Table of Contents

My Awesome Book

This file file serves as your book's preface, a great place to describe your book's content and ideas.

Brillo 介绍

Weave 介绍

制作Linux USB 启动盘

Requirements

Before you download and build the Brillo source, ensure your system meets the following requirements. Then see Initializing a Build Environment for installation instructions by operating system.

Hardware requirements

Your development workstation should meet or exceed these hardware requirements:

- A 64-bit environment is required.
- At least 100GB of free disk space for a checkout, 150GB for a single build, and 200GB or more for multiple builds. If you employ ccache, you will need even more space.
- If you are running Linux in a virtual machine, you need at least 16GB of RAM/swap.

Software requirements

Your workstation will need these software:

- A Linux or Mac OS operating system. It is also possible to build Brillo in a virtual machine on unsupported systems such as Windows. For Linux: Ubuntu 14.04(Trusty) For Mac: Mac OS v10.10(Yosemite) or later
- Java Development Kit(JDK): Java 8
- Python 2.6 -- 2.7, which you can download from python.org.
- GNU Make 3.81 -- 3.82, which you can download form gnu.org.
- Git 1.7 or newer. You can find it at git-scm.com.

Establishing a Build Environment

Setting up a Linux build environment

Installing the JDK

Installing required packages

Configuring USB Access

Using a separate output directory

Setting up a Mac OS build environment

Creating a case-sensitive disk image

Installing the JDK

Master branch

Optimazing a build environment (optional)

Setting up ccache

Downloading the Source

Building the System

Compiling ~~with Jack~~

Selecting Devices

Running Builds

Building Kernels

Known Issues

期末大作业 Brillo 源码分析

要求

- 语言简洁精炼
- 支持使用图表，流程图等工具
- 对目录下的 Makefile 进行重点分析，对重要的文件和目录进行解释说明
- 代码要有文字解释，禁止整篇复制代码
- 对重要的代码进行醒目标识，并添加注释，必要时进行进一步解释说明
- 列出用到的参考文献
- 在 Gitbook 上面编辑，并提交一份纸质版
- 纸质版提交截止日期：6月24日
- 纸质版作业需要有厦大课程作业的封面，并且需要注明组员的详细分工，分工比例
- 提交纸质版作业时请还回开发板，开发板上应同时有带有小组成员学号的 Brillo 系统

示例(/bdk/bionic/Android.mk)

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

- Android.mk 开始必须定义变量 LOCAL_PATH，它用来指定源文件的位置。
- my-dir: 编译系统提供的'my-dir'宏函数，被用来获取当前的目录

```
include $(call all-makefiles-under,$(LOCAL_PATH)) \
                $(call all-makefiles-under,$(LOCAL_PATH)/libc)
```

- 编译当前目录子目录下的，以及子目录的libc目录下的Android.mk文件。注意：
  1. include Android.mk文件，叫其他名字的mk文件，不include.
  2. 只include这个$(LOCAL_PATH)一级目录下的Android.mk文件，而不是所有子目录以及子目录下的Android.mk文件

参考链接

- Android.mk 常用宏和变量
- Android.mk 文件语法规范及使用模板

Weave 协议分析

编者：厦门大学信息学院通信工程系2015级研究生 徐惠、吴振阳、王一臻

源码目录位置：bdk/system/weave/

1、Weave介绍

Weave 是一个开放的通信协议，是专门设计用于物联网设备进行通讯的跨平台通用语言。如图一所示，Weave建立了Cloud-Phone-device的通信链接。

图一

通过weave协议确保接入你的设备，并且你的用户数据是安全和私有的。 Weave基于JSON数据格式。JSON格式是互联网中非常常见的格式，它也可以很方便整合到被收购的Firebase中，也很容易和Chrome浏览器打通。

2、Android.mk文件分析

Weaved目录下包括一个android.mk和brillo、buffet、common、libweaved文件夹，这个android.mk包括了所有Weaved目录下source的信息。接下来将详细分析android.mk内容。
Android.mk文件最开始为本地路径：
LOCAL_PATH := $(call my-dir)
Android.mk 开始必须定义变量 LOCAL_PATH，它用来指定源文件的位置。

my-dir: 编译系统提供的'my-dir'宏函数，返回当前 Android.mk 所在的目录的路径。
接下来是给出模块用到的一些参数：common variables
包括如下几个部分：

- 1、buffetCommonCppExtension:= .cc
  C++源码文件的扩展名为.cc
- 2、buffetCommonCppFlags
  指定其他的编译选项和宏定义
- 3、buffetCommonCIncludes
  是NDK根目录的相对路径。当编译C/C++、汇编文件时，这些路径将被追加到头文件搜索路径列表中。
- 4、buffetSharedLibraries
  运行时所依赖的库

