态以编程方式选定激活哪个特定的 strategy，而 Strategies 根据 App model 当前的状态来决定下一步应该采取怎样的 Exploration。

在每个设备都操作后，DD-2 获取设备屏幕截图与其他组件信息，App model 中的状态会相应地更新，然后新的状态在下一次迭代中作为 app 的当前状态使用。循环一直持续到某个 strategy 触发 terminate，终止程序。

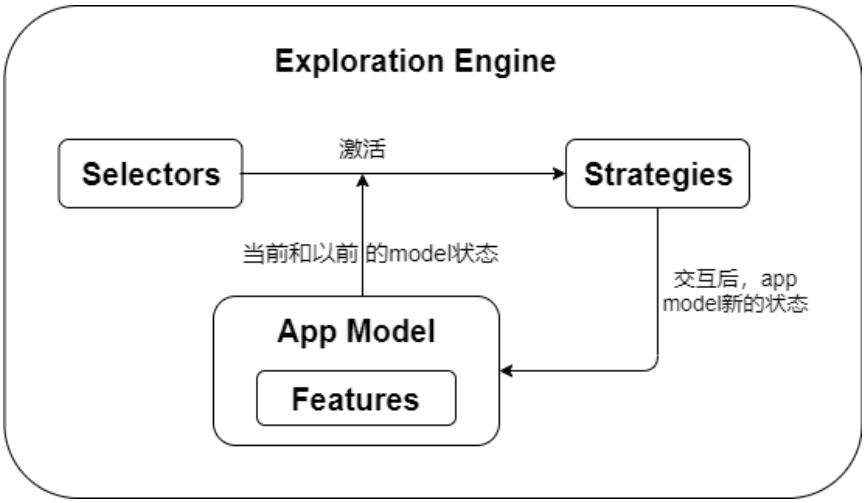

图 3. Exploration Engine 各组件的交互

2.2.1 Selectors

DD-2 支持定义标准，以确定不同情况下的最佳策略。这些标准就是 Selectors。

形式上 Selectors 被定义为一对 p,f(c)——>s 。
p 是其优先级，f(c)是一个映射函数 f,从一个模型上下文 c（如当前 状态或行动跟踪）获得的勘探策略。s 是根据 p、f(c)返回的最佳 strategy，用于派生下一个设备交互。
DD-2 提供一组 selectors 来激活默认 strategies，要使用的 selectors 集可以通过命令行或在配置文件中配置

```
object UiSelector {
    @JvmStatic
    val isWebView: SelectorCondition =
        { it, _ -> it.packageName == "android.webkit.WebView" || it.className == "android.webkit.WebView" }

    @JvmStatic
    val permissionRequest: SelectorCondition =
        { node, _ -> node.viewIdResourceName == "com.android.packageinstaller:id/permission_allow_button" }

    @JvmStatic
    val ignoreSystemElem: SelectorCondition =
        { node, _ ->
            node.viewIdResourceName?.let { !it.startsWith( prefix: "com.android.systemui") } ?: false
        }
```

图 4.Selectors 在工具中的实现的部分截图

2.2.2 Strategies

Strategies 根据 App model 当前的状态来决定下一步应该采取怎样的 Exploration。
与 app 的交互可以很简单，比如点击坐标（x,y）

```
object Gestures {
    /**
     * 创建单击手势的描述
     *
     * @param x 要单击的x坐标，不能为负
     * @param y 要单击的y坐标，不能为负
     *
     * @return 单击（x,y）的描述
     */
    fun createClick(x: Int, y: Int, duration: Long = 100): GestureDescription {
        val clickPath = Path()
        clickPath.moveTo(x.toFloat(), y.toFloat())
        clickPath.lineTo( x: x.toFloat() + 1, y.toFloat())
        val builder = GestureDescription.Builder()
        return builder
            .addStroke(StrokeDescription(clickPath,  startTime: 0, duration))
            .build()
    }
```

