



Unisoc Confidential For hiar

Android 应用预置指南

文档版本
发布日期

V1.3
2020-10-23

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。

Unisoc Confidential For hiar

紫光展锐（上海）科技有限公司



前言

概述

本文介绍 Android 系统常见预置的类型及其预置过程，包括应用、可执行文件、native service、so 库以及 jar 包等的预置。

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
JNI	Java Native Interface	Java 本地接口
APK	Android Package	Android 安装包
SO	Shared Object	Linux 下共享动态链接库
JAR	Java Archive	Java 归档文件
CTS	Compatibility Test Suite	兼容性测试套件
API	Application Programming Interface	应用程序编程接口

符号约定

在本文中可能出现下列标志，它所代表的含义如下。

符号	说明
 说明	用于突出重要/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。

变更信息

文档版本	发布日期	修改说明
V1.0	2018-12-07	第一次正式发布。
V1.1	2019-04-25	增加 bin、native service、jar、so 部分。
V1.2	2020-03-26	针对 android10.0 更新，添加常见问题 selinux 报错以及配置特殊权限群组问题。
V1.3	2020-10-24	更新文档格式

关键字

应用预置。

Unisoc Confidential For hiar

目 录

1 预置有源码的应用	1
2 预置 apk 应用	3
2.1 预置应用目录	3
2.2 常见预置需求	4
2.2.1 预置应用可删除/可恢复	4
2.2.2 预置应用可删除/不可恢复	4
2.2.3 预置应用不可删除	4
2.2.4 预置应用带 so 库	4
2.2.5 预置应用可升级	5
2.2.6 预置应用 odex 优化	5
2.2.7 预置应用替换其它 apk	5
2.3 预置 apk	7
3 预置 native service	8
4 预置可执行程序	10
5 预置 so 库	11
5.1 直接拷贝到指定位置	11
5.2 应用中使用 so 库	11
5.3 预置成系统 so 库	12
6 集成 JAR 包	13
6.1 应用中集成 JAR 包	13
6.2 系统中集成 JAR 包	14
6.2.1 应用共享类型 library	14
6.2.2 系统加载类型 library	15
7 预置注意事项	16
7.1 签名类型	16
7.2 系统签名替换	16
7.3 签名冲突	16
7.4 预置使用 v2 签名的应用	17
7.5 预置可卸载的应用	17
7.6 特殊权限群组配置	18

表目录

表 7-1 签名类型	16
------------------	----

Unisoc Confidential For hiar

1 预置有源码的应用

有源码的应用预置过程如下：

1. 在工程中加入源码和 Android.mk

示例如下：

将源代码文件拷贝到/vendor/sprd/platform/packages/apps/目录，并在源代码中添加 Android.mk 文件。

Android.mk 内容如下：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-subdir-java-files)
LOCAL_PACKAGE_NAME := Stk      //应用名字
LOCAL_PRIVATE_PLATFORM_APIS := true  //可使用hidden api
LOCAL_CERTIFICATE := platform    //指定签名
include $(BUILD_PACKAGE)
```

对于有 JNI（Java Native Interface）源码的应用，需在源码的 JNI 目录中添加编译 JNI 的 Android.mk，通过 LOCAL_SHARED_LIBRARIES/LOCAL_JNI_SHARED_LIBRARIES 指定生成的 JNI 库和 so（shared object）文件进行编译。

JNI 目录中编译 JNI so 库 libjni_sprd_facedetector 的 Android.mk 文件内容如下：

```
include $(CLEAR_VARS)
LOCAL_MODULE := libjni_sprd_facedetector
LOCAL_SRC_FILES := com_android_gallery3d_v2_discover_people_FaceDetector.cpp
LOCAL_SHARED_LIBRARIES := libjni_sprd_fa libjni_sprd_fd libjni_sprd_fv
LOCAL_LDLIBS := -llog
include $(BUILD_SHARED_LIBRARY)
```

然后在应用对应的 Android.mk 中使用 JNI 编译生成的 so 库 libjni_sprd_facedetector