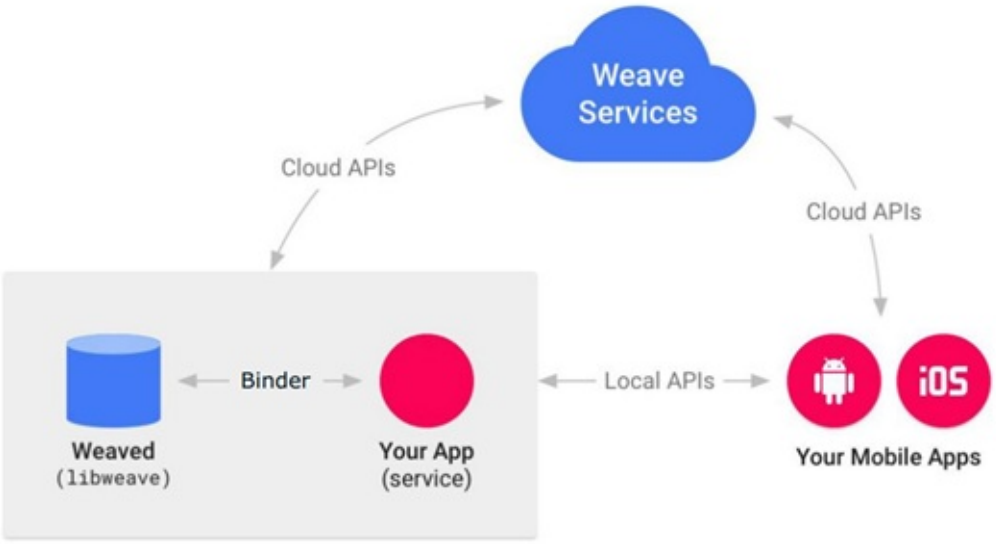
3、对模块库的编译和使用

3.1、weave-common模块分析

weave-common模块：编译静态库，主要是waved守护进程和客户端libweaved库的代码的共享。

- include $(CLEAR_VARS) //指向一个编译脚本, 必须在开始一个新模块之前包含这个脚本，用于重置除//LOCAL_PATH变量外的，所有LOCAL_XXX系列变量
- LOCAL_MODULE := weave-common //这是模块的名字，它必须是唯一的，而且不能包含空格
- LOCAL_CPP_EXTENSION := $(buffetCommonCppExtension) //指出C++ 扩展名。(可选)
- LOCAL_CFLAGS := $(buffetCommonCFlags) //一个可选的设置，在编译C/C++ source时添加如Flags。用来附加编译选项
- LOCAL_CPPFLAGS := $(buffetCommonCppFlags) //指定当编译C/C++源码的时候，传给编译器的标志。它一般用来指定其他的编译选项和宏定义。C++ Source 编译时添加的C Flags。这些Flags将出现在LOCAL_CFLAGS //flags 的后面
- LOCAL_C_INCLUDES := $(buffetCommonCIncludes) //一个路径的列表，是NDK根目录的相对路径（LOCAL_SRC_FILES中的文件相对于LOCAL_PATH）。当编译C/C++、汇编文件时，这些路径将被追加到头文件搜索路径列表中。一个可选的path列表。相对于NDK ROOT目录。编译时将会把这些目录附上。
- LOCAL_AIDL_INCLUDES := $(LOCAL_PATH)/brillo
- LOCAL_SHARED_LIBRARIES := $(buffetSharedLibraries) //要链接到本模块的动态库
- LOCAL_EXPORT_C_INCLUDE_DIRS := $(LOCAL_PATH) //一个可选的path列表。相对于NDK ROOT 目录。编译时，将会把这些目录附上

- LOCAL_CLANG := true
- LOCAL_SRC_FILES //包含将要打包如模块的C/C++ 源码
- include $(BUILD_EXECUTABLE) //编译为Native C可执行程序
- include$(BUILD_STATIC_LIBRARY), //该模块需要使用哪些静态库，以便在编译时进行链接
- include$(BUILD_SHARED_LIBRARY)， //表示模块在运行时要依赖的共享库（动态库），在链接时就需要，以便在生成文件时嵌入其相应的信息。
  LOCAL_SRC_FILES := \
  brillo/android/weave/IWeaveClient.aidl \
  brillo/android/weave/IWeaveCommand.aidl \
  brillo/android/weave/IWeaveService.aidl \
  brillo/android/weave/IWeaveServiceManager.aidl \
  brillo/android/weave/IWeaveServiceManagerNotificationListener.aidl \
  common/binder_constants.cc \
  common/binder_utils.cc \
  //包含将要打包如模块的C/C++ 源码，我们可以看到打包的文件包括../android/weave下的所有文件，和../common/ 下的binder_constants.cc和binder_utils.cc，这是因为weaved和app service之间是通过binder机制进程之间的通信，如图二所示。



图二 命令/数据流

3.2、weave-daemon-common模块

weave-daemon-common模块：编译静态库，这个模块是weaved守护进程和测试单元之间代码共享，它本质是weaved静态库的实现。
与前面的不同主要在LOCAL_SRC_FILES上。
LOCAL_SRC_FILES := \
brillo/weaved_system_properties.cc \
buffet/ap_manager_client.cc \

buffet/avahi_mdns_client.cc \
buffet/binder_command_proxy.cc \
buffet/binder_weave_service.cc \
buffet/buffet_config.cc \
buffet/dbus_constants.cc \
buffet/flouride_socket_bluetooth_client.cc \
buffet/http_transport_client.cc \
buffet/manager.cc \
buffet/shill_client.cc \
buffet/socket_stream.cc \
buffet/webserv_client.cc \
这个模块已经是各个设备、手机、网络等之间的互联，因此包括http_transport、socket_stream、flouride_socket_bluetooth_client等传输方式。也包括binder，系统属性等。最后代码是brillo的安全特征，引入的Oauth 2.0认证，加密证书。Weave在安全性方面还保证了用户数据的隐私，不仅在传输层，在链路层使用SSL（Secure Sockets Layer）/TLS（Transport Layer Security）协议保证用户的数据隐私。
ifdef BRILLO
LOCAL_SRC_FILES += buffet/keystore_encryptor.cc
else
LOCAL_SRC_FILES += buffet/fake_encryptor.cc
endif

3.3、Weaved模块

Weaved模块：编译为Native C可执行程序，weave守护进程的二进制 模块内调用前面编译的两个模块weave-common和weave-daemon-common。
LOCAL_STATIC_LIBRARIES := weave-common \
LOCAL_WHOLE_STATIC_LIBRARIES := weave-daemon-common
最后再打包main.cc构成一个完整的可执行程序
LOCAL_SRC_FILES := \
buffet/main.cc

3.4、Libweaved模块

Libweaved模块：编译动态库，客户端的给weave守护进程的库文件，如果你想要进行weaved通信就要链接上libweaved。app与weaved之间通过binder链接
这里只包含库weave-common模块，没有直接与外界相连。
LOCAL_STATIC_LIBRARIES := weave-common