图 5.工具中单击坐标（x,y）相关内容

也可以很复杂，如启用设备 Wi-fi,蓝牙等，这种复杂的交互作用被抽象为 exploration 行动，探索操作确定自动化引擎应该执行哪些特定的设备操作顺序。

```
//启动wifi
fun enableWifi() {
    val wfm = context.getSystemService(Context.WIFI_SERVICE) as WifiManager
    val success = wfm.setWifiEnabled(true)

    if (!success) {
        log.warn("Failed to ensure WiFi is enabled!")
    }
}
```

图 6.工具中启用 wi-fi 相关内容

DM-2 提供一些默认策略:
触发 home 按钮
Reset：重置启用 Wi-fi
Back：按下设备后退按钮
Terminate：关闭应用程序并完成探索
BiaseRandom：从当前屏幕中随机选择一个 UI 元素，点击或长按

```
//返回主页
fun pressHome() {
    service.performGlobalAction(AccessibilityService.GLOBAL_ACTION_HOME)
}
//后退一步
fun pressBack() {
    service.performGlobalAction(AccessibilityService.GLOBAL_ACTION_BACK)
}
//确认
fun pressEnter() {
    sendKeyEvent(KeyEvent.KEYCODE_ENTER)
}
//启动wifi
fun enableWifi() {
    val wfm = context.getSystemService(Context.WIFI_SERVICE) as WifiManager
    val success = wfm.setWifiEnabled(true)

    if (!success) {
        log.warn("Failed to ensure WiFi is enabled!")
    }
}
//终止
fun terminate() {
    canceled = true
    targetPackage = ""
}
```

图 7.默认策略的相关内容

2.2.3 App Model

在 Exploration 过程中构建的 App Model 由 UI 状态组成。
工具通过计算 id wld 的 UUID（来自 Java 默认库）形式唯一标识一个 UI 元素。唯一 id 的度量可以有效地重新识别概念上相同的 UI 状态，以及在不同 UI 状态中再次出现 UI 元素（如菜单或帮助按钮），而不依赖它们的位置或布局。

2.3 Screen Recorder

Screen Recorder 为 Automation Engine 提供可视化屏幕，用于更准确的状态识别。它通过官方媒体投影 API14 来创建一个虚拟显示器并请求操作系统将主显示器的内容 clone 到新的虚拟显示器中工作。

```
//创建一个虚拟显示器并请求操作系统将主显示器的内容clone到新的虚拟显示器中工作
//为Automation Engine提供可视化屏幕
class ScreenRecorder private constructor(
    private val context: Context,
    private val mediaProjectionIntent: Intent,
    private val imgQuality: Int = 10
) : HandlerThread( name: "screenRecorderThread"), IScreenshotEngine, CoroutineScope {
```

图 8. Screen Recorder 在工具中相关内容

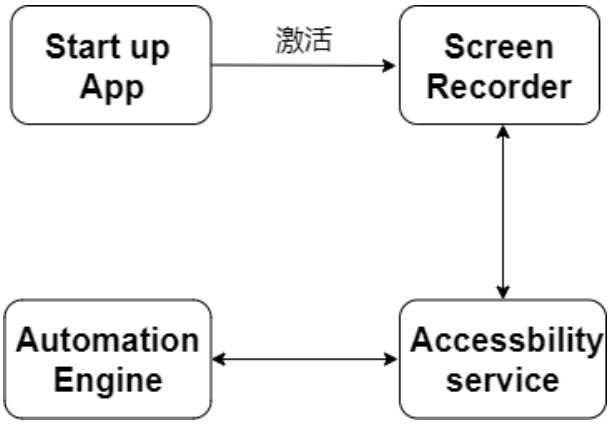

图 9. Screen Recorder 的相关交互

2.4 Scheduler

Scheduler 负责确定 AuT 的 UI 何时是稳定的，何时可以与之交互。它连接到 Accessibility service，并将所有与 AUT 的 UI 内容相关的传入 accessibility 事件排队。