```
LOCAL_JNI_SHARED_LIBRARIES += libjni_sprd_facedetector
```

2. 在对应的 board 中加入应用名，将应用 LOCAL_PACKAGE_NAME 加入到 PRODUCT_PACKAGES 中。

存在以下两种情况，以 SC7731E 为例加以说明。

- 如果单个工程需要，则加入对应工程目录下的 mk 中。

如 SC7731E 下仅 sp7731e_1h10 需要 SprdDialerGo，则加入到 device/sprd/pike2/sp7731e_1h10/sp7731e_1h10_native.mk 中：

```
PRODUCT_PACKAGES += SprdDialerGo
```

- 如果所有工程都需要，则加入芯片 board 的 common 目录中的 common_packages.mk 或 devicecommon.mk 下。

如 SC7731E 的 sp7731e_1h10、sp7731e_1h20 都需要 LogManager，则加入到 device/sprd/pike2/common/common_packages.mk 中：

```
PRODUCT_PACKAGES += LogManager
```

Unisoc Confidential For hiar

2

预置 apk 应用

2.1 预置应用目录

主要有以下 7 个预置应用目录：

- **system/app**
系统应用，预置到此目录中的应用不可卸载，恢复出厂设置仍存在。
如果应用需要申请系统权限，并且是比较重要的应用，建议预置到该目录中。
- **system/priv-app**
系统应用，预置到此目录中的应用不可卸载，恢复出厂设置仍存在。
一般不建议预置应用到该目录。**priv-app** 预置的是系统核心应用，该目录下的应用权限等级比 **app** 目录下的应用高。
- **system/preloadapp**
第三方应用预置目录，预置到此目录中的应用可卸载，卸载后恢复出厂设置能恢复。
preloadapp 目录中的应用在开机的过程是同步多线程扫描安装的（有利于加快开机速度）。
- **system/vital-app**
第三方应用预置目录，预置到此目录中的应用可卸载，卸载后恢复出厂设置能恢复。
Android 9.0 上和 **preloadapp** 目录功能一样。
- **data/app**
第三方应用预置目录，预置到此目录中的应用可卸载，卸载后恢复出厂设置不能恢复。
- **vendor/app**
系统应用目录，预置到此目录中的应用不可卸载，恢复出厂设置仍存在。
CTS 会检查 **vendor** 分区中 **apk** 使用的 API 使用是否合规。
- **vendor/priv-app**
系统应用目录，预置到此目录中的应用不可卸载，恢复出厂设置仍存在。
相对于 **vendor/app** 目录，此目录中的应用不签署 **platform** 签名也可获得 **vendor privledge permission**。

2.2 常见预置需求

不同的预置应用目录对应不同的安装需求，按预置应用的不同安装需求，使用 LOCAL_MODULE_PATH 指定预置应用目录。

2.2.1 预置应用可删除/可恢复

可删除、恢复出厂设置后可恢复：

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

preloadapp 和 vital-app 都是常见的第三方应用的预置目录。

- 相同点：目录下应用都可删除，恢复出厂设置后都可再次安装。
- 不同点：vital-app 目录中的应用在开机的过程中是同步安装的，会造成开机时间相应变长，一般需要进入 home 界面就要将安装好的应用预置到 vital-app 下，其它如游戏等预置到 preloadapp 目录下即可。（Android 9.0 之前）

Android 9.0 上 preloadapp 和 vital-app 安装目录已经没有区别，都是多线程扫描安装目录中的 apk 并和主线程进行同步。

2.2.2 预置应用可删除/不可恢复

```
LOCAL_MODULE_PATH := $(TARGET_OUT_DATA_APPS) // 预置到/data/app下即可
```

data 目录恢复出厂设置时候会被擦除，删除后若恢复出厂设置，预置的应用不可以恢复。

2.2.3 预置应用不可删除

预置到 system 分区：

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

相对于 system 分区，vendor 分区中的 apk 会被 cts 检测使用的 api 是否合规。

- 相同点：预置在该目录中的应用不可以卸载，恢复出厂设置后应用也存在。
- 不同点：priv-app 预置的是系统核心应用，不签署 platform 签名也可以获取 vendorprivilegedPermission。

2.2.4 预置应用带 so 库

预置为非系统 apk 时无需处理 so 库文件，安装时系统会自行处理。

预置为系统 apk 需手动解压 apk，将其中的 so 库文件预置到特定位置。

第三方应用可能同时支持多种类型的 so，如 x86_64、x86、armeabi-v7a、armeabi、arm64-v8a，应根据当前项目 Board 中 CPU 架构相关参数配置选择匹配的 so（如 arm 还是 x86，32 还是 64 位的）解压、预置。

预置为系统 so

共享库打包到/system/lib 目录下

