



Unisoc Confidential For kxdwww

# Android 14 应用预置指南


文档版本                      V1.0  
发布日期                      2023-08-18

## 版权所有 © 紫光展锐（上海）科技有限公司。保留一切权利。

本文件所含数据和信息都属于紫光展锐（上海）科技有限公司（以下简称紫光展锐）所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负任何与本文件相关的直接或间接的、任何伤害或损失。

请参照交付物中说明文档对紫光展锐交付物进行使用，任何人对紫光展锐交付物的修改、定制化或违反说明文档的指引对紫光展锐交付物进行使用造成的任何损失由其自行承担。紫光展锐交付物中的性能指标、测试结果和参数等，均为在紫光展锐内部研发和测试系统中获得的，仅供参考，若任何人需要对交付物进行商用或量产，需要结合自身的软硬件测试环境进行全面的测试和调试。

## 商标声明

**紫光展锐**、**UNISOC**、、展讯、Spreadtrum、SPRD、锐迪科、RDA 及其他紫光展锐的商标均为紫光展锐（上海）科技有限公司及/或其子公司、关联公司所有。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

## 免责声明

本文档可能包含第三方内容，包括但不限于第三方信息、软件、组件、数据等。紫光展锐不控制且不对第三方内容承担任何责任，包括但不限于准确性、兼容性、可靠性、可用性、合法性、适当性、性能、不侵权、更新状态等，除非本文档另有明确说明。在本文档中提及或引用任何第三方内容不代表紫光展锐对第三方内容的认可、承诺或保证。

用户有义务结合自身情况，检查上述第三方内容的可用性。若需要第三方许可，应通过合法途径获取第三方许可，除非本文档另有明确说明。

# 紫光展锐（上海）科技有限公司



# 前言

## 概述

本文档主要介绍 Android 系统常见预置需求，包括 apk、可执行文件、native service、so 库以及 jar 包的预置。

## 读者对象




本文档主要适用于 Android 平台系统开发人员。

## 缩略语

缩略语	英文全名	中文解释
API	Application Programming Interface	应用程序编程接口
JNI	Java Native Interface	Java 原生接口
NDK	Native Development Kit	Native 开发工具集
VNDK	Vendor Native Development Kit	供应商原生开发套件

## 符号约定

在本文中可能出现下列符号，每种符号的说明如下。

符号	说明
 <b>说明</b>	用于突出重要或关键信息、补充信息和小窍门等。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害。
 <b>注意</b>	用于突出容易出错的操作。 “注意”不是安全警示信息，不涉及人身、设备及环境伤害。
 <b>警告</b>	用于可能无法恢复的失误操作。 “警告”不是危险警示信息，不涉及人身及环境伤害。

## 变更信息

文档版本	发布日期	修改说明
V1.0	2023-08-18	第一次正式发布

## 关键字

应用预置、分区、Android.mk、Android.bp

Unisoc Confidential For kxdwww

# 目 录

1 应用预置概述.....	1
1.1 应用预置.....	1
1.2 应用预置目录.....	1
1.3 编译规则.....	2
1.3.1 Android.mk.....	3
1.3.2 Android.bp.....	6
1.4 常见需求.....	6
1.4.1 预置应用可卸载.....	7
1.4.2 预置应用不可卸载.....	7
1.4.3 预置应用可升级.....	7
1.4.4 apk odex 化.....	7
1.4.5 预置应用替换.....	8
2 预置操作.....	9
2.1 预置有源码的应用.....	9
2.2 预置无共享库的应用.....	10
2.3 预置有共享库的应用.....	12
2.3.1 动态链接器.....	12
2.3.2 namespace 访问权限.....	14
2.3.3 预置应用带有 so 库.....	21
2.4 预置可执行程序.....	23
2.5 预置 native service.....	23
2.6 预置 so 库.....	25
2.6.1 复制到指定位置.....	25
2.6.2 应用中使用 so 库.....	25
2.6.3 预置为系统 so 库.....	26
2.7 集成 jar 包.....	26
2.7.1 应用中集成 jar 包.....	26
2.7.2 系统中集成 jar 包.....	27
3 常见问题.....	30
3.1 替换系统签名.....	30
3.2 签名不一致.....	30
3.3 V2 签名后，应用安装失败.....	30
3.4 预置可卸载应用出错.....	31
3.5 预置应用配置特殊权限组群.....	31
3.6 manifest_checker 错误.....	33

---

4 参考文档.....	36
-------------	----

Unisoc Confidential For kxdwww

## 图目录

图 2-1 动态链接器加载机制 .....	12
图 2-2 预置源文件清单 .....	19
图 2-3 同名系统库说明 .....	19
图 2-4 应用预置共享库的清单文件 .....	25
图 2-5 系统库预置的清单文件 .....	26
图 3-1 uses_libs 和 uses_library 检查失败 .....	34

Unisoc Confidential For kxdwww

# 表目录

表 1-1 应用预置目录 .....	1
表 1-2 Android.mk 与 Android.bp 差异 .....	2
表 2-1 分区及属性配置 .....	9
表 2-2 关联的 namespace 支持列表 .....	13
表 2-3 system/system_ext app 可访问权限 .....	14
表 2-4 product/vendor app 可访问权限 .....	15
表 2-5 preloadapp app 可访问权限 .....	16
表 2-6 system 及 system_ext 分区可访问权限 .....	17
表 2-7 product 分区可访问权限 .....	20
表 2-8 vendor 分区可访问权限 .....	21

Unisoc Confidential For kxdwww



# 1 应用预置概述

## 1.1 应用预置

预置指移动智能终端设备出厂前，将文件预先安装到系统中。预置对象包括应用程序、可执行文件、so 库（.so 文件）、jar 包等。

预置方式有以下两种：

- 预编译方式
  - 如果文件有明确的预编译规则，且预编译不破坏当前程序完整性，则可使用预编译方式。
  - 如果文件有其他模块编译依赖，或需要系统签名，则需要定义一个预编译模块，例如一些 so 库、apk 文件、jar 包等。  
预置应用时，可使用变量 LOCAL\_CERTIFICATE 指定签名类型，常见签名类型见 [1.3.1 Android.mk](#)。
- 复制方式
  - 将文件复制到目标目录（使用 shell 命令“cp”进行复制），参与打包即可实现预置。
  - 如果文件只需预置到指定目录，将该文件添加到 PRODUCT\_COPY\_FILES 变量中即可。例如一些 bin 文件、配置文件。

## 1.2 应用预置目录

常见的应用预置分区包括 system、system\_ext、product、vendor、odm 等。

- 不同的分区下安装路径所对应的权限不相同。
- 不同的分区所属域不相同：system、system\_ext 属于 system 域，其余分区都属于 vendor 域。

定义宏 PARTITION 为 system\_ext、product、vendor、odm 中的任意一个分区，后文中以 {PARTITION} 表示。

应用预置目录说明如[表 1-1](#) 所示。

表1-1 应用预置目录

目录	说明	备注
system/app	系统核心应用目录，预置到该目录的应用不可卸载，设备恢复出厂设置后仍然存在。	建议将需要申请系统权限，且重要的应用，预置到该目录。
system/priv-app		普通应用不推荐预置进该目录。该目录下的应用权限比 system/app 高。
system/preloadapp	第三方应用预置目录，预置到该目录	preloadapp 中预置的应用在开机过程中同

目录	说明	备注
	的应用可卸载，应用被卸载后设备恢复出厂设置时可恢复。	步多线程扫描安装，可加快开机速度。
system/vital-app		预置在该目录和 system/preloadapp 目录下的效果一致。
{PARTITION}/app	系统应用目录，预置到该目录的应用不可卸载，设备恢复出厂设置后仍然存在。	相较于 system 分区，CTS 会检查预置进 vendor、product 分区中的应用 API（Application Programming Interface，应用程序编程接口）是否合规。
{PARTITION}/priv-app		相较于 {PARTITION}/app 目录，此目录中的应用不签署 platform 签名也可获得供应商特殊权限。

## 1.3 编译规则

Android.mk 与 Android.bp 是 Android 系统用于管理源码编译的两种规则文件，一部分基本声明差异项如表 1-2 所示。

当前 Android.mk 和 Android.bp 可以按需选择，建议选择 Android.bp 作为编译规则，因为 Android.mk 在 Android 15 及之后的版本将不再支持。

本章后续内容说明了如何编写 Android.mk 和 Android.bp 的编译规则。

表1-2 Android.mk 与 Android.bp 差异

基本声明差异项	Android.mk	Android.bp
模块名	LOCAL_PACKAGE_NAME := [PackageName] //编译生成的模块名	name: [PackageName] //编译生成的模块名
包含的 src 文件范围	LOCAL_SRC_FILES := \ \$(call all-java-files-under, src) \ \$(call all-java-files-under, src_ui_overrides) \ \$(call all-java-files-under, go/src)	srcs: [ "src/**/*.java", "src_shortcuts_overrides/**/*.java", "src_ui_overrides/**/*.java", "ext_tests/src/**/*.java", ],
包含的 res 文件范围	LOCAL_RESOURCE_DIR := \$(LOCAL_PATH)/quickstep/res	resource_dirs: [ "ext_tests/res", ],
导入静态 Java 库依赖	LOCAL_STATIC_JAVA_LIBRARIES := \ SystemUI-statsd \ SystemUISharedLib	static_libs:[ "Launcher3ResLib", "SystemUISharedLib", "SystemUI-statsd",

基本声明差异项	Android.mk	Android.bp
		],
指定编译后生成文件是否放到 <b>priv-app</b> （默认放到 <b>app</b> ）	LOCAL_PRIVILEGED_MODULE := true	privileged: true,
指定编译后生成文件是否放到 <b>system-ext</b> （默认放到 <b>system</b> ）	LOCAL_SYSTEM_EXT_MODULE := true	system_ext_specific: true,
指定覆盖编译（被指定的模块会在整体编译时被编译进去）	LOCAL_OVERRIDES_PACKAGES := Home Launcher2 Launcher3	overrides: [ "Home", "Launcher2", "Launcher3", ],
Manifest 文件	LOCAL_MANIFEST_FILE := quickstep/AndroidManifest.xml	manifest: "quickstep/AndroidManifest.xml",
附加 Manifest 文件	LOCAL_FULL_LIBS_MANIFEST_FILES := \ \$(LOCAL_PATH)/quickstep/AndroidManifest-launcher.xml \ \$(LOCAL_PATH)/AndroidManifest-common.xml	additional_manifests: [ "go/AndroidManifest.xml", "AndroidManifest-common.xml", ],
代码混淆控制	LOCAL_PROGUARD_ENABLED := disabled LOCAL_PROGUARD_FLAG_FILES:=proguard.flags	optimize: { proguard_flags_files: ["proguard.flags"], // Proguard is disable for testing. Derivarive prjects to keep proguard enabled enabled: false, },