3.5、weaved_test模块

weaved_test模块：weaved测试模块。
库包括前面的weave-common模块和weave-daemon-common模块和测试支持的库。
LOCAL_STATIC_LIBRARIES := \
libbrillo-test-helpers \
libchrome_test_helpers \
libgtest \
libgmock \
libweave-test \
weave-daemon-common \
weave-common \

4、参考链接

1、Android.mk 文件语法详解
http://www.cnblogs.com/wainiwann/p/3837936.html
2、Android.mk详解
http://blog.csdn.net/ly131420/article/details/9619269

安全分析

编者：厦门大学信息学院通信工程系2015级研究生：杨江河，阮晓杨，赖雅玲

源码目录位置：bdk/system/security/

1.概述

在Android中/system/ keystore进程提供了一个安全存储的服务。每一个Android用户都有一块其私有的安全存储区域。keystore是java的密钥库，用来进行通信加密，比如数字签名。同时keystore用来保存密钥对，比如公钥和私钥。所有秘钥信息使用一个随机key并用AES加密算法加密，加密好的密文采用另外一个key加密后保存到本地磁盘。在近期的一些Android版本中，证书管理（例如RSA算法的私有key）是可以通过专门的硬件做支持的。这也就是说，keystore的key只是用来标识存储在专有硬件上的真正key。Fig.1很好的阐述了keystore安全存储机制的工作原理。

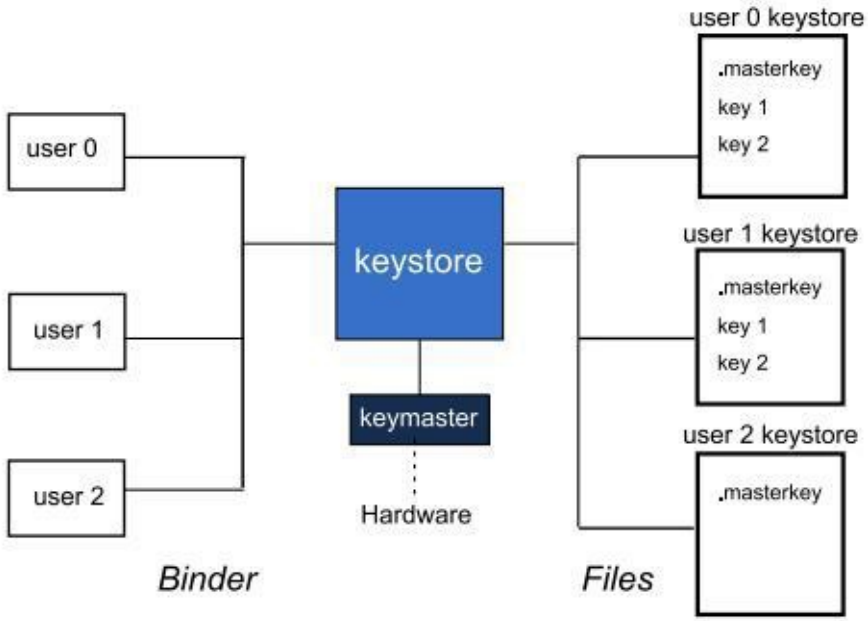


Fig.1 The keystore service

2.Security目录重要文件框架

```
+--Keystore
|   +--include
|   +--test
|     --keystore_main.cpp
|     --keystore_cli_v2.cpp
+--keystore-engine
|     --android_engine.cpp
|     --keyhandle.cpp
+--softkeymaster
|   +--include
|   |   --keymaster
|          --softkeymaster
|     --Android.mk
|     --keymaster_openssl.cpp
|     --module
--MODULE_LICENSE_APACHE2
--NOTICE
```

3.keystore主函数解释说明

*keystore_main.cpp

KeyStore是密钥对的安全容器。在这个工具中，每个文件储存在一个密钥对中。密钥被编码到文件中，键值用校验和加密。加密的键值被用户定义的密码保护。为了让事情变得简单，缓存总是大于所需的最大空间，所以缓存的边界检查总是被省略。

> keymaster2_device_t* dev; if (keymaster_device_initialize(&dev)) { ALOGE("keystore keymaster could not be initialized; exiting"); return 1; }

密钥设备初始化

> keymaster2_device_t* fallback; if (fallback_keymaster_device_initialize(&fallback)) { ALOGE("software keymaster could not be initialized; exiting"); return 1; }

回调密钥设备初始化程序

> android::IPCThreadState::self()->joinThreadPool();

因为只存在一个进程，所以我们只能把处理绑定事物处理当成一个单线程程序

> keymaster_device_release(dev);

删除密钥设备

KeyStore主要文件功能描述

文件名	功能描述
auth_token_table.cpp	基于标准算法的遍历模板封装以及授权的相关操作
blob.cpp	二进制大文件的读写和，大小控制和加密控制
entropy.cpp	加密算法的开启和随机数据的生成
IKeystoreService.cpp	Parcel数据的读写，Token接口数据的增加，删除，更新和列出
key_store_service.cpp	密钥服务的增加，删除，更新显示；密钥服务的状态获取和重启
keyblob_utils.cpp	获取软键头尺寸并添加软键头
keystore.cpp	密钥库的构造和析构，密钥库的初始化以及管理键的复制和读写
keystore_cli_v2.cpp	文件的读写和密钥的产生、删除和导出
keystore_client_impl.cpp	主要是密钥库客户端实现的操作，包括生成随机数加密，导入、导出、生成、删除和列出密钥，加密算法采用AES-256-CBC，并且授权算法是HMAC-SHA256
keystore_get.cpp	密钥库的获取
keystore_utils.cpp	文件的读写，添加密钥授权以及APP和用户ID的获取
operation.cpp	操作图的添加，获取，更新和移除
permissions.cpp	SELinux系统的配置，SELinux系统下密钥库的权限检查以及密钥库euid的获取
user_state.cpp	用户状态的初始化、设置和重启；键值文件的复制以及从密码中生成密钥