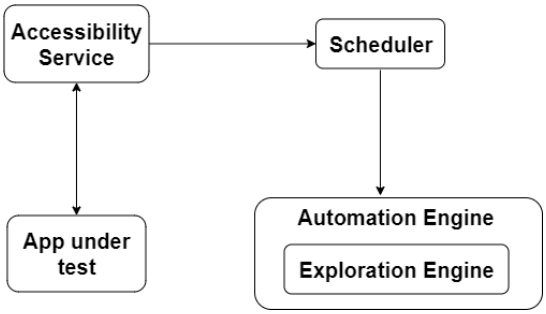

图 10.Scheduler 与其他模块的交互

Scheduler 有两个触发器来确定 UI 是否稳定。第一个是没有新事件的间隔，无论 AuT 中发生什么，它都会创建一个 accessibility 事件，Scheduler 会侦听

该事件，如果 Scheduler 指定时间内没有收到任何事件，假定 AuT UI 不变并通知 Automation Engine。

```
//第一个trigger
private suspend fun runScheduledTask(delayInterval: Long) {
    log.trace("Scheduling task with interval of $delayInterval ms")
    while (!isCanceled) {
        // 决定应该等多久
        val delayTime = delayInterval - (SystemClock.uptimeMillis() - lastTimestamp.get())
        log.trace("Delaying scheduled task for $delayTime ms")
        // 等待
        delay(delayTime)

        mutex.withLock {
            log.trace("(Locked) Checking if scheduled task must run")
            // 检查预期的时间戳
            val expectedTimestamp = lastTimestamp.get() + delayTime

            // 如果我们在预期的时间戳之后，通知idle
            val currentTime = SystemClock.uptimeMillis()
            if (currentTime > expectedTimestamp) {
                log.trace("(Locked) Scheduled task must run, notifying channel")
                // lastTimestamp.set(currentTime)
                idleNotificationChannel.send(currentTime)
            }
        }
    }
}
```

图 11.第一个触发器在工具中的相关内容

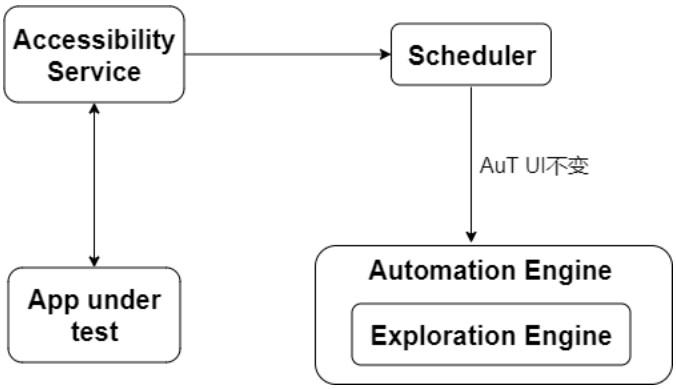

图 12.第一种情况的交互

第二个是操作超时，一些 app 和小部件会定期触发可访问性事件，这种情况下第一个触发器会失效。为防止 exploration 一直等待 UI 更新，Scheduler 有一个硬超时触发器，如果应用程序在最大时间间隔内没有稳定下来，Scheduler 会通知 Exploration Engine 与屏幕上任意部件进行交互。

```
}
//第二个trigger
private suspend fun runScheduledTimeout(delayInterval: Long) {
    log.trace("Scheduling timeout task with interval of $delayInterval ms")
    while (!isCanceled) {
        // 决定应该等多久
        val delayTime = delayInterval - (SystemClock.uptimeMillis() - lastTimeout.get())
        log.trace("Delaying timeout for for $delayTime ms")
        // 等待
        delay(delayTime)

        mutex.withLock {
            log.trace("(Locked) Checking if timeout task must run")
            // 检查预期的时间戳
            val expectedTimestamp = lastTimeout.get() + delayTime

            // 如果在预期的时间戳之后，更新时间戳并通知idle
            val currentTime = SystemClock.uptimeMillis()
            if (currentTime > expectedTimestamp) {
                log.trace("(Locked) Timeout task must run, notifying channel")
                lastTimestamp.set(currentTime)
                lastTimeout.set(currentTime)
                idleNotificationChannel.send(currentTime)
            }
```