### 1.3.1 Android.mk

Android.mk 中，每个编译模块都以 include \$(CLEAR\_VARS)开始，以 include \$(BUILD\_XXX)结束，示例如下。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := myapp
LOCAL_SRC_FILES := app/myapp.apk
LOCAL_MODULE_PATH := $(TARGET_OUT_VENDOR)/app
LOCAL_MODULE_CLASS := APPS
LOCAL_MODULE_TAGS := optional
```

```
LOCAL_CERTIFICATE := platform
LOCAL_SHARED_LIBRARIES := liba libc
LOCAL_MULTILIB := 64
include $(BUILD_PREBUILT)
```

- **LOCAL\_PATH**

模块编译路径，需要与 `Android.mk` 脚本在同一目录。每一个 `Android.mk` 文件必须先定义 `LOCAL_PATH`。`my-dir` 是系统提供的宏函数，`$(call my-dir)` 将返回当前目录（`Android.mk` 文件本身所在的目录）的路径。

Build 系统中还定义了一些便捷的函数以便在 `Android.mk` 中使用，例如：

- `$(call all-java-files-under,xxx)`：获取指定目录下的所有 Java 文件。
- `$(call all-c-files-under,xxx)`：获取指定目录下的所有 C 文件。
- `$(call all-aidl-files-under,xxx)`：获取指定目录下的所有 AIDL 文件。
- `$(call all-makefiles-under,xxx)`：获取指定目录下的所有 make 文件。

- **include \$(CLEAR\_VARS)**

`CLEAR_VARS` 是系统提供的宏变量，用于清理除 `LOCAL_PATH` 以外的所有 `LOCAL_XXX` 变量。整个编译上下文中，所有的变量都是全局变量，`CLEAR_VARS` 可以保证这些变量只在局部范围内起作用。`LOCAL_PATH` 要求在每个模块中都要设置，无需清空。

- **LOCAL\_MODULE**

模块名称，是模块在编译中的唯一标识，该值必须唯一且不包含空格。模块间的依赖关系通过模块名称进行引用。编译系统会根据编译类型自动添加适当的后缀，例如要编译成一个 `apk`，则生成 `myapp.apk`。

- **LOCAL\_SRC\_FILES**

模块生成目标文件所需要的所有 C 和/或 C++ 源文件列表，多个文件用空格隔开。

- **LOCAL\_MODULE\_PATH**

指定模块编译产物的输出路径，都是 `out` 目录下的子目录。如果 `Android.mk` 中没有指定该值，系统会根据“`LOCAL_MODULE_CLASS`”的值来生成。“`LOCAL_MODULE_PATH`”赋值主要应用在“`BUILD_PREBUILT`”模块的编译上，其他情况尽量采用默认值。常用值包括：

- `TARGET_OUT`：/system 目录
- `TARGET_OUT_ETC`：/system/etc 目录
- `TARGET_OUT_VENDOR`：/vendor 目录
- `TARGET_OUT_DATA`：/data 目录

以上目录均在 `build/core/envsetup.mk` 中有定义。

- **LOCAL\_MODULE\_CLASS**

模块编译类型，用于定制 `LOCAL_MODULE_PATH` 的路径。当模块 `include` 不同编译类型选项时，系统会默认指定当前的 `LOCAL_MODULE_CLASS` 的值。但 `include BUILD_PREBUILT` 编译选项时，需要明确指定 `LOCAL_MODULE_CLASS` 的值，帮助系统确定 `LOCAL_MODULE_PATH` 的值。如果不指定 `LOCAL_MODULE_CLASS`，编译产物不会放到系统中，会放在最后的 `obj` 目录下的对应目录。常用值包括：

- `APPS`：system/app 目录
- `SHARED_LIBRARIES`：system/lib 目录
- `ETC`：system/etc 目录

- **LOCAL\_MODULE\_TAGS**

模块标签，表明该模块在什么版本下才编译。常用值包括：

- **eng**: 工程师版本，用于开发调试
- **user**: 用户发布版本
- **userdebug**: 用户调试版本
- **test**: 测试版本
- **optional**: 所有版本，默认值

- **LOCAL\_CERTIFICATE**

模块签名方式，常用值包括：

- **testkey**: 非 user 版本默认签名
- **releasekey**: user 版本默认签名
- **platform**: 平台的核心应用签名，使用该签名的 apk 需要获取 platform signature，例如 Settings
- **shared**: 使用该签名方式的 apk 需要和 home/contacts 进程共享数据，例如 Launcher
- **media**: 使用该签名方式的 apk 是 media/download 系统中的一环，例如 Gallery
- **PERSIGNED**: 源 apk 自带的签名，编译过程不再重新签名。

- **LOCAL\_SHARED\_LIBRARIES**

模块在编译时依赖的动态库。模块的外部依赖一般包括：

- **LOCAL\_STATIC\_LIBRARIES**: 模块所依赖的静态库
- **LOCAL\_STATIC\_JAVA\_LIBRARIES**: 模块所依赖的 Java 静态库（jar 包等）
- **LOCAL\_JAVA\_LIBRARIES**: 模块所依赖的 Java 动态（共享）库

- **LOCAL\_MULTILIB**

使用 **LOCAL\_MULTILIB** 变量可以配置要编译的 arch 架构。若不指定，系统根据模块类型和其他 **LOCAL\_XXX** 变量，决定构建的架构，如 **LOCAL\_MODULE\_TARGET\_ARCH**。常用值包括：

- **both**: 同时构建 32 位和 64 位架构
- **32**: 仅构建 32 位架构
- **64**: 仅构建 64 位架构
- **first**: 仅构建第一个架构（在 32 位设备中构建 32 位架构，在 64 位设备中构建 64 位架构）

- **include \$(BUILD\_PREBUILT)**

表示该模块的编译类型，它指向一个 NDK（Native Development Kit，Native 开发工具集）的默认脚本，会收集从上次调用 **include \$(CLEAR\_VARS)** 后所有定义的 **LOCAL\_XXX** 变量，然后根据它们定义模块的目标、依赖关系、编译命令和编译参数等。常用值包括：

- **BUILD\_PREBUILT**: 编译成一个预置程序
- **BUILD\_PACKAGE**: 编译成一个 apk 或者资源包文件
- **BUILD\_JAVA\_LIARARY**: 编译成一个 Java 共享库
- **BUILD\_STATIC\_JAVA\_LIARARY**: 编译成一个 Java 静态库
- **BUILD\_EXECUTABLE**: 编译成一个可执行文件
- **BUILD\_SHARED\_LIARARY**: 编译成一个 native 共享库，前缀为 lib，后缀为 .so

**注意**

不同编译类型支持的宏变量有差异。例如，BUILD\_PACKAGE 类型中指定 LOCAL\_PACKAGE\_NAME 为模块名称，而 BUILD\_PREBUILT 类型中指定 LOCAL\_MODULE 为模块名称。

### 1.3.2 Android.bp

Android.bp 定义一个模块类型，模块中包含的属性示例如下。

```
android_app_import {  
    name: "myapp",  
    src: "app/myapp.apk",  
    uses_libs: ["liba", "libc"],  
    compile_multilib: "64",  
    vendor: true,  
    certificate: "platform",  
}
```

- **android\_app\_import**  
模块编译类型，类似 Android.mk 中的 “BUILD\_XXX”。
- **name**  
模块名称，唯一且必须存在，类似 Android.mk 中的 “LOCAL\_MODULE”。
- **src**  
模块源文件，类似 Android.mk 中的 “LOCAL\_SRC\_FILES”。
- **uses\_libs**  
模块在编译时依赖的动态库列表，类似 Android.mk 中的 “LOCAL\_USES\_LIBRARIES”。
- **compile\_multilib**  
使用 compile\_multilib 变量控制此模块是为 32 位、64 位还是两者都编译。常用值包括：
  - both: 同时构建 32 位和 64 位架构
  - 32: 仅构建 32 位架构
  - 64: 仅构建 64 位架构
  - first: 仅构建第一个架构
- **vendor**  
当 vendor 设置为 true 时，安装该模块至 /vendor 目录，如果 vendor 分区不存在，则安装到 /system/vendor 目录。
- **certificate**  
模块签名方式，类似 Android.mk 中的 “LOCAL\_CERTIFICATE”。

## 1.4 常见需求

对于预置应用的常见需求，主要修改预置应用的安装位置。

- Android.mk 对应的是 LOCAL\_MODULE\_PATH，不同位置对应的安装策略不同。
- Android.bp 默认 system 分区，其他分区及属性配置请参见表 2-1。

### 1.4.1 预置应用可卸载

preloadapp 和 vital-app 都是第三方应用预置目录。其目录下应用都可卸载，恢复出厂设置后可恢复。这种预编译规则目前只适合 Android.mk，Android.bp 暂时无法支持。

```
LOCAL_MODULE_PATH := $(TARGET_OUT)/preloadapp //预置到system/preloadapp
LOCAL_MODULE_PATH := $(TARGET_OUT)/vital-app //预置到system/vital-app
```

### 1.4.2 预置应用不可卸载

可预置到 system、vendor、system\_ext、product、odm、oem 等分区。示例如下：

```
LOCAL_MODULE_PATH := $(TARGET_OUT)/app //预置到system/app
LOCAL_MODULE_PATH := $(TARGET_OUT)/priv-app //预置到/system/priv-app
LOCAL_MODULE_PATH := $(TARGET_OUT_VENDOR)/app //预置到vendor/app
```

#### 说明

相对于 system 分区，CTS 会检查 vendor 分区中 apk 使用的 API 是否合规。

### 1.4.3 预置应用可升级

如果预置应用要求可升级，应确保应用 apk 文件可获取预置时的 apk 签名，否则无法升级。对于第三方应用，预置时应保留原来签名。

- Android.mk 设置  
 LOCAL\_CERTIFICATE := PRESIGNED
- Android.bp 设置  
 presigned: true,

### 1.4.4 apk odex 化

apk 进行 odex 化指在编译时提取出 apk 包中的 classes.dex 进行优化，生成.odex 文件删除并替换原 apk 包中的 classes.dex。

#### 说明

odex 化可加快设备开机速度。

- Android.mk  
 使用 LOCAL\_DEX\_PREOPT 参数可使能或禁止预置 apk 进行 odex 化。  
 LOCAL\_DEX\_PREOPT := false //false表示禁止对该apk进行odex化
- Android.bp  
 配置如下参数：  
 dex\_preopt: {  
 enabled: false, //false表示禁止对该apk进行odex化，默认是true  
 },



## 1.4.5 预置应用替换

- Android.mk

添加 LOCAL\_OVERRIDES\_PACKAGES 指定需替换的 apk 名，被替换的 apk 不加入编译。示例如下：