4.Makefile文件分析

keystore-engine/Android.mk

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

提示当前文件的路径，必须定义在文件开头 my-dir 返回当前Android.mk所在的目录路径。

```
include $(CLEAR_VARS)
```

CLEAR_VARS 变量由Build System提供。并指向一个指定的GNU Makefile，由它负责清理很多LOCAL_xxx.
例如：LOCAL_MODULE, LOCAL_SRC_FILES, LOCAL_STATIC_LIBRARIES等等。但不清理LOCAL_PATH.
这个清理动作是必须的，因为所有的编译控制文件由同一个GNU Make解析和执行，其变量是全局的。所以清理后才能避免相互影响。

```
ifneq (,$(wildcard $(TOP)/external/boringssl/flavor.mk))
    include $(TOP)/external/boringssl/flavor.mk
else
    include $(TOP)/external/openssl/flavor.mk
endif
```

如果boringssl/flavor.mk文件存在，则先读取/boringssl中的flavor.mk。
否则读取/openssl下的flavor.mk文件。

```
LOCAL_MODULE := libkeystore-engine
```

指定当前模块的名字为libkeystore-engine，这个模块名字是唯一的且不能被更改。
Build System会自动添加适当的前缀和后缀。例如：foo；要产生动态库则生成libfoo.so。 但请注意！如果模块名被定为：libfoo.则生成libfoo.so不再加前缀。

```
  LOCAL_SRC_FILES := \
    android_engine.cpp
```

指定当前模块所包含的文件为android_engine.cpp。 不必列出头文件，build System会自动帮我们找出依赖文件

```
LOCAL_C_INCLUDES+= \
    external/openssl/include \
    external/openssl
```

设置头文件的搜索路径，默认的头文件的搜索路径是LOCAL_PATH目录。

```
LOCAL_SHARED_LIBRARIES += \
    libcrypto \
    liblog \
    libcutils \
    libutils \
    libbinder \
    libkeystore_binder
```

表示模块在运行时要依赖的共享库（动态库），在链接时就需要，以便在生成文件时嵌入其相应的信息。
注意：它不会附加列出的模块到编译图，也就是仍然需要在Application.mk中把它们添加到程序要求的模块中。

```
LOCAL_MODULE_TAGS := optional
```

指定模块编译运行的版本，一共有四个选项：
user: 指该模块只在user版本下才编译
eng: 指该模块只在eng版本下才编译
tests: 指该模块只在tests版本下才编译
optional:指该模块在所有版本下都编译

```
LOCAL_ADDITIONAL_DEPENDENCIES := $(LOCAL_PATH)/Android.mk
```

增加额外的依赖 如果模块需要依靠一些并不直接建立的文件，就可以将这些编译标签加入到LOCAL_MODULE_TAGS里面去，通常这个是一些无法自动创建的附属工程文件。

```
include $(BUILD_SHARED_LIBRARY)
```

BUILD_SHARED_LIBRARY：是Build System提供的一个变量，指向一个GNU Makefile Script。
它负责收集自从上次调用 include $(CLEAR_VARS) 后的所有LOCAL_XXX信息。并决定编译为什么。

---

softkeymaster/Android.mk

```
ifeq ($(USE_32_BIT_KEYSTORE), true)
LOCAL_MULTILIB := 32
endif
```

设置如果使用的是32位的静态库的时候，要将LOCAL_MULTILIB的值设置为32。

```
LOCAL_CFLAGS = -fvisibility=hidden -Wall -Werror
```

一个可选的设置，在编译C/C++ source 时添加如Flags。
用来附加编译选项。 注意：不要尝试在此处修改编译的优化选项和Debug等级。它会通过您Application.mk中的信息自动指定。
也可以指定include 目录通过：LOCAL_CFLAGS += -I。 这个方法比使用LOCAL_C_INCLUDES要好。因为这样也可以被ndk-debug使用。

---

keystore/Android.mk

```
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
```

LOCAL_MODULE_CLASS 标识了所编译模块最后放置的位置，如果不指定，不会放到系统中，之后放在最后的obj目录下的对应目录中。

代码	注释
LOCAL_MODULE_CLASS := ETC	# 表示放于system/etc目录
LOCAL_MODULE_CLASS := EXECUTABLES	# 放于/system/bin
LOCAL_MODULE_CLASS := SHARED_LIBRARIES	# 放在/system/lib下

---

5. 参考链接

- Java ™ Cryptography Architecture (JCA) Reference Guide
- 深入理解Android之Java Security第一部分
- android.mk 详解 其他makefile文件类似
- Android.mk 文件语法规范及使用模板
- 深入了解android平台的jni(二)
- Android.mk 文件语法详解
- Android开发文档中关于SSL方面的知识。
- Android开发文档中关于keystore方面的知识。

