OpenHarmony XTS 测试框架分析与实践报告

目录

一、XTS 测试框架的结构与运行机制

1.1 框架整体架构分析

OpenHarmony XTS 采用"宿主机驱动、设备执行"的分离式架构。这一设计决策源于嵌入式设备的资源限制——OpenHarmony 目标设备可能是内存仅有 128MB 的轻量级 IoT 设备，无法承载完整的测试框架运行时。因此 XTS 将测试调度、结果收集、报告生成等计算密集型任务放在 PC 端的 xdevice 工具中，设备端仅负责执行具体的 shell 命令并返回结果。

这种架构带来了两个显著优势：第一，测试用例可以使用 Python 编写，开发效率远高于 C/C++，且无需交叉编译；第二，测试框架的升级迭代不影响设备端系统镜像，降低了维护成本。但也存在局限性——所有测试都必须通过 shell 命令或 HDC 协议间接完成，无法直接调用设备端的 native API 或进行内存级别的白盒测试。

1.2 测试计划的三层配置模型

XTS 的测试计划采用 JSON 格式，体现了"声明式配置"的设计理念。以 ActsPCSPyTest.json 为例，其结构分为三层：

环境层声明测试所需的硬件环境。type 为 device 表示需要真实设备，这与 host 类型（纯宿主机测试）形成对比。环境层的抽象使得同一套用例可以在不同硬件配置下复用，只需修改环境声明即可。**驱动层**定义测试执行引擎。DeviceTestSuite 驱动负责加载 Python 测试套件，按 suitecases 数组顺序依次执行用例。这里的设计思考是：为什么不自动发现所有 test*.py 文件，而要显式列出？原因在于兼容性认证场景需要精确控制用例集合和执行顺序，自动发现机制可能引入不确定性。**资源层**管理测试资源的生命周期。PushKit 在测试开始前将 hap 安装包推送到设备的 /data/local/tmp 目录，ShellKit 在测试结束后执行 bm uninstall 清理残留。这种"setup-teardown"模式确保了测试环境的可重复性——每次执行都从干净状态开始，避免用例间的相互干扰。

1.3 用例生命周期与设备抽象

每个测试用例继承自 TestCase 基类，遵循 setup → process → teardown 的三阶段生命周期。这一设计借鉴了 JUnit 的 @Before/@Test/@After 模式，但针对嵌入式测试场景做了适配：setup 阶段通常用于检查设备连接状态和前置条件，process 阶段执行核心测试逻辑，teardown 阶段清理测试产生的临时文件或状态变更。

设备抽象是框架的核心设计。TestCase 通过 self.device1 对象与设备交互，该对象在本地调试时是 _DummyDevice（打印命令并返回模拟输出），在真实执行时由 xdevice 注入通过 HDC 协议通信的真实控制器。这种依赖注入模式使得用例代码无需修改即可在两种环境下运行，大幅提升了开发调试效率。

1.4 执行流程的关键节点

xdevice 的执行流程可以概括为：解析配置 → 设备发现 → 资源部署 → 用例执行 → 结果收集 → 报告生成。其中设备发现阶段通过 `hdc list targets` 枚举可用设备，这与 Android 的 `adb devices` 机制类似。资源部署阶段的 PushKit 实际上封装了 `hdc file send` 命令，将本地文件传输到设备指定路径。

用例执行阶段，xdevice 为每个 TestCase 创建独立的执行上下文，捕获 setup/process/teardown 中的所有输出和异常。如果 process 阶段抛出 AssertionError，该用例被标记为 FAIL；如果抛出其他异常，被标记为 ERROR；正常完成则为 PASS。这种异常驱动的结果判定机制简化了用例编写——开发者只需使用 Python 的 assert 语句表达预期，无需显式调用结果上报 API。