```
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := libFFmpeg                                // 模块名字需要加入上述PRODUCT_PACKAGES变量中
LOCAL_MODULE_CLASS := SHARED_LIBRARIES                  // 共享库文件
LOCAL_SRC_FILES := lib/libFFmpeg.so                      // so文件
include $(BUILD_PREBUILT)
```

预置成应用 so（如共享库中已经存在一个同名的 so 库文件了）

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)
```

2.2.5 预置应用可升级

如果预置应用要求可升级，则应保证以后升级的 apk 文件能够获取预置时的 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：

```
LOCAL_CERTIFICATE := PRESIGNED
```

2.2.6 预置应用 odex 优化

Android.mk 中使用 LOCAL_DEX_PREOPT 参数可以使能/禁止预置 apk 进行 odex 优化（odex 优化会加快开机速度）。

```
LOCAL_DEX_PREOPT := false // false表示禁止对该apk进行odex优化
```

apk 进行 odex 优化，编译时会提取 apk 包的 classes.dex 进行优化，生成 odex 文件并删除原 apk 包中的 classes.dex。

2.2.7 预置应用替换其它 apk

Android.mk 使用 LOCAL_OVERRIDES_PACKAGES 指定被替换的 apk，被替换的 apk 不会加入编译。

示例：

```
# 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)
```

Unisoc Confidential For hiar

2.3 预置 apk

以预置 wenxin.apk 为例，过程如下：

1. 创建存放 apk（android package）文件的目录及 Android.mk 文件。

- a 创建 prebuilt_apps 目录（如已存在请忽略）

```
/vendor/sprd/partner/prebuilt_apps
```

- b 创建用于放置第三方 apk 的目录

```
/vendor/sprd/partner/prebuilt_apps/app/
```

- c 创建用于编译预置 apk 的 Android.mk 文件

```
/vendor/sprd/partner/prebuilt_apps/Android.mk
```

2. 将 wenxin.apk 拷贝到路径/vendor/sprd/partner/prebuilt_apps/app/weixin.apk。

3. 修改/vendor/sprd/partner/prebuilt_apps/Android.mk 文件。

修改后内容如下：

```
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)
```

4. 在工程中添加编译模块

以 SC9832E 为例来说明如何修改 mk 文件来增加编译模块，有以下两种情况：

- SC9832E 芯片所有工程都需要预置该应用

修改 device/sprd/sharkle/common/common_packages.mk 文件

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

- 仅某个特定工程（如 sp9832e_1h10_go_native）需要预置该应用

修改 device/sprd/sharkle/sp9832e_1h10_go/sp9832e_1h10_go_native.mk 文件

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

3

预置 native service

以 modemlog_connmgr_service 为例来说明 native service 预置过程。

1. 创建目录并编写相应的 Android.mk 文件，编译生成 native service 对应的 bin 文件。

修改 vendor/sprd/proprieties-source/slogmodem/service/Android.mk，内容如下：

```
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. 添加编译模块

修改 device/sprd/sharkl3/common/DeviceCommon.mk 将 service LOCAL_MODULE 加入 PRODUCT_PACKAGES 中。

DeviceCommon.mk 修改内容如下：

```
#add for log
PRODUCT_PACKAGES += \
```

```
vendor.sprd.hardware.log@1.0-impl \  
vendor.sprd.hardware.log@1.0 \  
vendor.sprd.hardware.log@1.0-service \  
srtld \  
ylog_common \  
log_service \  
  vendor.sprd.hardware.aprd@1.0-impl \  
  vendor.sprd.hardware.aprd@1.0-service \  
  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. 编写 rc 注册文件

编写 vendor/sprd/proprieties-source/slogmodem/service/modemlog_connmgr_service.rc 文件，通知 init 开机时启动 modemlog_connmgr_service。

rc 注册文件内容如下：

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

Unisoc Confidential For hiar

4

预置可执行程序

可执行程序预置过程举例说明如下：

1. 编写相应可执行程序的 Android.mk 文件

编写 vendor/sprd/partner/brcm/wl/Android.mk，内容如下：

```
LOCAL_PATH:= $(call my-dir)
ifeq ($(strip $(BOARD_WLAN_DEVICE)),bcmhdhd)
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第
```