系统恢复分析

编者：厦门大学信息学院通信工程系2015级研究生 韩国安、李文聪、蔡志标

源码目录位置：bdk/bootable/recovery/

1、recovery目录结构分析

recovery包含很多个文件夹：

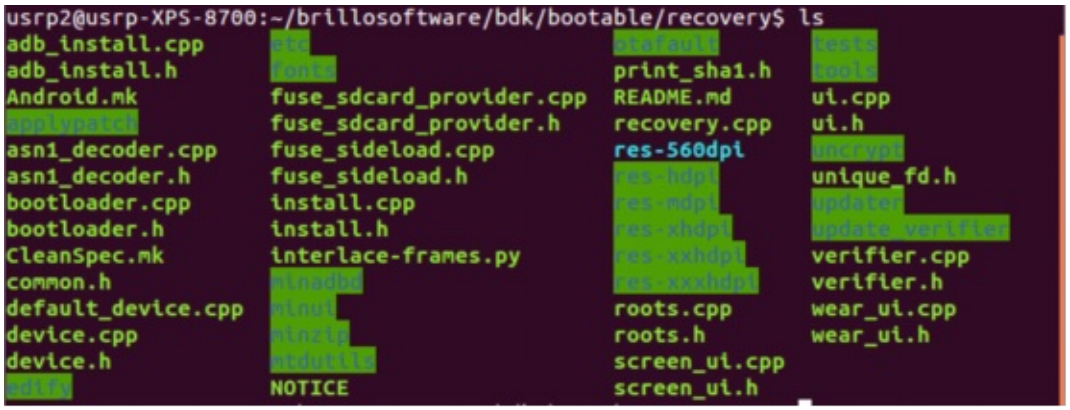


表1.1 主要目录的意义如下表所示：

目录	意义
Applypatch	文件夹是应用补丁
Edify	是类似android系统用来运行updater-scripts（刷机脚本）的Edify语言
Etc	是保存系统的配置文件
Fonts	系统字体文件
Minadbd	在android中使用mina框架处理socket通信方法
adb	本身是一个客户端
Minui	在recovery的UI显示部分应用层调用minui库实现了图形的描绘以及固定大小的文字显示
Minzip	提供简单的Zip文件支持
Mtdutils	是mtd-utils工具如使用mtd-util工具向nand flash写入文件系统jffs2.img
Otafault	默认的OTA升级是指系统提供的标准软件升级方式
res-560dpi、res-hdpi、res-mdpi、res-xhdpi、res-xxhdpi、res-xxhdpi	不同像素的图片资源
Tests	测试文件
Tools	工具包
Uncrypt	在系统recovery时如果系统是加密的就解密
Updater	在recovery中使用updater脚本来完成升级
update_verifier	更新认证
update_verifier	是操作系统内核运行之前运行的可以初始化硬件设备建立内存空间映射图从而将系统的软硬件环境带到一个合适状态以便最终调用操作系统内核

2.recovery目录下Andriod.mk文件分析

LOCAL_PATH := $(call my-dir)
%Android.mk开始必须定义变量LOCAL_PATH它用来指定源文件的位置。my-dir: 编译系统提供的'my-dir'宏函数，被用来获取当前的目录。

%首先创建一个名为**libfusesideload**的第一个模块，代码如下：

include $(CLEAR_VARS)
%编译系统提供CLEAR_VARS变量，它指向了一个用来清除LOCAL_XXX开头的变量（例如LOCAL_MODULE, LOCAL_SRC_FILES但是LOCAL_PATH除外)的makefile文件，需要它的

原因是整个的编译上下文中，所有的变量都是全局的，这样就可以保证这些变量只在局部范围内起作用。

LOCAL_SRC_FILES := fuse_sideload.cpp
%LOCAL_SRC_FILES必须包含一系列的C/C++源文件，它将会被建立和装载到模块中。在该代码将源文件fuse_sideload.cpp建立并装载到模块中。

LOCAL_CLANG := true
%启用clang编译器

LOCAL_CFLAGS := -O2 -g -DADB_HOST=0 -Wall -Wno-unused-parameter
LOCAL_CFLAGS += -D_XOPEN_SOURCE -D_GNU_SOURCE
%LOCAL_CFLAGS为C编译器传递额外的参数，例如该代码中的-g表示生成调试信息，其他的gcc命令读者可自行查阅相关文献

LOCAL_MODULE := libfusesideload
%创建一个LOCAL_MODULE，MODULE名字必须是唯一的并且不包含任何的空格，编译系统将自动的修改生成文件的前缀和后缀，该代码创建了一个名为libfusesideload的LOCAL_MODULE。

LOCAL_STATIC_LIBRARIES := libcutils libc libmincrypt
%LOCAL_STATIC_LIBRARIES的作用是链接进来静态库的模块，后面跟着的是需要加进来的静态库的模块的名称，而不是静态库的名称，该代码的作用是将libcutils，libc以及libmincrypt三个静态库模块链接进来。

include $(BUILD_STATIC_LIBRARY)
%BUILD_STATIC_LIBRARY是编译系统提供的变量，指向一个GNU Makefile脚本，负责收集自从上次调用'include $(CLEAR_VARS)'以来，定义在LOCAL_XXX变量中的所有信息，并且决定编译什么，如何正确地去做，并根据其规则生成静态库。
%至此完成模块libfusesideload的创建

%接下来创建名为**recovery**的第二个模块

include $(CLEAR_VARS)
%再次重置除LOCAL_PATH外的以开头LOCAL_XXX的变量，以便接下进行新的赋值。

LOCAL_SRC_FILES := \
adb_install.cpp \
asn1_decoder.cpp \
bootloader.cpp \
device.cpp \
fuse_sdcard_provider.cpp \
install.cpp \
recovery.cpp \

roots.cpp \
screen_ui.cpp \
ui.cpp \
verifier.cpp \
wear_ui.cpp \
%将以上的.cpp源文件建立并装载到模块中。

LOCAL_MODULE := recovery
%创建名为recovery的LOCAL_MODULE

LOCAL_FORCE_STATIC_EXECUTABLE := true
%LOCAL_FORCE_STATIC_EXECUTABLE作用：如果编译时候需要链接的动态库存在静态库形式，那么在这个编译变量等于true的情况下，将会链接到对应的静态库而不是动态库，这个只有文件系统中/sbin目录下的应用程序会用到。

ifeq ($(TARGET_USERIMAGES_USE_F2FS),true)
ifeq ($(HOST_OS),linux)
LOCAL_REQUIRED_MODULES := mkfs.f2fs
endif
endif
%判断TARGET_USERIMAGES_USE_F2FS是否是ture以及HOST_os是否为linux，如果二者都满足这将LOCAL_REQUIRED_MODULES设置为mkfs.f2fs。
LOCAL_REQUIRED_MODULES的作用在于指定模块运行所依赖的模块，即在加载本模块是同步加载mkfs.f2fs模块