二、测试用例设计与子系统分析

2.1 子系统选择的技术考量

本项目选择了系统服务管理（SAMGR/init）、电源管理、包管理、文件系统四个子系统作为测试目标。选择这些子系统的原因如下：

系统服务管理是 OpenHarmony 的核心基础设施。init 进程是系统启动后的第一个用户态进程，负责解析 /system/etc/init 下的 .cfg 配置文件并启动所有系统服务。如果 init 配置存在语法错误或关键字段缺失，将导致系统服务无法正常启动，进而影响整个系统的可用性。因此针对 init 配置的测试具有高优先级。

电源管理影响设备的续航和用户体验。power-shell 工具提供了查询电源状态、设置息屏时间等接口，是验证电源子系统功能完整性的有效入口。选择电源管理还有一个实际考虑：其命令行接口相对稳定，不同设备型号间的差异较小，测试用例的可移植性较好。

包管理是应用生态的基础。bm 工具负责应用的安装、卸载、查询等操作，其正确性直接决定了第三方应用能否正常运行。包管理测试还涉及签名验证、权限校验等安全相关功能，是兼容性认证的重点领域。

2.2 用例设计思路详解

testShowDateCommand

设计思路：时间同步是分布式系统的基础能力。OpenHarmony 设备可能在无网络环境下运行，此时需要通过 PC 端手动同步时间。该用例验证 date 命令的双向能力：既能读取当前时间，也能设置新时间。

技术难点：date 命令的时间格式因 Toybox/Busybox 版本而异。经过调研，OpenHarmony 使用的格式为 MMDDhhmmYYYY.ss（月日时分年.秒），与 Linux 标准的 date -s "YYYY-MM-DD HH:MM:SS" 不同。用例中使用 Python 的 strftime 生成正确格式的时间字符串，避免了硬编码。

验证策略：设置时间后等待 1 秒再读取，通过检查返回值中是否包含预期的年份和月份缩写（如 Nov、2025）来判断同步是否成功。这种"宽松匹配"策略容忍了时区差异和秒级误差，提高了用例的鲁棒性。

testListDataDir

设计思路：/data 目录是 OpenHarmony 的用户数据分区，其子目录结构反映了系统服务的部署状态。如果 samgr、service 等关键目录缺失，说明系统服务管理子系统存在严重问题。

技术难点：不同设备型号的 /data 目录结构可能略有差异。用例定义了 app、log、samgr、service、vendor 五个"必须存在"的核心目录，这是基于对多个 OpenHarmony 发行版的分析得出的最小公共集合。

验证策略：采用"白名单检查"而非"完全匹配"——只要核心目录存在即通过，允许存在额外目录。这种策略平衡了严格性和兼容性。

testInitServicesMetaCheck

设计思路：init 配置文件中的 services 数组定义了系统服务的元数据，其中 name 字段是服务标识（必须存在），apl（Application Privilege Level）字段声明服务的权限等级。缺少 apl 声明的服务可能存在权限配置遗漏，是潜在的安全风险点。

技术难点：/system/etc/init 目录下可能有上百个 .cfg 文件，部分文件可能不是有效的 JSON（如注释文件或空文件）。用例需要健壮地处理这些异常情况，避免因单个文件解析失败而中断整体扫描。

验证策略：采用"统计审计"模式而非"严格断言"——缺少 name 字段的服务会触发断言失败（这是配置错误），但缺少 apl 字段仅记录警告并统计数量。这种分级处理反映了两类问题的严重程度差异。实际执行结果显示，在 120 个配置文件中有 83 个服务缺少 apl 声明，这一数据为后续的安全加固工作提供了量化依据。

testPowerShellHelp

设计思路：电源管理子系统提供了 power-shell 命令行工具，支持 dump（查询状态）、timeout（设置息屏时间）、suspend（休眠）、wakeup（唤醒）等操作。该用例选择 dump 和 timeout 两个低风险操作进行验证，避免在自动化测试中触发设备休眠导致后续用例无法执行。

技术难点：power-shell dump 需要 -a 参数才能输出完整信息，这是通过阅读源码和实际调试发现的。不带参数的 dump 命令返回空输出，容易被误判为命令不可用。