图 13.第二个触发器

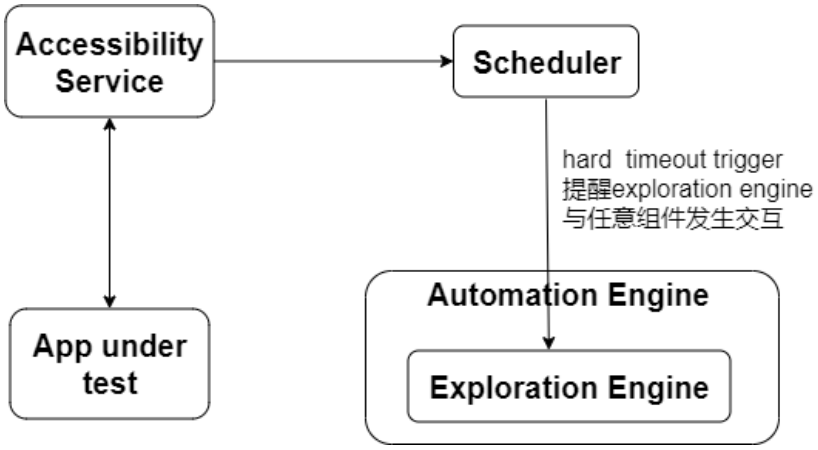

图 14.第二种情况下的交互

2.5 Automation Engine

Automation Engine 是设备和测试生成策略之间的桥梁。它有两个角色，当 Scheduler 通知获取当前 UI 状态，以及将 Exploration Engine 的高级交互转换为设备命令。

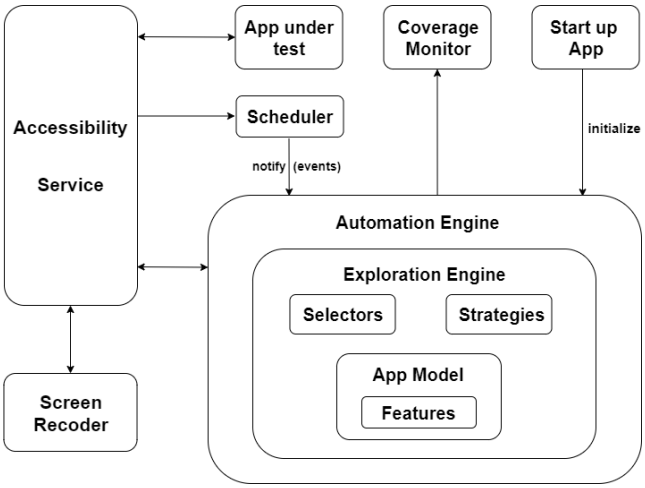

图 15. Automation Engine 相关交互

为获取 UI 状态，Automation Engine 查询 Screen Recorder 以获取屏幕截图和现有小部件的 Accessibility service。

```
//获取屏幕截图
override fun takeScreenshot(actionNr: Int): Bitmap? {
    return screenshotEngine.takeScreenshot(actionNr)
}

override fun getAndStoreImgPixels(bm: Bitmap?, actionId: Int, delayedImgTransfer: Boolean): ByteArray {
    return screenshotEngine.getAndStoreImgPixels(bm, actionId, delayedImgTransfer)
}
```

图 16. Automation Engine 获取屏幕截图

然后使用 App Model 将这些数据转换为 UI 状态，并将其转发给 Exploration Engine，由其决定如何与 App under test 交互。