RECOVERY_API_VERSION := 3
RECOVERY_FSTAB_VERSION := 2
%设置RECOVERY_API以及RECOVERY_FSTAB的版本号

LOCAL_CFLAGS += -DRECOVERY_API_VERSION=$(RECOVERY_API_VERSION)
LOCAL_CFLAGS += -Wno-unused-parameter
LOCAL_CLANG := true
%启用clang编译器，为C编译器传递额外的参数

LOCAL_C_INCLUDES += \
system/vold \
system/extras/ext4_utils \
system/core/adb \
%LOCAL_C_INCLUDES作用为添加额外的C/C++编译头文件路径，该代码添加了三个路径：system/vold，system/extras/ext4_utils以及system/core/adb

LOCAL_STATIC_LIBRARIES := \
libbatterymonitor \
libext4_utils_static \

libsparse_static \
libminzip \
libz \
libmtdutils \
libmincrypt \
libminadbd \
libfusesideload \
libminui \
libpng \
libfs_mgr \
libbase \
libcutils \
libutils \
liblog \
libselinux \
libm \
libc
%将以上列出的静态库模块链接进来

LOCAL_HAL_STATIC_LIBRARIES := libhealthd

ifeq ($(TARGET_USERIMAGES_USE_EXT4), true)
LOCAL_CFLAGS += -DUSE_EXT4
LOCAL_C_INCLUDES += system/extras/ext4_utils
LOCAL_STATIC_LIBRARIES += libext4_utils_static libz
endif
%判断$(TARGET_USERIMAGES_USE_EXT4)是否为ture，如果是则在C编译器中传入-DUSE_EXT4参数，添加C/C++编译头文件路径system/extras/ext4_utils，链接静态库的libext4_utils_static以及libz模块

LOCAL_MODULE_PATH := $(TARGET_RECOVERY_ROOT_OUT)/sbin
%LOCAL_MODULE_PATH作用是指定模块的生成路径为$(TARGET_RECOVERY_ROOT_OUT)/sbin

ifeq ($(TARGET_RECOVERY_UI_LIB),)
LOCAL_SRC_FILES += default_device.cpp
else
LOCAL_STATIC_LIBRARIES += $(TARGET_RECOVERY_UI_LIB)
endif
%判断$(TARGET_RECOVERY_UI_LIB)，如果符合条件将default_device.cpp添加到模块中，反之，将静态库模块$(TARGET_RECOVERY_UI_LIB)链接进来

include $(BUILD_EXECUTABLE)
%生成静态库，至此完成名为recovery的第二个模块的创建

%创建名为**libverifier**的第三个模块

include $(CLEAR_VARS)
%重置除LOCAL_PATH外的以开头LOCAL_XXX的变量，以便接下进行新的赋值。

LOCAL_CLANG := true
%启用clang编译器

LOCAL_MODULE := libverifier
%创建名为libverifier的LOCAL_MODULE

LOCAL_MODULE_TAGS := tests
%添加该模块的标记为

LOCAL_SRC_FILES := \
asn1_decoder.cpp \
verifier.cpp \
ui.cpp
%将以上的.cpp源文件tests建立并装载到模块中。

include $(BUILD_STATIC_LIBRARY)
%生成静态库，至此完成名为libverifier的第三个模块的创建

%导入所有指定目录下的**Android.mk**文件

include $(LOCAL_PATH)/minui/Android.mk \
$(LOCAL_PATH)/minzip/Android.mk \
$(LOCAL_PATH)/minadbd/Android.mk \
$(LOCAL_PATH)/mtdutils/Android.mk \
$(LOCAL_PATH)/tests/Android.mk \
$(LOCAL_PATH)/tools/Android.mk \
$(LOCAL_PATH)/edify/Android.mk \
$(LOCAL_PATH)/uncrypt/Android.mk \
$(LOCAL_PATH)/otafault/Android.mk \
$(LOCAL_PATH)/updater/Android.mk \
$(LOCAL_PATH)/update_verifier/Android.mk \
$(LOCAL_PATH)/applypatch/Android.mk

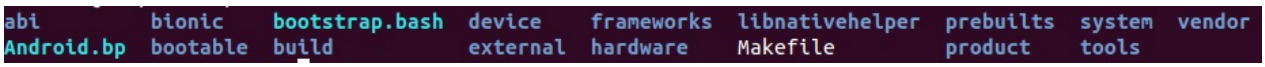
顶层目录结构分析

编者：厦门大学信息学院通信工程系2015级研究生 吴泽石、谭云生、郑明炀

源码目录: bdk/

4.1 bdk目录概述

bdk目录下的文件



每个文件/文件夹作用

文件夹/文件名	作用
abi	abi相关代码。ABI：application binary interface，应用程序二进制接口
bionic	bionic C库
bootable	启动引导相关代码
build	存放系统编译规则及generic等基础开发配置包
device	是类似android系统用来运行updater-scripts（刷机脚本）的Edify语言
external	brillo使用的一些开源的模组
frameworks	核心框架--C语言，是brillo应用程序的框架。
hardware	主要是硬件适配层HAL代码
libnativehelper	安卓类库的支持函数
prebuilts	x86和arm架构下预编译的一些资源
product	一些简单的例程
system	文件系统、应用及组件
tools	默认的OTA升级是指系统提供的标准软件升级方式
vendor	产商提供bsp（板级支持包）存放的目录