验证策略：通过输出长度判断 dump 命令是否正常工作（正常输出超过 200 字符），通过返回值中是否包含 success 字样判断 timeout 设置是否生效。这种基于输出特征的验证方式不依赖具体的数值，适应性更强。

testBundleManagerHelp

设计思路：bm 是包管理子系统的命令行入口，其 help 子命令返回所有可用操作的列表。该用例作为包管理测试的"冒烟测试"，验证 bm 工具的基本可用性，为后续的 install、uninstall 等复杂操作测试奠定基础。

验证策略：仅断言输出非空。这种极简断言的设计考量是：help 命令的输出格式可能随版本变化，过于具体的断言（如检查是否包含 install 字样）可能导致误报。

2.3 用例间的依赖与执行顺序

本项目的 10 个用例被设计为相互独立，可以任意顺序执行。这是 XTS 用例设计的最佳实践——每个用例应该是自包含的，不依赖其他用例的执行结果或副作用。

但在实际的测试计划中，我们仍然按照"基础检查 → 子系统验证 → 复杂场景"的顺序排列用例。例如 testSimpleShellLs 和 testListDataDir 被放在前面，因为它们验证的是最基础的 shell 命令执行能力；如果这两个用例失败，后续依赖 shell 命令的用例大概率也会失败，提前暴露问题可以节省调试时间。

三、自动化批量执行脚本设计

3.1 设计目标与技术选型

批量执行脚本的设计目标是：在无真实设备的开发环境下，能够快速验证所有用例的语法正确性和基本逻辑；生成可视化的测试报告，便于快速定位失败用例。技术选型上，我们选择了 subprocess 模块调用 Python 解释器执行单个用例脚本，而非在同一进程中 import 执行。这一决策基于以下考虑：第一，进程隔离确保单个用例的崩溃不会影响其他用例的执行；第二，与真实 xdevice 的执行方式保持一致，减少环境差异导致的问题；第三，便于捕获每个用例的完整标准输出和错误输出。

3.2 报告生成的设计思考

测试报告采用 HTML 格式而非纯文本或 Markdown，原因是 HTML 支持交互式元素。报告中每个用例的详细日志默认折叠，点击"查看日志"按钮展开。这种设计适应了两种使用场景：快速浏览时只看汇总数据和 PASS/FAIL 状态；深入调试时展开失败用例的完整日志。报告的视觉设计参考了 Jenkins 和 Allure 等主流 CI 工具的风格：绿色表示通过，红色表示失败，使用表格呈现结构化数据。这种设计降低了用户的认知负担，无需阅读文档即可理解报告内容。