```
# GooglePackageInstaller
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := GooglePackageInstaller
LOCAL_MODULE_CLASS := APPS
LOCAL_MODULE_TAGS := optional
LOCAL_BUILT_MODULE_STEM := package.apk
LOCAL_MODULE_SUFFIX := $(COMMON_ANDROID_PACKAGE_SUFFIX)
LOCAL_PRIVILEGED_MODULE := true
LOCAL_CERTIFICATE := PRESIGNED
LOCAL_OVERRIDES_PACKAGES := PackageInstaller      //覆盖系统自带PackageInstaller
LOCAL_SRC_FILES := $(LOCAL_MODULE).apk
include $(BUILD_PREBUILT)
```

- Android.bp

添加如下参数。

```
Overrides: [ "PackageInstaller" ]
```



# 2 预置操作

## 2.1 预置有源码的应用

### 1. 工程中加入源码和 Android.bp。

将源代码拷贝到/vendor/sprd/platform/packages/apps/目录下，并在源代码中添加 Android.bp 文件。

```
android_app {
    name: "Stk", //模块名
    libs: ["telephony-common"], //编译依赖的Java库
    static_libs: [
        "com.google.android.material_material",
        "androidx.legacy_legacy-support-core-utils",
    ], //编译依赖的Java静态库
    srcs: ["**/*.java"],
    platform_apis: true,
    certificate: "platform", //签名类型
}
```

上述编译规则中，没有配置安装路径，那么默认应用的安装路径为 system/app。若要安装到其他分区，请参见表 2-1 进行配置。

表2-1 分区及属性配置

分区	属性配置
system/priv-app	privileged: true
system_ext	system_ext_specific: true
product	product_specific: true
vendor	vendor: true

对于有 JNI（Java Native Interface，Java 原生接口）源码的应用，需要在源码的 JNI 目录中添加编译 JNI 的规则 Android.bp，通过 jni\_libs 指定生成的 JNI 库和对 so 文件进行编译。示例如下：

#### – Android.bp 中编译 JNI 库 libjni\_validationtools

```
cc_library_shared {
    name: "libjni_validationtools", //jni 库模块名
    compile_multilib: "first",
```

```

srcs: [
    "src/jniutils.cpp",
],
shared_libs: [//依赖的共享库
    "libcutils",
    "libutils",
    "liblog",
    "libandroid_runtime",
],
}

```

- Android.bp 中使用 JNI 编译生成的 so 库 libjni\_validationtools
- ```
jni_libs: ["libjni_validationtools"],
```

## 2. 在对应 board 中加入应用名。

将应用 LOCAL\_PACKAGE\_NAME 加入到 PRODUCT\_PACKAGES 中，便于编译。存在以下两种情况，以 SC9863A 为例进行说明。

- 如果单个工程需要，则加入对应工程目录下的 mk 中。  
如 SC9863A 目录下仅 s9863a1h10\_go\_32b\_general 需要 Stk，则加入到 device/sprd/sharkl3/s9863a1h10\_go\_32b/product/s9863a1h10\_go\_32b\_general/var.mk 中。  
`PRODUCT_PACKAGES += Stk`
- 如果所有工程都需要，则加入对应模块的公共目录。  
如其他项目也需要 Stk，则加入到 device/sprd/mpool/module/app/main.mk 中。  
`PRODUCT_PACKAGES += Stk`

## 2.2 预置无共享库的应用

### 1. 创建存放 apk 文件的目录以及编写对应的 Android.mk/Android.bp。示例如下：

- a 创建 prebuilt\_apps 目录（如已存在请跳过此步骤）。  
/vendor/sprd/partner/prebuilt\_apps
- b 创建用于放置第三方 apk 的目录。  
/vendor/sprd/partner/prebuilt\_apps/weixin/
- c 创建编译预置 apk 的 Android.mk/Android.bp 文件。
  - Android.mk  
/vendor/sprd/partner/prebuilt\_apps/weixin/Android.mk
  - Android.bp  
/vendor/sprd/partner/prebuilt\_apps/weixin/Android.bp

### 2. 放置 apk 文件。

将预置应用的 apk 文件（如 weixin.apk），放到如下目录。

/vendor/sprd/partner/prebuilt\_apps/weixin

### 3. 修改 Android.mk/Android.bp 文件。

- 增加编译参数（以 weixin.apk 为例）

- Android.mk

路径：/vendor/sprd/partner/prebuilt\_apps/weixin/Android.mk

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := weixin                //module名字
LOCAL_MODULE_CLASS := APPS             //该预置为预置apk
LOCAL_CERTIFICATE := PRESIGNED         //签名方式
LOCAL_MODULE_PATH := $(TARGET_OUT)/app //安装位置
LOCAL_SRC_FILES := app/weixin.apk     //apk源文件位置
include $(BUILD_PREBUILT)
```

- Android.bp

路径：/vendor/sprd/partner/prebuilt\_apps/weixin/Android.bp

```
android_app_import {
    name: "weixin",
    apk: "app/weixin.apk",
    presigned: true,
}
```

#### 说明

从 Android 13 开始引入了上层 Java 库的强制检查，采用预编译方式编译时出现“manifest\_checker”错误，请参见 [3.6 manifest\\_checker 错误](#)。

- 增加编译模块（确保修改的 mk 被 include 到了编译系统中，此处以 SC9863A 为例）

- SC9863A 所有工程都需要预置某应用

修改 device/sprd/mpool/module/app/路径下的 main.mk，或添加相关应用目录和 mk 文件。

```
PRODUCT_PACKAGES += \
    FMPlayer \
+   weixin           //此处增加预置应用的LOCAL_MODULE参数，一般和应用同名
```

- 某个特定工程需要预置某应用（例如 s9863a1h10\_go\_32b\_general）

修改 device/sprd/shark13/s9863a1h10\_go\_32b/product/s9863a1h10\_go\_32b\_general/路径下的 var.mk。

```
PRODUCT_PACKAGES += \
+   weixin           //此处增加预置应用的LOCAL_MODULE参数，一般和应用同名
```

## 2.3 预置有共享库的应用

### 2.3.1 动态链接器

预置有共享库的 apk 时，涉及共享库的加载访问。目前共享库的加载访问依赖动态链接器实现。因为动态链接器解决如下两大问题。

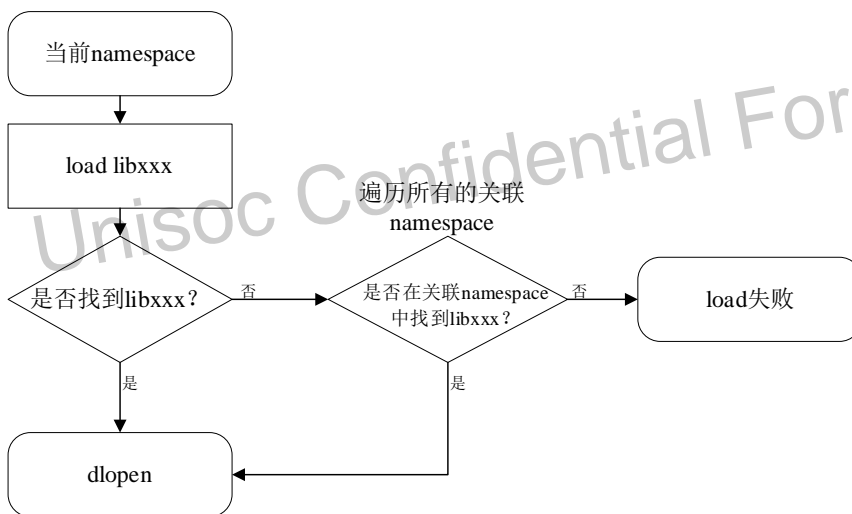
- 同名库的访问
- 库依赖加载检查

链接器的 namespace 由动态链接器提供，可隔离不同链接器 namespace 中的共享库访问。避免引用相同库名称和不同符号的库时发生冲突。例如存在以下两种情况：

- 动态链接器负责加载 DT\_NEEDED 条目中指定的共享库
- 由 dlopen()或 android\_dlopen\_ext()的参数指定的共享库

在上述两种情况下，动态链接器会找出调用方所在的链接器 namespace，并尝试将相关依赖项加载到同一个链接器 namespace 中。如果动态链接器无法将共享库加载到指定的链接器 namespace 中，它会向关联的链接器 namespace 索取导出的共享库。具体流程如图 2-1 所示。

图2-1 动态链接器加载机制



1. 在当前 namespace 中查找 libxxx。  
依次查找 searched\_paths、white\_listed、ld\_library\_paths 等目录。
2. 如果当前域中没有找到目标库，则在关联 namespace 中查找 libxxx。  
依次查找 searched\_paths、white\_listed、ld\_library\_paths 等目录。可以指定在关联的 namespace 中查找，或只在一个库文件列表中查找。如果找到 libxxx，当前进程可成功加载共享库。  
例如，namespace A 链接到 namespace B，namespace B 链接到 namespace C。如果动态链接器在 namespace A 中找不到相应的库，它仅会搜索 namespace B，而不会搜索 namespace C。

## namespace 访问路径

一个完整的 namespace 访问路径包括两部分，当前 namespace 本身可访问的路径和当前 namespace 关联域可访问的路径。实际访问路径与具体的安装位置，略有差异。例如：

- 当前创建的 namespace 名称包含了 shared，则会继承 parent namespace 的关联域，可以访问关联域可访问的路径。
- 当前创建的 namespace 名称不包含 shared，则不会将 parent namespace 的关联域添加到当前创建的应用域的关联域中，只会复制 sharedgroup 信息到当前的 namespace。