2. 添加编译模块

将 LOCAL_MODULE 添加到 PRODUCT_PACKAGES 中，具体操作参见 3 预置 native service 添加编译模块部分。

5 预置 so 库

预置 so 库分以下三种情况：

- 直接拷贝到指定位置
- 应用中使用 so 库
- 预置成系统 so 库

5.1 直接拷贝到指定位置

修改 device/sprd 目录中对应工程的 mk 文件，使用 PRODUCT_COPY_FILES 将待预置的 so 库直接拷贝到指定位置。

mk 文件修改示例如下：

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

5.2 应用中使用 so 库

LOCAL_PREBUILT_LIBS 变量指定 prebuilt so 库的别名及文件路径，其语法规则如下：

```
LOCAL_PREBUILT_LIB := nickname : path
```

说明

- *nickname* 为 so 库的别名，*path* 为 so 库的文件路径。
- 别名一般不可改变，特别是第三方 JAR（Java Archive）包使用 so 库的情况。
- 别名不含 .so 后缀
- so 库文件路径是第三方 so 库的存放路径。

预置应用中使用的 so 库的示例如下：

编写 vendor/sprd/platform/packages/apps/ValidationTools/Android.mk，内容如下：

```
LOCAL_REQUIRED_MODULES := libCameraVerification libopencv_java3
LOCAL_JNI_SHARED_LIBRARIES += libCameraVerification

include $(BUILD_PACKAGE)
include $(CLEAR_VARS)
LOCAL_MULTILIB := 64
LOCAL_PREBUILT_LIBS := libCameraVerification:libs/arm64-v8a/libCameraVerification.so
include $(BUILD_MULTI_PREBUILT)
```

5.3 预置成系统 so 库

预置成系统 so 库的示例如下：

```
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或者/system/lib64
include $(BUILD_PREBUILT)
```

Unisoc Confidential For hiar

6

集成 JAR 包

6.1 应用中集成 JAR 包

LOCAL_STATIC_JAVA_LIBRARIES 变量用于指定 JAR 包别名，其语法规则如下：

```
LOCAL_STATIC_JAVA_LIBRARIES := nickname
```

LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES 变量用于指定 PREBUILT JAR 包的别名及第三方 JAR 包的路径，其语法规则如下：

```
LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES := nickname : path
```

说明

- *nickname* 为 JAR 包别名，*path* 为 JAR 文件路径。
- LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES 中 JAR 包别名与 LOCAL_STATIC_JAVA_LIBRARIES 中 JAR 包别名必须保持一致。
- 别名不包含 .jar 后缀。
- JAR 文件路径是第三方 JAR 包的存放路径。
- 使用 BUILD_MULTI_PREBUILT 编译。

应用中集成 JAR 包的示例如下：

```
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))
```

6.2 系统中集成 JAR 包

6.2.1 应用共享类型 library

1. 打 JAR 包

可编译生成 JAR 文件

示例如下：

```
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)
```

也可以直接集成 JAR 文件到 system/framework 目录

示例如下：

```
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)
```

然后在 device/sprd 目录对应工程中添加模块

PRODUCT_PACKAGES += 中添加模块 LOCAL_MODULE，生成系统 jar 包放在 system/framework 下面。

2. 声明 JAR 包

编写 xml 文件声明 JAR 包，使系统能够识别此 JAR 包。

JAR 包声明示例如下：

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
<library name="android.user.library" file="/system/framework/libandroid_user.jar"/>
</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 system\etc\permissions, name 是 JAR 包的名字, file 是 JAR 包的路径。

3. 使用 JAR 包

在所需使用的 AndroidManifest.xml 中添加相应的引用后, apk 即可使用对应的 JAR 包。

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

6.2.2 系统加载类型 library

1. 生成 JAR 包

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

Android.mk 文件内容如下:

```
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、配置 JAR 包

device/sprd 目录对应工程的 mk 文件中添加模块:

PRODUCT_PACKAGES += 和 PRODUCT_BOOT_JARS := 末尾分别增加 LOCAL_MODULE 的定义值

mk 文件内容如下:

```
# The order of PRODUCT_BOOT_JARS matters.
PRODUCT_BOOT_JARS := \
    core-oj \
    core-libart \
    conscrypt \
    okhttp \
    legacy-test \
    bouncycastle \
    test // 增加module test
```

3. 添加 JAR 包白名单

修改 build/core/tasks/check_boot_jars/package_whitelist.txt, 文件末尾增加 test.jar 包名

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

7

预置注意事项

7.1 签名类型

预置应用时可使用 LOCAL_CERTIFICATE 指定签名类型，常见签名类型见表 7-1。

表7-1 签名类型

类型	说明
PRESIGNED	编译过程中不再重新签名，使用 apk 原本自带的签名。
shared	sharedUserId android.uid.shared 应用使用此签名类型，签名文件为 shared.pk8。
platform	apk 需要获取 platform signature 权限，签名文件为 platform.pk8。
media	sharedUserId android.media 应用使用此签名类型，签名文件为 media.pk8。
testkey	非 user 版本默认 testkey，签名文件为 testkey.pk8。
releasekey	user 版本默认签名，签名文件为 releasekey.pk8。

7.2 系统签名替换

签名文件放置在目录 build/target/product/security 中，其中 user 版本签名文件位于该目录下的 release 文件夹中。替换系统签名文件时，使用新签名文件覆盖原来的签名文件即可。

7.3 签名冲突

应用安装或更新失败，有可能是签名冲突导致，两种常见签名冲突如下：

- 相同 shared userid 的应用签名类型不一致

应用安装时会检查相同 shared userid 应用的签名类型是否一致，如不一致将无法安装。

安装 Log 中出现 INSTALL_FAILED_SHARED_USER_INCOMPATIBLE 标记，表明应用安装失败原因是签名类型与使用相同 shared userid 的其它应用的签名类型不一致。

如 setting 和 descklock 两个应用的 shared userid 都为 “android.uid.system”，setting 的签名为 platform，那么 descklock 也必须是 platform，否则只有其中一个应用会安装成功。

- 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。

安装 Log 中出现 `INSTALL_FAILED_UPDATE_INCOMPATIBLE` 标记时，表明应用升级失败是由升级前后签名不一致导致。

7.4 预置使用 v2 签名的应用

apk 的两种签名方案如下：

- Signature Scheme v1

Signature Scheme v1 是 jar Signature 来自 JDK，该签名方案通过 ZIP 条目进行验证，apk 签署后可进行许多修改（如可以移动甚至重新压缩文件）。

- Signature Scheme v2

Android 7.0 引入了 apk Signature Scheme v2，该签名方案验证压缩文件的所有字节，而不是单个 ZIP 条目，签名后无法再更改（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，从而安装失败。

预置使用 Signature Scheme v2 签名的 apk 时，直接拷贝到指定位置，避免再次编译。

直接拷贝不编译的示例如下：

```
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
```

7.5 预置可卸载的应用

预置可卸载的应用时，根据出现的错误配置相应 selinux 权限。

如 facebook 报错，log 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
.....
```

上述情况需在 `untrusted_app_27` 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;
```

7.6 特殊权限群组配置

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 中进行额外的声明。

例如 facebook 需要申请 CHANGE_DEVICE_IDLE_TEMP_WHITELIST

```
<permission name="android.permission.CHANGE_DEVICE_IDLE_TEMP_WHITELIST"/>
```

若未添加申请会出现无法开机现象，log 如下：

```
E AndroidRuntime: *** FATAL EXCEPTION IN SYSTEM PROCESS: main
E AndroidRuntime: java.lang.IllegalStateException: Signature|privileged permissions not in privapp-permissions whitelist: {com.facebook.services:
android.p      ermission.CHANGE_DEVICE_IDLE_TEMP_WHITELIST}
E AndroidRuntime:      at com.android.server.pm.permission.PermissionManagerService.systemReady(PermissionManagerService.java:2961)
E AndroidRuntime:      at com.android.server.pm.permission.PermissionManagerService.access$100(PermissionManagerService.java:122)
```

Unisoc Confidential For hiar