4.2 部分目录详细分析

4.2.1 abi

应用程序二进制接口（application binary interface，ABI）描述了应用程序和操作系统之间，一个应用和它的库之间，或者应用的组成部分之间的低接口 。ABI不同于API ，API定义了源代码和库之间的接口，因此同样的代码可以在支持这个API的任何系统中编译 ，然而ABI允许编译好的目标代码在使用兼容ABI的系统中无需改动就能运行。 ABI掩盖了各种细节，例如:调用约定控制着函数的参数如何传送以及如何接受返回值；系统调用的编码和一个应用如何向操作系统进行系统调用；以及在一个完整的操作系统ABI中，对象文件的二进制格式、程序库等等。一个完整的ABI，像 Intel二进制兼容标准 (iBCS) ，允许支持它的操作系统上的程序不经修改在其他支持此ABI的操作体统上运行。

4.2.2 bionic

Bionic是Android的C library。libc是GNU/Linux以及其他类Unix系统的基础函数库，最常用的就是GNU的libc，也叫glibc。Android之所以采用bionic而不是glibc，有几个原因：

- 版权问题，因为glibc是LGPL
- 库的体积和速度，bionic要比glibc小很多。
- 提供了一些Android特定的函数，getprop LOGI等

4.2.3 bootable

bootable中只包含recovery文件夹，也就是用于系统的恢复使用。 该目录包含以下结构：

文件夹	作用
Applypatch	文件夹是应用补丁
Edify	是类似android系统用来运行updater-scripts（刷机脚本）的Edify语言
Etc	是保存系统的配置文件
Fonts	系统字体文件
Minadbd	在android中使用mina框架处理socket通信方法
adb	本身是一个客户端
Minui	在recovery的UI显示部分应用层调用minui库实现了图形的描绘以及固定大小的文字显示
Minzip	提供简单的Zip文件支持
Mtdutils	是mtd-utils工具如使用mtd-util工具向nand flash写入文件系统jffs2.img
Otafault	默认的OTA升级是指系统提供的标准软件升级方式
res-560dpi、res-hdpi、res-mdpi、res-xhdpi、res-xxhdpi、res-xxhdpi	不同像素的图片资源
Tests	测试文件
Tools	工具包
Uncrypt	在系统recovery时如果系统是加密的就解密
Updater	在recovery中使用updater脚本来完成升级
update_verifier	更新认证
update_verifier	是操作系统内核运行之前运行的可以初始化硬件设备建立内存空间映射图从而将系统的软硬件环境带到一个合适状态以便最终调用操作系统内核

4.2.4 build

build子目录存放编译系统的核心代码，build主要文件结构如下所示：

文件夹	作用
buildspec.mk.default	buildspec的模版文件，可定义一些变量比如TARGET_BUILD_VARIANT:=user，TARGET_BUILD_TYPE:=release
CleanSpec.mk	增量编译时，会执行该文件里的命令，这些命令一般用于清除中间文件
core	编译系统的核心文件放在该目录，主要是一些makefile
envsetup.sh	编译时需先用source envsetup.sh设置好环境变量，该脚本提供了许多有用的命令，比如cout,croot,cgrep,在详细介绍Android编译步骤时会列出来
libs	是一个C++模块，编译后可生成libhost.a静态库，里面的函数主要用于与编译主机交互
target	包含编译目标相关的makefile，它有两个子文件夹 board和product，产品都在该目录下定义，比如generic,full产品，定义设备产品时，会从这里继承产品
tools	各种工具，多数使用python编写，工具有用于签名的signpak, 用于下载device配置的roomservice.py等，后续将详细介绍

4.2.5 product

由google官方提供的基本例程。包括一个common 目录，提供简单接口如gpio、i2c、keyboard等基础组件。以及一个services使用的例程example-ledflasher。本次实验的实践部分就是通过修改ledflahser服务完成的。

4.2.6 tools

工程编译过程中方便使用的小工具，特别是brunch，可以通过brunch的指令bsp、config、product等对源码或者环境进行管理。

4.2.7 vendor

该目录是对应产商提供bsp（板级支持包）存放的目录。edison板子由intel提供，因此里面存放的是edsion板子uboot固件以及蓝牙和wifi的驱动。

out 目录分析

编者：厦门大学信息学院通信工程系2015级研究生 马继勇、李浩、罗文彬

out目录

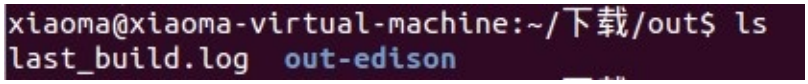
程序编译完成后，将在根目录中生成一个out文件夹，所有生成的内容均放置在这个文件夹中。

主要目录

out目录包含两个文件：last_build.log、out-edison

```
out/
  -- last_build.log
  -- out-edison
```

```
xiaoma@xiaoma-virtual-machine:~/下载/out$ ls
```



两个主要目录的意义如下表所示：

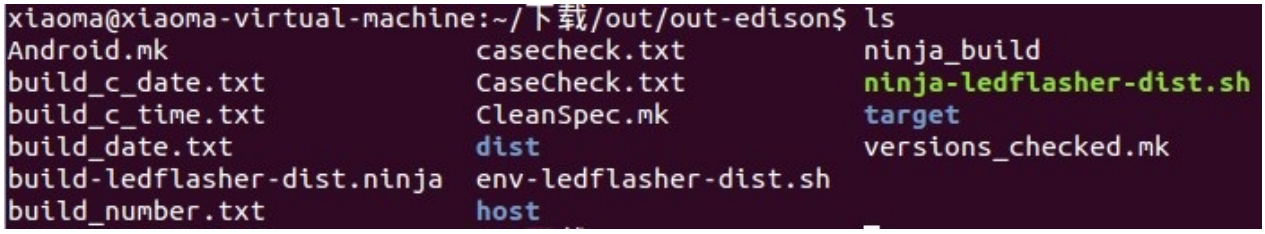
目录	意义
last_build.log	用来记录最后一次运行时系统的状态、配置等信息
out-edison	文件夹的总体结构路径

out-edison

进入out-edison目录，可以看到其结构路径。

```
out-edison/
  --Android.mk
  --host
  --target
  --build_c_time.txt
  --build_c_date.txt
  --dist
  --build_date.txt等等
```

```
xiaoma@xiaoma-virtual-machine:~/下载/out/out-edison$ ls
```