所有关联的 namespace 支持列表如表 2-2 所示。

表2-2 关联的 namespace 支持列表

| namespace                        | 可访问的共享库                                                  | 备注                          |
|----------------------------------|----------------------------------------------------------|-----------------------------|
| caller namespace (shared)        | 以 caller 进程加载的共享库所在域实际权限为准。                              | 以 caller 进程加载的共享库所在域实际权限为准。 |
| default (App 创建的 link namespace) | /etc/public.libraries.txt                                | 属于 public NDK。              |
| com.android.vndk                 | /linkerconfig/apex.libraries.config.txt (Public+JNI)     | 无                           |
| com_android_art                  | /linkerconfig/apex.libraries.config.txt (Public+JNI)     | 无                           |
| com_android_i18n                 | /linkerconfig/apex.libraries.config.txt (Public+JNI)     | 无                           |
| com_android_neuralnetworks       | /linkerconfig/apex.libraries.config.txt (Public+JNI)     | 无                           |
| com_android_appsearch            | /linkerconfig/apex.libraries.config.txt (Public+JNI)     | 无                           |
| com_android_conscrypt            | /linkerconfig/apex.libraries.config.txt (Public+JNI)     | 无                           |
| com_android_os_statsd            | /linkerconfig/apex.libraries.config.txt (Public+JNI)     | 无                           |
| com_android_tethering            | /linkerconfig/apex.libraries.config.txt (Public+JNI)     | 无                           |
| vndk/vndk_product                | /apex/com.android.vndk.v{ }/etc/vndksp.libraries.{ }.txt | 属于 vndk-sp。                 |
| sphal                            | /vendor/etc/public.libraries.txt                         | 属于 vendor 分区的 public 库。     |

如果当前应用的 namespace 为 shared 类型，parent namespace 的 links 会添加到当前 namespace 的 link namespace 中。非 shared 类型不会添加 parent namespace 的 links。

以预置应用到 system/app 为例，可访问的共享库及路径如下：

/system/\${LIB}

/system\_ext/\${LIB}

//Caller进程对应的namespace的Links可访问的路径，可在/linkerconfig/ld.config.txt中查看

libnativehelper.so

```
libicu18n.so:libicuuc.so:libicu.so
libneuralnetworks.so
/vendor/etc/public.libraries.txt
```

预置到 system/app 的应用创建的 namespace 属于 shared，如果动态加载的目标被其他共享 namespace 加载，则可直接使用。共享特性的 namespace，可加载 parent namespace 可访问的路径下的目标库。

```
M0D621B 07-13 04:07:44.801 26928 26928 D linker : dlopen(name="libview-right-web-client-wrap.so",
flags=0x0, extinfo=[flags=0x200, reserved_addr=0x0, reserved_size=0x0, relro_fd=0, library_fd=0,
library_fd_offset=0x0, library_namespace=classloader-namespace-shared @0xb084d6d0],
caller="/apex/com.android.art/lib/libnativeloader.so", caller_ns=com_android_art@0xb084d0d0,
targetSdkVersion=29) ...
```

- libview-right-web-client-wrap.so: 需要加载的目标库。
- classloader-namespace-shared: App 进程的 library\_namespace。
- com\_android\_art@0xb084d0d0: caller 进程的 namespace。

若 caller namespace 无法查询到当前需要访问共享库的信息，则将默认的 parent namespace 设置为 default。否则将 parent namespace 设置为 caller 进程的 namespace。

## 查看当前 parent namespace 对应的权限

通过查看/linkerconfig/ld.config.txt 查看当前 parent namespace 对应的权限。如当前的 parent namespace default 对应的 linked namespace 如下。详细 namespace 访问权限见 [2.3.2 namespace 访问权限](#)。

```
namespace.default.links =
com_android_adbd,com_android_i18n,com_android_art,com_android_resolve,com_android_neuralnetworks,com_
android_os_statsd
```

## 2.3.2 namespace 访问权限

VNDK（Vendor Native Development Kit，供应商原生开发套件）是一组专门用于供应商实现其 HAL 的 lib 库。每一个应用进程都会创建一个对应的域，动态链接器会根据分区的特性以及 VNDK 的要求，动态生成当前域的访问规则。应用进程运行时，会根据当前的配置查找并加载目标共享库，具体如[表 2-3](#)、[表 2-4](#)、[表 2-5](#) 所示。

表2-3 system/system\_ext app 可访问权限

| 共享库的安装位置                | system/app(priv-app)                             | system_ext/app(priv-app)                         |
|-------------------------|--------------------------------------------------|--------------------------------------------------|
| /system/lib[64]         | All                                              | All                                              |
| /system_ext/lib[64]     | All                                              | All                                              |
| /system/product/lib[64] | 不可访问                                             | 不可访问。                                            |
| /product/lib[64]        | etc/public.libraries.*.txt (扩展的公共库)              | etc/public.libraries.*.txt (扩展的公共库)。             |
| /vendor/lib[64]         | vendor/etc/public.libraries.txt                  | vendor/etc/public.libraries.txt                  |
| /apex/com.android.vndk/ | /linkerconfig/apex.libraries.config.txt (Public) | /linkerconfig/apex.libraries.config.txt (Public) |
| /apex/com_android_art   | /linkerconfig/apex.libraries.config.txt          | /linkerconfig/apex.libraries.config.txt          |

| 共享库的安装位置                             | system/app(priv-app)                                | system_ext/app(priv-app)                            |
|--------------------------------------|-----------------------------------------------------|-----------------------------------------------------|
|                                      | (Public)                                            | (Public)                                            |
| /apex/com_android_i18n               | /linkerconfig/apex.libraries.config.txt<br>(Public) | /linkerconfig/apex.libraries.config.txt<br>(Public) |
| /apex/com_android_neuralnet<br>works | /linkerconfig/apex.libraries.config.txt<br>(Public) | /linkerconfig/apex.libraries.config.txt<br>(Public) |
| /apex/com_android_appsearch          | /linkerconfig/apex.libraries.config.txt<br>(JNI)    | /linkerconfig/apex.libraries.config.txt<br>(JNI)    |
| /apex/com_android_conscrypt          | /linkerconfig/apex.libraries.config.txt<br>(JNI)    | /linkerconfig/apex.libraries.config.txt<br>(JNI)    |
| /apex/com_android_os_statsd          | /linkerconfig/apex.libraries.config.txt<br>(JNI)    | /linkerconfig/apex.libraries.config.txt<br>(JNI)    |
| /apex/com_android_tethering          | /linkerconfig/apex.libraries.config.txt<br>(JNI)    | /linkerconfig/apex.libraries.config.txt<br>(JNI)    |

表2-4 product/vendor app 可访问权限

| 共享库的安装位置                             | product/app(priv-app)                                                                                                                                                                                                                                                                                   | vendor/app(priv-app)                                                                                                                                                                                                                        |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| /system/lib[64]                      | etc/public.libraries.txt (public NDK)                                                                                                                                                                                                                                                                   | etc/public.libraries.txt (public NDK)                                                                                                                                                                                                       |
| /system_ext/lib[64]                  | 不可访问                                                                                                                                                                                                                                                                                                    | 不可访问。                                                                                                                                                                                                                                       |
| /system/product/lib[64]              | All                                                                                                                                                                                                                                                                                                     | 不可访问。                                                                                                                                                                                                                                       |
| /product/lib[64]                     | All                                                                                                                                                                                                                                                                                                     | 不可访问。                                                                                                                                                                                                                                       |
| /vendor/lib[64]                      | vendor/etc/public.libraries.txt                                                                                                                                                                                                                                                                         | All                                                                                                                                                                                                                                         |
| /apex/com.android.vndk/              | <ul style="list-style-type: none"> <li>• /linkerconfig/apex.libraries.config.txt (Public)</li> <li>• /linkerconfig/apex.libraries.config.txt (Public)</li> <li>• /apex/com.android.vndk.v{ }/etc/llndk.libraries.{ }.txt</li> <li>• /apex/com.android.vndk.v{ }/etc/vndksp.libraries.{ }.txt</li> </ul> | <ul style="list-style-type: none"> <li>• /linkerconfig/apex.libraries.config.txt (Public)</li> <li>• /apex/com.android.vndk.v{ }/etc/llndk.libraries.{ }.txt</li> <li>• /apex/com.android.vndk.v{ }/etc/vndksp.libraries.{ }.txt</li> </ul> |
| /apex/com_android_art                | /linkerconfig/apex.libraries.config.txt<br>(Public)                                                                                                                                                                                                                                                     | /linkerconfig/apex.libraries.config.txt<br>(Public)                                                                                                                                                                                         |
| /apex/com_android_i18n               | /linkerconfig/apex.libraries.config.txt<br>(Public)                                                                                                                                                                                                                                                     | /linkerconfig/apex.libraries.config.txt<br>(Public)                                                                                                                                                                                         |
| /apex/com_android_neuralnet<br>works | /linkerconfig/apex.libraries.config.txt<br>(Public)                                                                                                                                                                                                                                                     | /linkerconfig/apex.libraries.config.txt<br>(Public)                                                                                                                                                                                         |
| /apex/com_android_appsearch          | /linkerconfig/apex.libraries.config.txt<br>(JNI)                                                                                                                                                                                                                                                        | /linkerconfig/apex.libraries.config.txt<br>(JNI)                                                                                                                                                                                            |



| 共享库的安装位置                    | product/app(priv-app)                         | vendor/app(priv-app)                          |
|-----------------------------|-----------------------------------------------|-----------------------------------------------|
| /apex/com_android_conscrypt | /linkerconfig/apex.libraries.config.txt (JNI) | /linkerconfig/apex.libraries.config.txt (JNI) |
| /apex/com_android_os_statsd | /linkerconfig/apex.libraries.config.txt (JNI) | /linkerconfig/apex.libraries.config.txt (JNI) |
| /apex/com_android_tethering | /linkerconfig/apex.libraries.config.txt (JNI) | /linkerconfig/apex.libraries.config.txt (JNI) |

表2-5 preloadapp app 可访问权限

| 共享库的安装位置                         | system/preloadapp(vital-app)                     |
|----------------------------------|--------------------------------------------------|
| /system/lib[64]                  | etc/public.libraries.txt (public NDK)            |
| /system_ext/lib[64]              | etc/public.libraries.*.txt (扩展的公共库)              |
| /system/product/lib[64]          | 不可访问                                             |
| /product/lib[64]                 | etc/public.libraries.*.txt (扩展的公共库)              |
| /vendor/lib[64]                  | vendor/etc/public.libraries.txt                  |
| /apex/com.android.vndk/          | /linkerconfig/apex.libraries.config.txt (Public) |
| /apex/com_android_art            | /linkerconfig/apex.libraries.config.txt (Public) |
| /apex/com_android_i18n           | /linkerconfig/apex.libraries.config.txt (Public) |
| /apex/com_android_neuralnetworks | /linkerconfig/apex.libraries.config.txt (Public) |
| /apex/com_android_appsearch      | /linkerconfig/apex.libraries.config.txt (JNI)    |
| /apex/com_android_conscrypt      | /linkerconfig/apex.libraries.config.txt (JNI)    |
| /apex/com_android_os_statsd      | /linkerconfig/apex.libraries.config.txt (JNI)    |
| /apex/com_android_tethering      | /linkerconfig/apex.libraries.config.txt (JNI)    |