Exploration Engine 产生高层交互，如“启动应用程序”、“terminate exploration”，Automation Engine 将其转变为低级别的可访问性事件序列，并转发给 Accessibility service。如图 17，它将“启动应用”交互翻译成显示和清除最近的应用列表，打开应用菜单，搜索应用程序，并单击应用程序节点。

```
// 启动应用程序
val loadTime = measureTimeMillis {
    // 启动
    val intent = context.applicationContext.packageManager
        .getLaunchIntentForPackage(appPackageName)

    // 清除任何以前的实例，防止在同一屏幕上重新打开应用程序
    intent?.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK)

    // 更新环境
    launchedMainActivity = try {
        intent?.component?.className ?: ""
    } catch (e: IllegalStateException) {
        ""
    }
    debugOut( msg: "determined launch-able main activity for pkg=$launchedMainActivity", debugFetch)
    GlobalScope.launch(Dispatchers.Main) {  this: CoroutineScope
        context.startActivity(intent)
    }
    delay(launchActivityDelay)
    success = uiHierarchy.waitFor( env: this, interactiveTimeout, UiSelector.actionableAppElem)
    log.trace("Fallback app launch strategy succeeded = $success")

    // 在应用程序启动后静音音频(对于复杂的应用程序，需要一个contentObserver侦听音频设置更改)
    val audio = context.getSystemService(Context.AUDIO_SERVICE) as AudioManager
    audio.adjustStreamVolume(AudioManager.STREAM_MUSIC, AudioManager.ADJUST_MUTE,  flags: 0)
    audio.adjustStreamVolume(AudioManager.STREAM_RING, AudioManager.ADJUST_MUTE,  flags: 0)
    audio.adjustStreamVolume(AudioManager.STREAM_ALARM, AudioManager.ADJUST_MUTE,  flags: 0)
    // }
```

图 18. Automation Engine 对“启动应用”的翻译

最后在接收到“终止探索”交互后，Exploration Engine 关闭 Automation Engine，停用 Accessibility service，并存储提取的 App Model 以及相关的 Coverage Data(代码覆盖率数据)。

```
// 收到terminate的命令，终止探索
    fun terminateExploration() {
        // 返回主页
        pressHome()
        // 再次为用户打开可访问性设置以禁用可访问性应用程序
        val intent = Intent(Settings.ACTION_ACCESSIBILITY_SETTINGS)
        intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK
        context.startActivity(intent)

        try {
            Files.createFile(explorationDoneFile)
        } catch (e: IOException) {
            //无法创建
        }
```

图 19.收到“terminate”交互

```
//获取屏幕截图
override fun takeScreenshot(actionNr: Int): Bitmap? {
    return screenshotEngine.takeScreenshot(actionNr)
}
//获取信息
override fun getAndStoreImgPixels(bm: Bitmap?, actionId: Int, delayedImgTransfer: Boolean): ByteArray
    return screenshotEngine.getAndStoreImgPixels(bm, actionId, delayedImgTransfer)
}
```

图 20.存储提取的数据

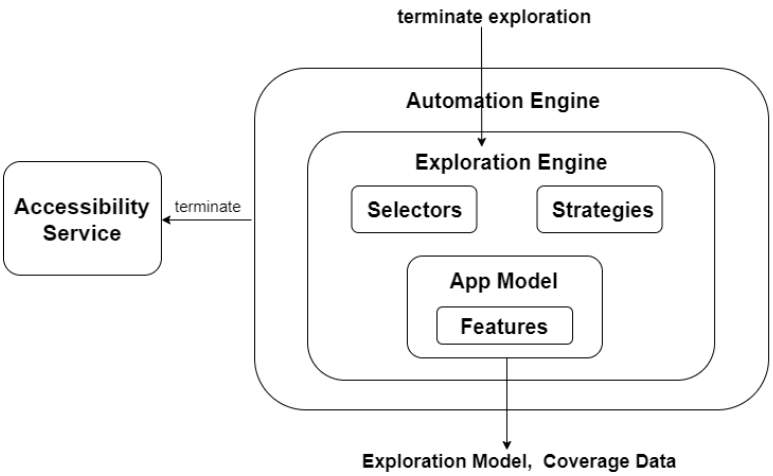

图 21.收到“终止探索”后的交互

3. 运行要求

3.1Building

Build 可以在 Android Studio 中配置好环境后直接 build 或是命令行中使用 `./gradlew clean build`

3.2 Run

1）将 `defaultConfig.properties` 文件 push 到 `/sdcard/` ，因为我们不能在默认情况下使用 Konfig 加载它作为资源

adb push scripts/defaultConfig.properties /sdcard/

2）打开工具

3）接受权限

4）选择一个 app

3.3 Notes

1）这款应用在 Android 10 上运行得更好，Android 10 的 accessibility service 更稳定

2）结果存储在 `/sdcard/DM-2/`