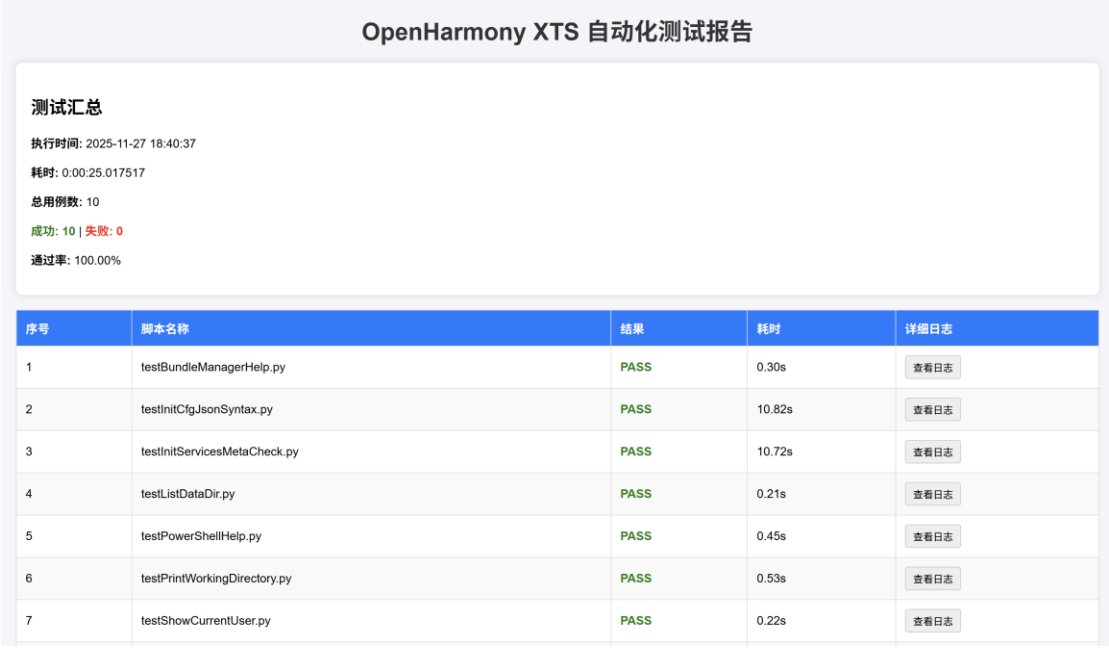

图 1 XTS 自动化测试报告

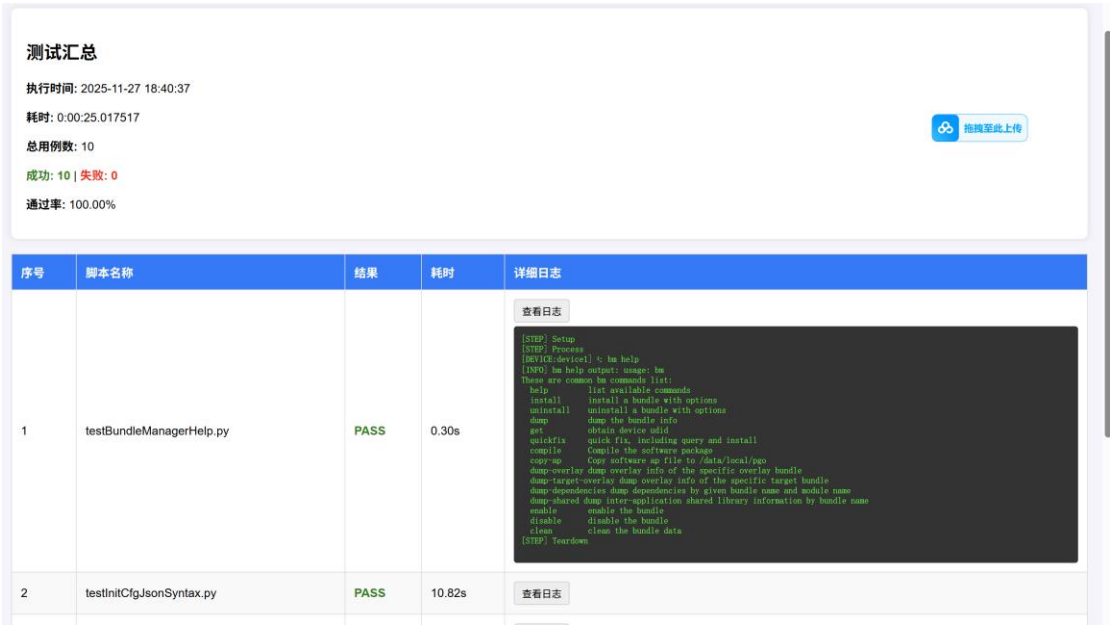

图 2 自动化测试报告详细日志

3.3 与 xdevice 的集成

除了独立的 run_all.py，我们还实现了 xts_batch_runner.py 用于与标准 xdevice 环境集成。该脚本的核心能力是解析 xdevice 生成的 JUnit XML 报告，提取用例级别的统计数据。JUnit XML 是业界通用的测试报告格式，其结构为：testsuite 节点包含多个 testcase 节点，每个 testcase 通过 failure、error、skipped 子节点表示非通过状态。我们的解析逻辑遍历所有 testcase 节点，根据子节点类型统计 passed/failed/skipped 计数，最终生成汇总的 JSON 和 Markdown 报告。这使批量执行多个测试计划成为可能——每个计划生成独立的 JUnit XML，xts_batch_runner.py 将它们聚合为统一的全局报告，便于一次性查看所有子系统的测试结果.