### 2.3.2.1 system 和 system\_ext 分区

本节分别介绍了 **system\_ext 分区**和 **system 分区**，罗列出 **system 和 system\_ext 分区访问控制权限**，并以预置 Wink.apk 为例，给出**示例**。

#### system\_ext 分区

如果预置应用的动态库与系统路径下的共享库同名，则无法预置到 system(system\_ext)/app(priv-app)路径下，需预置到其他分区或者目录。

- 预置应用

预置应用到 system(system\_ext)/app(priv-app)路径下时，对应的 namespace classloader-namespace-shared 为 shared 类型。



- 将目标应用依赖的共享库预置到 `system/lib(64)`、`system_ext/lib(64)`、`/system/app(priv-app)/XXX/lib/arm(64)` 路径下，应用都可正常加载目标共享库。
- `/system/priv-app/XXX/XXX.apk!/lib/armeabi-v7a` 为应用加载内存中的虚拟路径，无法直接预置。
- 查找顺序
 

预置应用到 `system(system_ext)/app(priv-app)` 路径下时，查找顺序如下：

  - a 共享库加载时优先查找系统库路径 `system/lib(64)`、`system_ext/lib(64)`。  
此路径下的共享库对其他应用进程可见，存在安全隐患。
  - b 查找 `/system/app(priv-app)/XXX/lib/arm(64)`。  
建议将应用依赖的共享库预置到 `/system/app(priv-app)/XXX/lib/arm(64)` 路径下。

## system 分区

应用是否有与系统共享库同名库，都可预置到 `system` 路径下。但预置到此路径下的应用，可卸载，恢复出厂设置后可恢复。

- 预置应用
 

预置应用到 `system/preloadapp(vital-app)` 路径下时（针对第三方应用预置），对应的 `namespace classloader-namespace` 为非 `shared` 类型。

预置到 `system` 路径下的目标应用（`apk` 包含共享库），系统会自动提取 `apk` 中的共享库，并安装到 `data/app-lib/XXX`。
- 查找顺序
 

优先查找 `data/app-lib/XXX`。

## system 和 system\_ext 分区访问控制权限

`system` 和 `system_ext` 分区具体的访问控制权限如表 2-6 所示。

表2-6 system 及 system\_ext 分区可访问权限

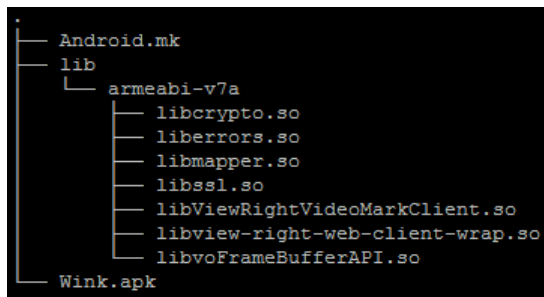
| 预置路径                         | namespace                                 | 访问路径                                                                                                                                                                                                                                                                                                                                                                                                        | 共享库预置路径建议                                     |
|------------------------------|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------|
| <code>system/priv-app</code> | <code>classloader-namespace-shared</code> | <ul style="list-style-type: none"> <li>• <code>/system/priv-app/XXX/lib/arm</code></li> <li>• <code>/system/priv-app/XXX/XXX.apk!/lib/armeabi-v7a</code></li> <li>• <code>/system/lib</code></li> <li>• <code>/system_ext/lib</code></li> <li>• <code>/vendor/etc/public.libraries.txt</code> 定义的库</li> <li>• 特定 apex 库列表：<br/><code>/linkerconfig/apex.libraries.config.txt</code> (Public+JNI)</li> </ul> | <code>/system/priv-app/XXX/lib/arm(64)</code> |
| <code>system/app</code>      | <code>classloader-namespace-shared</code> | <ul style="list-style-type: none"> <li>• <code>/system/app/XXX/lib/arm</code></li> <li>• <code>/system/app/XXX/XXX.apk!/lib/armeabi-v7a</code></li> <li>• <code>/system/lib</code></li> <li>• <code>/system_ext/lib</code></li> </ul>                                                                                                                                                                       | <code>/system/app/XXX/lib/arm(64)</code>      |

| 预置路径                | namespace                    | 访问路径                                                                                                                                                                                                                                                                                                              | 共享库预置路径建议                             |
|---------------------|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------|
|                     |                              | <ul style="list-style-type: none"> <li>/vendor/etc/public.libraries.txt 定义的库</li> <li>特定 apex 库列表：<br/>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul>                                                                                                                                            |                                       |
| system_ext/priv-app | classloader-namespace-shared | <ul style="list-style-type: none"> <li>/system/priv-app/XXX/lib/arm</li> <li>/system/priv-app/XXX/XXX.apk!/lib/armeabi-v7a</li> <li>/system/lib</li> <li>/system_ext/lib</li> <li>/vendor/etc/public.libraries.txt 定义的库</li> <li>特定 apex 库列表：<br/>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul> | /system_ext/priv-app/XXX/lib/arm(64)  |
| system_ext/app      | classloader-namespace-shared | <ul style="list-style-type: none"> <li>/system/app/XXX/lib/arm</li> <li>/system/app/XXX/XXX.apk!/lib/armeabi-v7a</li> <li>/system/lib</li> <li>/system_ext/lib</li> <li>/vendor/etc/public.libraries.txt 定义的库</li> <li>特定 apex 库列表：<br/>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul>           | /system_ext/app/XXX/lib/arm(64)       |
| system/preloadapp   | classloader-namespace        | <ul style="list-style-type: none"> <li>/data/app-lib/XXX</li> <li>/system/preloadapp/XXX/XXX.apk!/lib/armeabi-v7a</li> <li>/system/etc/public.libraries.txt (NDK)</li> <li>/vendor/etc/public.libraries.txt</li> <li>特定 apex 库列表：<br/>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul>             | 不建议单独预置共享库，如果 apk 压缩文件中包含共享库，系统会自行安装。 |
| system/vital-app    | classloader-namespace        | <ul style="list-style-type: none"> <li>/data/app-lib/XXX</li> <li>/system/preloadapp/XXX/XXX.apk!/lib/armeabi-v7a</li> <li>/system/etc/public.libraries.txt (NDK)</li> <li>/vendor/etc/public.libraries.txt</li> <li>特定 apex 库列表：<br/>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul>             | 不建议单独预置共享库，如果 apk 压缩文件中包含共享库，系统会自行安装。 |

## 示例

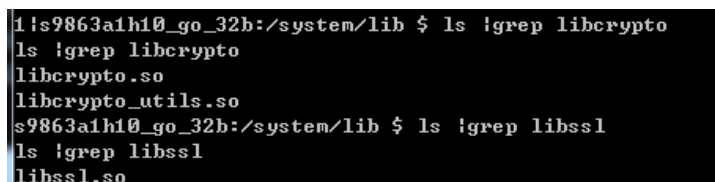
以预置 Wink.apk 为例进行说明，预置源文件清单如图 2-2 所示。

图2-2 预置源文件清单



当前有 apk 依赖与系统库同名的两个库 libcrypto 和 libssl，如图 2-3 所示。

图2-3 同名系统库说明



上述共享库存在于设备的 system/lib(64)路径下。如果预置 Wink.apk 到 system(system\_ext)/app(priv-app)下，此时应用对应的 namespace 为 classloader-namespace-shared。其他应用进程加载同名库会影响 Wink.apk 进程的使用。

具体分场景讨论：

- 如果同名库与应用兼容，且升级后也兼容，则无需单独预置 libcrypto 和 libssl 这两个库，使用系统库即可。其他共享库可以预置到 system/lib(64)、system\_ext/lib(64)、/system/app(priv-app)/XXX/lib/arm(64)下。
- 如果同名库与应用不兼容，需要单独预置 libcrypto 和 libssl 这两个库。此应用无法预置到 system(system\_ext)/app(priv-app)。

### 2.3.2.2 product 分区

#### 预置应用

预置应用到 product/app(priv-app)路径下，应用进程对应的 namespace 为 vendor-classloader-namespace。

将应用依赖的共享库预置到/product/app(priv-app)/XXX/lib/arm(64)、product/lib(64)、system/product/lib(64)。

#### 查找顺序

共享库加载时，优先查找系统库路径 product/lib(64)、system/product/lib(64)。此路径下的共享库对其他进程也可见，存在安全隐患。建议将共享库预置到/product/app(priv-app)/XXX/lib/arm。

当前 namespace 不是 shared 类型，其他进程加载同名库，不影响当前应用的加载。只要预置的目标路径下不包含同名库，这些路径都可以预置依赖的共享库。

## product 分区访问控制权限

product 分区具体的访问控制权限如表 2-7 所示。

表2-7 product 分区可访问权限

| 预置路径             | namespace                    | 访问路径                                                                                                                                                                                                                                                                                                                                                                              | 共享库预置路径建议                         |
|------------------|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| product/priv-app | vendor-classloader-namespace | <ul style="list-style-type: none"> <li>• /product/priv-app/XXX/lib/arm</li> <li>• /product/priv-app/XXX/XXX.apk!/lib/armeabi-v7a</li> <li>• /product/lib</li> <li>• /system/product/lib</li> <li>• /system/etc/public.libraries.txt (NDK)</li> <li>• /vendor/etc/public.libraries.txt</li> <li>• 特定 apex 库列表：<br/>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul> | /product/priv-app/XXX/lib/arm(64) |
| product/app      | vendor-classloader-namespace | <ul style="list-style-type: none"> <li>• /product/app/XXX/lib/arm</li> <li>• /product/app/XXX/XXX.apk!/lib/armeabi-v7a</li> <li>• /product/lib</li> <li>• /system/product/lib</li> <li>• /system/etc/public.libraries.txt (NDK)</li> <li>• /vendor/etc/public.libraries.txt</li> <li>• 特定 apex 库列表：<br/>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul>           | /product/app/XXX/lib/arm(64)      |

### 2.3.2.3 vendor 分区

#### 预置应用

预置应用到 vendor/app(priv-app)路径下，应用进程对应的 namespace 为 vendor-classloader-namespace。应用依赖的共享库可以预置到/vendor/app(priv-app)/XXX/lib/arm(64)、/vendor/lib(64)。

#### 查找顺序

库加载时优先查找系统库路径/vendor/lib(64)，此路径下的共享库对其他进程也可见，有安全隐患。建议将共享库预置到/vendor/app(priv-app)/XXX/lib/arm 路径下。

当前 namespace 不是 shared 类型，其他进程加载同名库，不影响当前 App 的加载。只要预置的目标路径下不包含同名库，这些路径都可以预置依赖的共享库（与 product 分区类似）。

## vendor 分区访问控制权限

vendor 分区具体的访问控制权限如表 2-8 所示。

表2-8 vendor 分区可访问权限

| 预置路径            | namespace                    | 访问路径                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | 共享库预置路径建议                        |
|-----------------|------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| vendor/priv-app | vendor-classloader-namespace | <ul style="list-style-type: none"> <li>• /vendor/priv-app/XXX/lib/arm</li> <li>• /vendor/priv-app/XXX/XXX.apk!/lib/armeabi-v7a</li> <li>• /vendor/lib</li> <li>• 特定的 system 共享库列表： <ul style="list-style-type: none"> <li>- /system/etc/public.libraries.txt (NDK)</li> <li>- + LLNDK</li> <li>- /apex/com.android.vndk.v{}/etc/vndksp.libraries.{}.txt</li> </ul> </li> <li>• 特定 apex 库列表： <ul style="list-style-type: none"> <li>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul> </li> </ul> | /vendor/priv-app/XXX/lib/arm(64) |
| vendor/app      | vendor-classloader-namespace | <ul style="list-style-type: none"> <li>• /vendor/app/XXX/lib/arm</li> <li>• /vendor/app/XXX/XXX.apk!/lib/armeabi-v7a</li> <li>• /vendor/lib</li> <li>• 特定的 system 共享库列表： <ul style="list-style-type: none"> <li>- /system/etc/public.libraries.txt (NDK)</li> <li>- + LLNDK</li> <li>- /apex/com.android.vndk.v{}/etc/vndksp.libraries.{}.txt</li> </ul> </li> <li>• 特定 apex 库列表： <ul style="list-style-type: none"> <li>/linkerconfig/apex.libraries.config.txt (Public+JNI)</li> </ul> </li> </ul>           | /vendor/app/XXX/lib/arm(64)      |

## 2.3.3 预置应用带有 so 库

对于带有 so 库的应用，可通过预编译和复制两种方式预置。从 Android 13 开始引入了上层 Java 库的强制检查，任意编译方式编译时出现“manifest\_checker”错误，请参见 3.6 manifest\_checker 错误。

### 2.3.3.1 预编译方式

建议使用 Android.mk 编译规则，因为 Android.bp 无法安装到自定义的目录。

- 预置非系统 apk 时，无需处理 so 库文件，安装时系统会自动处理。
- 预置系统 apk 时，需手动解压 apk，将其中的 so 库文件预置到特定位置。

第三方应用中可能同时支持多种类型的 so，如 x86\_64、x86、armeabi-v7a、armeabi、arm64-v8a。建议根据当前项目 board 中 CPU 架构相关参数配置选择匹配的 so 解压、预置。

## 示例

如共享库中已经存在一个同名的 so 库文件，建议预置成应用 so，非共享库会被打包到对应应用目录下。

```
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := weixin
LOCAL_MODULE_CLASS := APPS
LOCAL_CERTIFICATE := PRESIGNED           //不重新签名
LOCAL_MODULE_PATH := $(TARGET_OUT)/app   //预置位置
LOCAL_SRC_FILES := app/weixin.apk        //apk文件位置
LOCAL_PREBUILT_JNI_LIBS := lib/libFFmpeg.so \ //列出所有apk so库文件
lib/ libfingerprintauth.so
include $(BUILD_PREBUILT)
```

### 2.3.3.2 复制方式

在 device/sprd 目录对应工程里面添加以下内容。

```
PRODUCT_COPY_FILES += $(LOCAL_PATH)/libapp.so:system/lib64/libapp.so
```

- libapp.so:system: src 目录
- lib64/libapp.so: dest 目录

从 Android 13 开始，预编译 ELF 文件时，不允许使用 PRODUCT\_COPY\_FILES。如果想用复制的方式，必须把添加的内容放到 board 中去。

## 示例

使用 shell 命令 “cp” 进行复制。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := Wink
LOCAL_MODULE_CLASS := APPS
LOCAL_CERTIFICATE := PRESIGNED
LOCAL_MODULE_PATH := $(TARGET_OUT)/app
LOCAL_SRC_FILES := Wink.apk
$(shell mkdir -p $(TARGET_OUT)/app/Wink/lib/arm/) //使用shell命令复制共享库
$(shell cp -rf $(LOCAL_PATH)/lib/armeabi-v7a $(TARGET_OUT)/Wink/lib/arm/)
include $(BUILD_PREBUILT)
```

## 2.4 预置可执行程序

### 1. 编写对应的 Android.mk/Android.bp 文件。

#### – Android.mk

以 vendor/sprd/partner/brcm/wl/Android.mk 为例。

```
LOCAL_PATH := $(call my-dir)
ifeq ($(strip $(BOARD_WLAN_DEVICE)),bcmhdh)
include $(CLEAR_VARS)
LOCAL_MODULE := wl
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_CLASS := EXECUTABLES //可执行程序
LOCAL_MODULE_PATH := $(TARGET_OUT)/bin
LOCAL_SRC_FILES := $(LOCAL_MODULE) //可执行程序位置
include $(BUILD_PREBUILT)
endif
```

#### – Android.bp

Android.bp 中指定 cc\_prebuilt\_binary 为预置可执行文件库的模块类型。

```
cc_prebuilt_binary {
    name: "mybinary",
    srcs: ["bin/mybinary"],
    shared_libs: ["liba", "libc"]
}
```

#### 📖 说明

“shared\_libs”表示该模块所依赖的外部共享库，类似 [1.3.1 Android.mk](#) 中的“LOCAL\_SHARED\_LIBRARIES”。

### 2. 将生成的模块加入编译。

将 LOCAL\_MODULE 添加到 PRODUCT\_PACKAGES 中，请参见 [2 在对应 board 中加入应用名](#)。

## 2.5 预置 native service

### 1. 生成 bin 文件。

创建目录，编写对应的 Android.mk 文件，编译生成 native service 对应的 bin 文件，init 启动进程执行该文件。

以 vendor/sprd/proprieties-source/slogmodem/service/Android.mk 为例。

```
include $(CLEAR_VARS)
LOCAL_INIT_RC := modemlog_connmgr_service.rc
LOCAL_MODULE := modemlog_connmgr_service //modemlog_connmgr_service bin文件
LOCAL_MODULE_TAGS := optional
```

```
LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libnetutils \
    liblog \
    libutils \
    libhidlbase \
    libsysutils \
    vendor.sprd.hardware.cplog_connmgr@1.0 \
    libhidltransport

LOCAL_SRC_FILES := service.cpp \
    ConnectControlCallback.cpp \
    hidl_server.cpp \
    modem_state_hidl_server.cpp \
    modem_time_sync_hidl_server.cpp \
    wcn_state_hidl_server.cpp
```

```
LOCAL_CFLAGS += -DLOG_TAG=\"CPLOG_CONNMGR\"
include $(BUILD_EXECUTABLE)
CUSTOM_MODULES += modemlog_connmgr_service
```

2. 将模块加入编译。  
将 service LOCAL\_MODULE 加入 PRODUCT\_PACKAGES 中。  
以 slogmodem 为例：device/sprd/mpool/module/log/slogmodem/main.mk

```
PRODUCT_PACKAGES += \
    vendor.sprd.hardware.cplog_connmgr@1.0-impl \
    vendor.sprd.hardware.cplog_connmgr@1.0 \
    vendor.sprd.hardware.cplog_connmgr@1.0-service \
    modemlog_connmgr_service //将service LOCAL_MODULE加入
```

3. 为 service 写 rc 注册文件。  
rc 文件可以让 android init 知道系统中有该 service 存在。  
以 vendor/sprd/proprieties-source/slogmodem/service/modemlog\_connmgr\_service.rc 为例。

```
service modemlog_connmgr_service /system/bin/modemlog_connmgr_service //name + bin文件具体地址
    class main
    user root
```



## 2.6 预置 so 库

预置 so 库分为以下三种场景：

- [复制到指定位置](#)
- [应用中使用 so 库](#)
- [预置为系统 so 库](#)

### 2.6.1 复制到指定位置

在 device/sprd 目录对应工程里面添加如下内容。

```
PRODUCT_COPY_FILES += $(LOCAL_PATH)/libapp.so:system/lib64/libapp.so // src目录: dest目录
```

### 2.6.2 应用中使用 so 库

LOCAL\_PREBUILT\_LIBS 指定 prebuilt so 库的 nickname 及文件路径。语法规则如下：

```
LOCAL_PREBUILT_LIBS:=nickname:path //nickname为so库的nickname，path为so库文件路径
```

- nickname 一般不可改变，特别是第三方 jar 包使用 so 库时。
- nickname 不含.so 后缀。
- so 文件路径为存放第三方 so 文件的路径。

#### 示例

以 vendor/sprd/platform/packages/apps/ValidationTools/Android.mk 为例。应用预置共享库的清单文件如 [图 2-4](#) 所示。

图2-4 应用预置共享库的清单文件

```
/A11/vendor/sprd/platform/packages/apps/ValidationTool$ tree -L 3
.
├── Android.mk
├── libs
│   └── arm64-v8a
│       └── libCameraVerification.so
```

1. 目标应用的编译规则中添加编译依赖。

```
LOCAL_REQUIRED_MODULES := libCameraVerification
```

```
LOCAL_JNI_SHARED_LIBRARIES += libCameraVerification //指定jni lib，应用中使用
```

2. 预编译为共享库的规则。

```
#预编译jni库
```

```
include $(BUILD_PACKAGE)
```

```
include $(CLEAR_VARS)
```

```
LOCAL_MULTILIB := 64
```

```
LOCAL_PREBUILT_LIBS := libCameraVerification:libs/arm64-v8a/libCameraVerification.so
```

```
include $(BUILD_MULTI_PREBUILT)
```

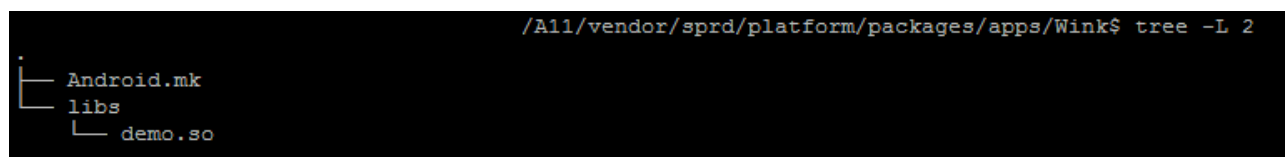
## 2.6.3 预置为系统 so 库

### Android.mk

预置系统 so 库示例如下。系统库预置的清单文件如图 2-5 所示。