四、已有 XTS 用例缺陷分析与修复方案

在阅读官方提供的文档 https://gitcode.com/openharmony/xts_acts 后，我们发现了几个错误，并给所有的发现提了 issue.（[Bug]: 本报告列出了在 `pcs/pcs_py/` 目录下所有测试用例文件中发现的代码错误和问题。-xts_acts-AtomGit | GitCode）

图 3 gitcode 提 issue 截图

4.1 缺陷发现过程

在阅读 pcs/pcs_py 目录下的官方用例代码时，我们采用了"模式匹配"的审查方法：首先识别出用例中的常见操作模式（如 execute_shell_command 调用、文件路径引用、属性访问），然后检查这些模式的使用是否一致。通过这种方法，我们发现 testSysCapNameOnly.py 中的 cat 命令使用了相对路径 ./system/...，而其他所有用例都使用绝对路径 /system/...。这种不一致引起了我们的警觉，进一步分析确认这是一个 bug。

PermissionUtils.py 的问题则是通过静态分析发现的：isInPermissionList 方法引用了 self.permissionList，但类定义中只有 systemPermissionList 和 userPermissionList 两个属性。这种"属性名拼写错误"是 Python 动态类型语言的常见陷阱，静态类型检查工具（如 mypy）可以自动发现此类问题。

4.2 缺陷一：相对路径导致文件读取失败

testSysCapNameOnly.py 第 36 行使用 `cat ./system/etc/param/syscap.para` 读取系统能力配置文件。在设备端执行 shell 命令时，当前工作目录由 HDC 会话决定（通常为 / 或 /data/local/tmp），相对路径 ./system/... 无法正确解析。

该用例在任何设备上执行都会失败，返回 "No such file or directory" 错误。后续的 sysCapUtils.getAllSysCaps 解析空字符串时会返回空列表，导致断言 `len(sysCapsList) == len(set(sysCapsList))` 虽然通过（空列表的长度等于空集合的长度），但实际上并未完成预期的唯一性校验。

将相对路径改为绝对路径 `/system/etc/param/syscap.para`。这是最小化修复，不改变用例的其他逻辑。在嵌入式测试场景中，应始终使用绝对路径引用设备端文件。可以考虑在 TestCase 基类中添加路径校验逻辑，对包含 ./ 或 ../ 的路径发出警告。

4.3 缺陷二：属性引用错误导致运行时异常

PermissionUtils.py 的 isInPermissionList 方法引用了不存在的 self.permissionList 属性。Python 在运行时才进行属性解析，因此这个错误只有在实际调用该方法时才会暴露。

任何调用 isInPermissionList 方法的用例都会抛出 AttributeError 异常。由于该方法目前可能未被使用，这个 bug 处于"潜伏"状态，但一旦有新用例依赖该方法，就会立即触发。

根据方法的语义（检查权限是否在"权限列表"中），合理的实现应该是检查 systemPermissionList 和 userPermissionList 的并集。修复后的代码为 `return permission in self.systemPermissionList or permission in self.userPermissionList`。建议在 XTS 用例仓库中引入静态类型检查（如 mypy）和单元测试覆盖，在代码合入前自动发现此类问题。

五、XTS 与 Google GTest、Android CTS 测试机制对比

5.1 架构层面的本质差异

GTest 采用"进程内测试"模式。测试代码通过 TEST 宏定义，与被测代码编译链接为同一可执行文件。运行时，测试进程既是驱动者也是被测对象，可以直接调用被测函数、访问内部数据结构、甚至 mock 依赖组件。这种模式的优势是测试粒度细、执行效率高，适合单元测试和模块级回归测试；劣势是测试代码与被测代码强耦合，难以测试跨进程或跨设备的场景。

XTS 和 CTS 采用"宿主机驱动"模式。测试代码运行在 PC 端，通过 HDC（XTS）或 ADB（CTS）协议与被测设备通信。测试进程与被测系统运行在不同的硬件上，只能通过命令行接口、文件系统、网络协议等"黑盒"方式进行交互。这种模式的优势是测试环境与生产环境一致、可以测试完整的系统行为；劣势是测试粒度粗、执行效率低、难以进行白盒测试。