```
include $(CLEAR_VARS)
LOCAL_MODULE := demo.so
LOCAL_SRC_FILES := demo.so
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_CLASS := SHARED_LIBRARIES           //指明模块类型为共享库
LOCAL_MODULE_PATH := $(TARGET_OUT)/lib/           //指明预置位置/system/lib
include $(BUILD_PREBUILT)
```

图2-5 系统库预置的清单文件



### Android.bp

Android.bp 中指定 cc\_prebuilt\_library\_shared 为预置 so 库的模块类型：

```
cc_prebuilt_library_shared {
    name: "libx",
    srcs: ["lib/libx.so"],
}
```

### 说明

编译类型 cc\_prebuilt\_library\_shared 中，有如下定义。

- “srcs”：源文件的路径，与“android\_app\_import”中的“apk”作用一致。
- “vendor: true”：编译最终的输出目录是 vendor/ 目录。

## 2.7 集成 jar 包

### 2.7.1 应用中集成 jar 包

- LOCAL\_STATIC\_JAVA\_LIBRARIES 用于指定 jar 包的 nickname，nickname 取任意值。  
LOCAL\_STATIC\_JAVA\_LIBRARIES :=nickname
- LOCAL\_PREBUILT\_STATIC\_JAVA\_LIBRARIES 指定 prebuilt jar 包的 nickname 及第三方 jar 包的路径。

`LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES :=nickname:path` //nickname为jar包的nickname, path为jar文件路径

### 说明

- nickname 一定要与 `LOCAL_STATIC_JAVA_LIBRARIES` 里所取的 nickname 一致
- nickname 不含.jar 后缀。
- jar 文件路径为存放第三方 jar 文件的路径。
- 使用 `BUILD_MULTI_PREBUILT` 编译。

### 示例

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_STATIC_JAVA_LIBRARIES := libbaidumapapi
LOCAL_SRC_FILES := $(call all-subdir-java-files)
LOCAL_PACKAGE_NAME := MyMaps
include $(BUILD_PACKAGE)

include $(CLEAR_VARS)
LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES :=libbaidumapapi:libs/baidumapapi.jar
LOCAL_PREBUILT_LIBS :=libBMapApiEngine_v1_3_1:libs/armeabi/libBMapApiEngine_v1_3_1.so
LOCAL_MODULE_TAGS := optional
include $(BUILD_MULTI_PREBUILT)

# Use the following include to make our testapk.
include $(call all-makefiles-under,$(LOCAL_PATH))
```

## 2.7.2 系统中集成 jar 包

### 2.7.2.1 应用共享类型 library

#### 1. 打包 jar 包。

生成 jar 文件并集成 jar 文件到 `system/framework` 目录中。

##### a 打包 jar 包，可编译生成 jar 文件。

```
LOCAL_PATH := $(call my-dir)
# the library
include $(CLEAR_VARS)
LOCAL_MODULE:= libandroid_user
LOCAL_MODULE_TAGS := eng
LOCAL_SRC_FILES := \
    $(call all-subdir-java-files)
```

```
include $(BUILD_JAVA_LIBRARY)
```

- b 集成 jar 文件到 system/framework 路径下。

```
include $(CLEAR_VARS)
```

```
LOCAL_MODULE := libandroid_user.jar
```

```
LOCAL_MODULE_TAGS := optional
```

```
LOCAL_MODULE_CLASS := JAVA_LIBRARIES
```

```
# This will install the file in /system/framework
```

```
LOCAL_MODULE_PATH := $(TARGET_OUT_JAVA_LIBRARIES) //预置到
/system/framework/
```

```
LOCAL_SRC_FILES := libandroid_user.jar //jar文件路径
```

```
include $(BUILD_PREBUILT)
```

device/sprd 目录对应工程 PRODUCT\_PACKAGES += 中添加模块 “LOCAL\_MODULE”，生成的系统 jar 包放在 system/framework 下。

2. 声明 jar 包。

编写 xml 文件声明 jar 包，使系统能够识别此 jar 包。

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<permissions>
```

```
<library name="android.user.library" file="/system/framework/libandroid_user.jar"/> // name是所使用jar
包的名字，file是jar包的路径
```

```
</permissions>
```

```
LOCAL_MODULE := libandroid_user.xml
```

```
LOCAL_MODULE_CLASS := ETC
```

```
LOCAL_MODULE_PATH := $(TARGET_OUT_ETC)/permissions
```

```
LOCAL_SRC_FILES := $(LOCAL_MODULE)
```

```
include $(BUILD_PREBUILT)
```

文件存放路径为 system/etc/permissions。

3. 使用 jar 包。

在所需要使用的 AndroidManifest.xml 中添加相应的引用，apk 即可使用 jar 包。

```
<uses-library android:name="android.user.library" />
```

### 2.7.2.2 系统加载类型 library

以 Android.mk 为例。如需使用 Android.bp，可通过 androidmk 开源工具将 Android.mk 转换为 Android.bp。转换步骤见 [Android.mk 转换为 Android.bp](#)。

1. 生成 jar 包。

源码 framework/opt 目录中新建文件夹 test，将 test.jar 复制到此目录，并新建立 Android.mk。

```
LOCAL_PATH := $(call my-dir)
```

```
#test.jar
```

```
include $(CLEAR_VARS)
```

```
LOCAL_MODULE := test
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_CLASS := JAVA_LIBRARIES           //jar库
LOCAL_SRC_FILES := test.jar                     //需要预置的jar文件路径
include $(BUILD_PREBUILT)
```

2. 打包并配置 jar 包。
3. device/sprd 目录对应工程中添加模块 “PRODUCT\_PACKAGES +=” 和 “PRODUCT\_BOOT\_JARS :=”，末尾分别增加 LOCAL\_MODULE 的定义值。

# The order of PRODUCT\_BOOT\_JARS matters.

```
PRODUCT_BOOT_JARS := \
    core-oj \
    core-libart \
    conscrypt \
    okhttp \
    legacy-test \
    bouncycastle \
    test                //增加module test
```

4. 将 jar 包加入白名单。  
在 build/core/tasks/check\_boot\_jars/路径下的 package\_whitelist.txt 文件末尾增加 test.jar 包名。

```
# test.jar
com\.king\.test\..*
```

## Android.mk 转换为 Android.bp

1. 使用以下命令获取 androidmk 工具。

```
cd <root of source tree>
source build/envsetup.sh
lunch <lunch-target>
m androidmk
```

2. 在根目录下，使用 androidmk 工具将 mk 文件转换为 bp 文件。

```
androidmk <path-to-Android.mk>/Android.mk >> Android.bp
```

# 3 常见问题

## 3.1 替换系统签名

Userdebug 版本的签名文件放置在 build/target/product/security 目录中，User 版本的签名文件路径从 Android 13 开始为 vendor/sprd/release/apk\_key。其中 User 版本的签名仓库为闭源发布，即释放的 IDH 包内无此仓库。需要在编译前创建 vendor/sprd/release/apk\_key 仓库并生成客制化 key 文件。创建方法参考《Android 14 IDH 包编译使用指南》。

## 3.2 签名不一致

由于签名不一致，导致应用安装失败。因此，为确保应用安装成功，需要签名保持一致。

常见签名不一致有如下两种情况：

- 与 shared user 的应用签名不一致。  
相同 shared user 的应用签名需要保持一致。如 setting 和 deskclock 两个应用的 shareduserid 都为“android.uid.system”，setting 的签名为 platform，则 deskclock 也必须是 platform。否则只有其中一个应用会安装成功。

Log 中安装失败的标记如下。

```
INSTALL_FAILED_SHARED_USER_INCOMPATIBLE
```

- 应用升级前、后签名不一致。  
应用更新时会检查前、后版本的应用签名是否一致，如果不一致将无法更新。  
Log 中安装失败的标记如下。

```
INSTALL_FAILED_UPDATE_INCOMPATIBLE
```

## 3.3 V2 签名后，应用安装失败

apk 的两种签名方案如下：

- Signature Scheme V1  
Signature Scheme V1 是 jar Signature，来自 JDK，通过 ZIP 条目进行验证。apk 签署后可进行修改，例如移动甚至重新压缩文件。
- Signature Scheme V2  
Android 7.0 中引入了 apk Signature Scheme V2。该签名验证压缩文件的所有字节，而非单个 ZIP 条目。因此，签名后无法更改（包括 zipalign）。

使用 Signature Scheme V2 签名的 apk 经过编译系统重新编译打包，会导致系统解析时无法获取到该 apk 的签名，导致安装失败。

## 【解决方法】

预置使用 Signature Scheme V2 签名的 apk 时，直接复制到指定位置，避免再次编译。

编译配置参考代码如下（直接复制不编译）：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
$(shell mkdir -p $(TARGET_OUT)/preloadapp/Deliveryclub)
$(shell cp -r $(LOCAL_PATH)/Deliveryclub.apk $(TARGET_OUT)/preloadapp/Deliveryclub)
LOCAL_PACKAGE_NAME := Deliveryclub
```

## 3.4 预置可卸载应用出错

预置可卸载的应用时，根据出现的错误配置对应的 SELinux 权限。

### 示例

以 facebook 报错为例，log 信息如下：

```
E AndroidRuntime: java.lang.UnsatisfiedLinkError: couldn't find DSO to load: libbreakpad.so result: 0

type=1400 audit(1585044723.864:1187): avc: denied { read } for comm="facebook.katana" name="zoneinfo"
dev="proc" ino=4026531860 scontext=u:r:untrusted_app_27:s0:c141,c256,c512,c768
tcontext=u:object_r:proc_zoneinfo:s0 tclass=file permissive=0

type=1400 audit(1585044723.864:1187): avc: denied { read } for comm="facebook.katana" name="zoneinfo"
dev="proc" ino=4026531860 scontext=u:r:untrusted_app_27:s0:c141,c256,c512,c768
tcontext=u:object_r:proc_zoneinfo:s0 tclass=file permissive=0

type=1400 audit(1585044725.424:1200): avc: denied { open } for comm="EnsureDelegate" path="/data/app-lib/Facebook_260.0.0.42.118/libsuperpackmerged.so" dev="mmcblk0p38" ino=463
scontext=u:r:untrusted_app_27:s0:c141,c256,c512,c768 tcontext=u:object_r:system_data_file:s0 tclass=file
permissive=0

.....
```

## 【解决方法】

上述情况需在 untrusted\_app\_27.te 特殊权限群组中添加以下权限：

```
allow untrusted_app_27 proc_zoneinfo:file read;
allow untrusted_app_27 system_data_file:file open;
```

## 3.5 预置应用配置特殊权限组群

预置到 priv-app 目录下的应用，如需申请 signature 权限，需在 frameworks/base/data/etc/privapp-permissions-platform.xml 中进行额外声明。

## 示例

以预置 Wink，需要申请 `android.permission.SET_WALLPAPER_COMPONENT` & `android.permission.BIND_WALLPAPER` 权限为例。

```
<permission name=" android.permission.SET_WALLPAPER_COMPONENT "/>
<permission name=" android.permission.BIND_WALLPAPER"/>
```

未添加权限申请会无法开机，log 信息如下：

```
06-30 06:54:55.838 1578 1578 E AndroidRuntime: *** FATAL EXCEPTION IN SYSTEM PROCESS: main
06-30 06:54:55.838 1578 1578 E AndroidRuntime: java.lang.IllegalStateException: Signature|privileged
permissions not in privapp-permissions allowlist: {com.android.wallpaper (/system_ext/priv-app/ThemePicker):
android.permission.SET_WALLPAPER_COMPONENT, com.android.wallpaper (/system_ext/priv-
app/ThemePicker): android.permission.BIND_WALLPAPER}
06-30 06:54:55.838 1578 1578 E AndroidRuntime: at
com.android.server.pm.permission.PermissionManagerService.systemReady(PermissionManagerService.java:452
4)
06-30 06:54:55.838 1578 1578 E AndroidRuntime: at
com.android.server.pm.permission.PermissionManagerService.access$800(PermissionManagerService.java:186)
06-30 06:54:55.838 1578 1578 E AndroidRuntime: at
com.android.server.pm.permission.PermissionManagerService$PermissionManagerServiceImpl.onSystem
Ready(PermissionManagerService.java:4992)
```

## 【解决方法】

将应用申请的白名单配置文件编译到目标分区中。

1. 新增资源文件 `Wink_permissions.xml`。

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
  <privapp-permissions package="com.android.wink">
    <permission name=" android.permission.SET_WALLPAPER_COMPONENT "/>
    <permission name=" android.permission.BIND_WALLPAPER "/>
  </privapp-permissions>
</permissions>
```

2. 修改 `Android.mk` 编译规则，确保配置文件和预置目标应用位于同一分区。

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := Wink
LOCAL_MODULE_CLASS := APPS
LOCAL_CERTIFICATE := PRESIGNED
LOCAL_MODULE_PATH := $(TARGET_OUT)/app
LOCAL_SRC_FILES := Wink.apk
```



```

LOCAL_REQUIRED_MODULES := Wink_permissions.xml    //建立编译依赖

$(shell mkdir -p $(TARGET_OUT)/app/Wink/lib/arm/)

$(shell cp -rf $(LOCAL_PATH)/lib/armeabi-v7a $(TARGET_OUT)/Wink/lib/arm/)

include $(BUILD_PREBUILT)

include $(CLEAR_VARS)

LOCAL_MODULE := Wink_permissions.xml    //新增配置文件说明
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_CLASS := ETC

LOCAL_MODULE_PATH := $(TARGET_OUT_ETC)/permissions    //安装路径，必须跟应用位于
同一分区

LOCAL_SRC_FILES := Wink_permissions.xml    //源配置文件
include $(BUILD_PREBUILT)

```

## 3.6 manifest\_checker 错误

从 Android 13 开始引入 AndroidManifests 强制检查 Java 库的机制，确保编译和运行时的 ClassLoaderContext 无差异。为实现这种机制，Google 开发了 manifest\_checker 工具在打包时进行检查，如果检查失败就会报编译错误。

编译时声明库依赖有必选库和可选库两种。

- 必选库  
Java 库或应用在运行时必须依赖的库，如果库加载失败，对应的程序无法运行。

```

* Android.bp properties: `uses_libs`
* Android.mk variables: `LOCAL_USES_LIBRARIES`

```

- 可选库  
Java 库或应用在运行时非必须依赖的库，如果库加载失败，不影响程序的运行。

```

* Android.bp properties: `optional_uses_libs`
* Android.mk variables: `LOCAL_OPTIONAL_USES_LIBRARIES`

```

### 示例

以 EngineerInternal 为例进行说明。

#### 【现象描述】

Android.bp 中，当前编译依赖如下所示。

```

libs: [
    "org.apache.http.legacy"
]

```

编译 EngineerInternal 模块时，库的依赖使用了 libs 属性进行配置，编译系统无法利用当前的 libs 属性推断编译时对应的 ClassPath。但是 AndroidManifests 里面又声明了<uses-library>，即运行时的 ClassPath，如下所示。

AndroidManifests.xml 中：

```
<uses-library android:name="org.apache.http.legacy" android:required="false"/>
```

因此 ManifestChecker 工具校验不通过，导致编译报错。

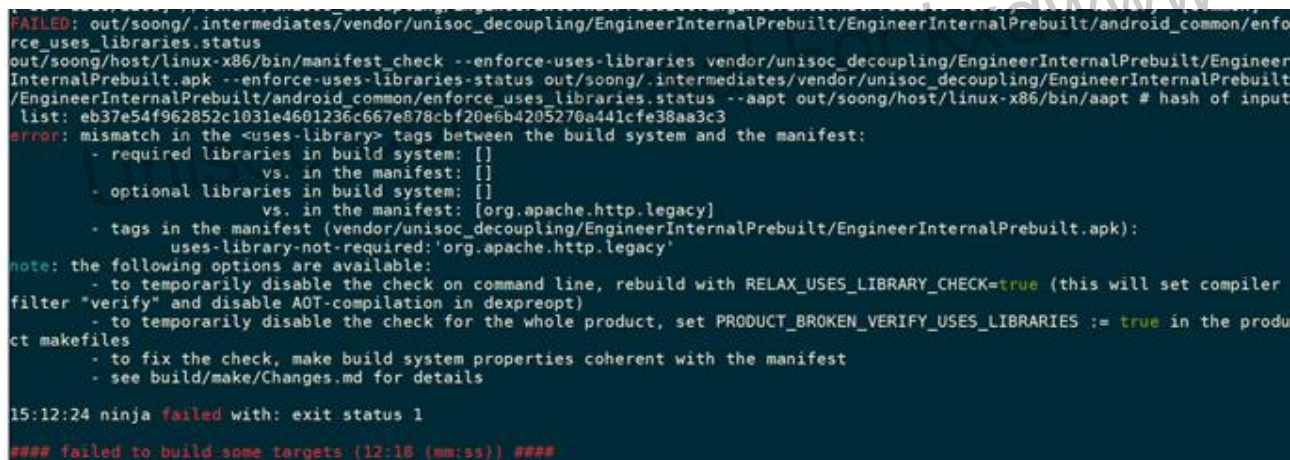
## 【原因分析】

根本原因是未使用正确的属性来配置 ClassPath。当前 “org.apache.http.legacy” 是可选的 Java 库，需要使用 optional\_uses\_libs 进行配置，即新增如下配置。

```
optional_uses_libs: [  
    "org.apache.http.legacy"  
]
```

编译当前模块时，如果当前模块依赖的 Java 库不是系统核心库（不会被 BootClassLoader 加载的库），需要使用 optional\_uses\_libs/uses\_libs 在 Android.bp 中进行配置。如果使用的属性不正确，并且开发人员在清单文件中使用了<uses-library>标签，但是预编译规则中并没有使用正确的宏或者属性进行配置，编译系统检查失败而报编译错误，如图 3-1 所示。

图3-1 uses\_libs 和 uses\_library 检查失败



```
FAILED: out/soong/.intermediates/vendor/unisoc_decoupling/EngineerInternalPrebuilt/EngineerInternalPrebuilt/android_common/enforce_uses_libs.status  
out/soong/host/linux-x86/bin/manifest_check --enforce-uses-libraries vendor/unisoc_decoupling/EngineerInternalPrebuilt/EngineerInternalPrebuilt.apk --enforce-uses-libraries-status out/soong/.intermediates/vendor/unisoc_decoupling/EngineerInternalPrebuilt/EngineerInternalPrebuilt/android_common/enforce_uses_libs.status --aapt out/soong/host/linux-x86/bin/aapt # hash of input list: eb37e54f962852c1031e4601236c667e878cbf20e6b4205278a441cfe38aa3c3  
error: mismatch in the <uses-library> tags between the build system and the manifest:  
- required libraries in build system: []  
  vs. in the manifest: []  
- optional libraries in build system: []  
  vs. in the manifest: [org.apache.http.legacy]  
- tags in the manifest (vendor/unisoc_decoupling/EngineerInternalPrebuilt/EngineerInternalPrebuilt.apk):  
  uses-library-not-required: 'org.apache.http.legacy'  
note: the following options are available:  
- to temporarily disable the check on command line, rebuild with RELAX_USES_LIBRARY_CHECK=true (this will set compiler filter "verify" and disable AOT-compilation in dexpreopt)  
- to temporarily disable the check for the whole product, set PRODUCT_BROKEN_VERIFY_USES_LIBRARIES := true in the product makefiles  
- to fix the check, make build system properties coherent with the manifest  
- see build/make/Changes.md for details  
15:12:24 ninja failed with: exit status 1  
### failed to build some targets (12:18 (mm:ss)) ###
```

## 【解决方法】

修改如下配置信息：

- <uses-library>标签和 uses\_libs/optional\_uses\_libs 对应关系

```
<uses-library android:name="org.apache.http.legacy" android:required="true"/>
```

\* Android.bp properties: `uses\_libs`

\* Android.mk variables: `LOCAL\_USES\_LIBRARIES`

```
<uses-library android:name="org.apache.http.legacy" android:required="false"/>
```

```
* Android.bp properties: `optional_uses_libs`
```

```
* Android.mk variables: `LOCAL_OPTIONAL_USES_LIBRARIES`
```

- 预编译 apk, Android.bp 配置

```
android_app_import {  
    name: "EngineerInternalPrebuilt",  
    privileged: false,  
    certificate: "platform",  
    system_ext_specific: true,  
    overrides: ["EngineerInternal"],  
    apk: "EngineerInternalPrebuilt.apk",  
    optional_uses_libs: ["org.apache.http.legacy"],  
}
```

- AndroidManifests.xml 配置

```
<uses-library android:name="org.apache.http.legacy" android:required="false"/>
```

android:required="false"表示这是一个可选的 Java 库。

- Android.mk 配置

```
LOCAL_OPTIONAL_USES_LIBRARIES += org.apache.http.legacy
```

Unisoc Confidential For kxdwww

# 4

## 参考文档

---

1. 《Android 14 IDH 包编译使用指南》

Unisoc Confidential For kxdwww