5.2 用例组织方式的设计

GTest 遵循"代码即配置"的理念。测试用例通过 TEST(TestSuiteName, TestName) 宏定义，框架在链接时自动收集所有测试并注册到全局列表。这种设计简化了用例管理——开发者只需编写测试函数，无需维护额外的配置文件。但缺点是难以实现灵活的用例筛选和分组，通常需要借助 --gtest_filter 命令行参数。

XTS 采用"显式计划文件"的方式。ActsPCSPyTest.json 中的 suitecases 数组明确列出了所有待执行的用例脚本，执行顺序与数组顺序一致。这种设计的优势是：同一套用例代码可以组合成不同的测试计划（如完整认证计划、快速冒烟计划、特定子系统计划），无需修改用例代码；劣势是需要手动维护计划文件，新增用例时容易遗漏。

CTS 的设计介于两者之间。用例通过 Java 注解（@Test）标记，但需要在 AndroidTest.xml 中声明测试模块的元数据。TradeFed 框架根据 XML 配置加载测试模块，支持通过 --include-filter 和 --exclude-filter 参数动态筛选用例。

5.3 断言机制与失败处理

GTest 提供了两类断言宏：EXPECT_* 系列在断言失败后继续执行后续代码，ASSERT_* 系列在断言失败后立即终止当前测试函数。这种设计允许开发者在一个测试函数中进行多个断言，即使前面的断言失败，后面的断言仍会执行，便于一次性发现多个问题。

XTS 使用 Python 的原生 assert 语句。断言失败会抛出 AssertionError 异常，立即终止当前用例的 process 方法。这种"快速失败"的行为与 ASSERT_* 类似，但缺少 EXPECT_* 的"继续执行"能力。如果需要在一个用例中进行多个独立检查，开发者需要手动捕获异常或使

用 try-except 结构。

CTS 基于 JUnit 框架，提供了 assertEquals、assertTrue、assertNotNull 等断言方法。JUnit 4 的断言失败会抛出 AssertionError，行为与 Python assert 类似；JUnit 5 引入了 assertAll 方法，支持"软断言"模式，可以收集所有失败后再统一报告。

5.4 报告与结果分析

GTest 通过 --gtest_output=xml:report.xml 参数生成报告，格式为 JUnit 兼容的 XML。报告包含每个测试的名称、执行时间、失败信息，但不包含标准输出内容（除非使用 --gtest_print_time 等参数）。

XTS 的 xdevice 工具生成更丰富的报告，包括 JUnit XML、HTML 可视化报告、以及包含完整日志的文本文件。报告目录结构按执行时间戳组织，便于追溯历史执行记录。

CTS 的 TradeFed 框架生成的报告最为完整，包括设备信息、测试计划、用例结果、logcat 日志、bugreport 等。CTS 还提供了 cts-tradefed 交互式控制台，支持在测试过程中实时查看结果和日志。

5.5 适用场景总结

GTest 最适合 C/C++ 项目的单元测试和模块级回归测试，尤其是需要 mock 依赖、测试内部实现细节的场景。OpenHarmony 的 native 组件（如 HDF 驱动、图形子系统）内部也大量使用 GTest 进行开发阶段的自测。

XTS 最适合系统级兼容性验证，验证设备是否符合 OpenHarmony 兼容性规范。其"黑盒"测试方式确保了测试结果的客观性——只要设备对外行为符合规范，内部实现可以自由变化。

CTS 的定位与 XTS 类似，是 Android 生态的兼容性认证工具。两者在架构设计上高度相似，都采用宿主机驱动模式和显式计划文件，这种相似性并非巧合——OpenHarmony 的 XTS 框架在设计时参考了 CTS 的成熟经验。

在实际工程中，三种框架可以组合使用：底层模块使用 GTest 进行细粒度的单元测试，系统集成阶段使用 XTS 进行端到端的兼容性验证，两者互为补充，共同保障系统质量。