其中比较重要的几个文件是Android.mk、host、target，下面分别对他们进行说明。

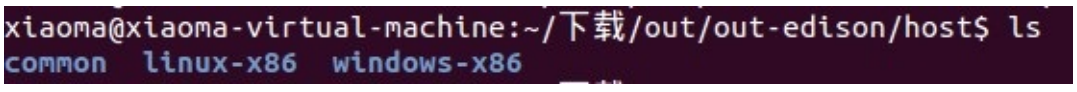
- Android.mk

  Android.mk用来向编译系统描述你的源代码。具体来说，该文件是GNU Makefile的一小部分，会被编译系统解析一次或多次。用户可以在每一个Android.mk file中定义一个或多个模块。

- host host文件夹目录的结构如下所示:

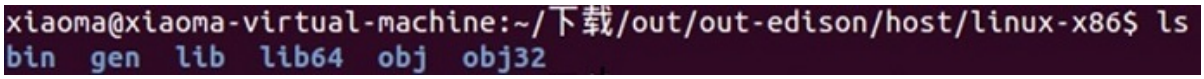
  ```
  out-edison/host/
    --common
    --linux-x86
    --windows-x86
  ```

  ```
  xiaoma@xiaoma-virtual-machine:~/下载/out/out-edison/host$ ls
  ```

  将host的子目录展开，各个子目录和文件的功能如下所示：

```
out-edison/host/
  --common
      --obj（JAVA库）
  --linux-x86
      --bin（二进制程序）
      --gen（自动生成的文件目录，一般并不需要去修改）
      --lib（共享库*.so）
      --lib64（同上）
      --obj（中间生成的目标文件）
      --obj32（同上）
  --windows-x86
      --obj（中间生成的目标文件）
```



总的来说，host目录是一些在主机上用的工具，有一些是二进制程序，有一些是JAVA的程序。

- target

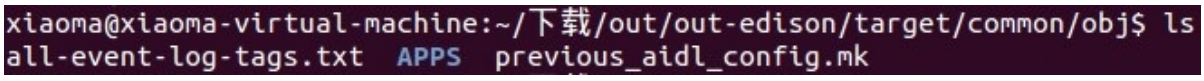
  target目录包含common与product，其中common目录表示通用内容，produt中则是针对产品的内容。

  将target目录展开，其子目录和文件的功能和意义描述如下：

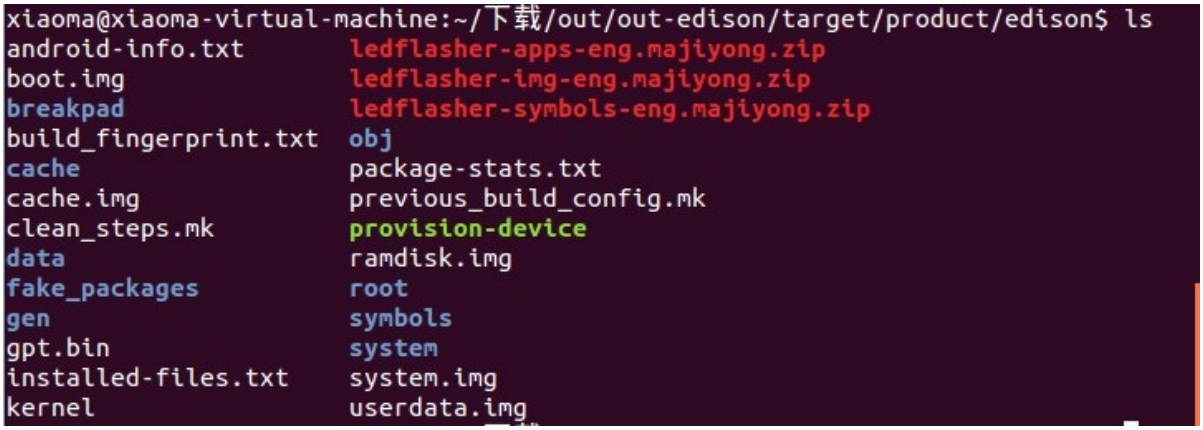
  out-edison/target/

```
--common
    --obj
--Apps（包含了JAVA应用程序生成的目标，每个应用程序对应其中一个子目录，将结合每个应用程序的原始文件升策划国内Android应用程序的APK包）
        --all-event-log-tags.txt（为所有知道的事件生成一个文件）
        --previous-aidl-config（之前定义进程间的通信接口的配置）
--product
    --edison
        --breakpad
        --cache
        --data（存放数据的文件系统）
        --fake_packages
        --gen
        --obj（存放所有动态库、静态库等）
        --root
        --symbols
        --system（存放主要的文件系统）
        --android-info.txt
        --boot.img
        --build_fingerprint.txt
        --cache.img
        --clean_steps.mk
        --gpt.bin
        --installed-files.txt
        --kernel
        --package-stats.txt
        --previous_build_config.mk
        --provision-device
        --ramdisk.img（内存盘的根文件系统映像）
        --system.img     （文件系统的映像）
        --userdata.img（数据内容映像）
```

```
xiaoma@xiaoma-virtual-machine:~/下载/out/out-edison/target/common/obj$ ls
```




```
xiaoma@xiaoma-virtual-machine:~/下载/out/out-edison/target/product/edison$ ls
```


xiaoma@xiaoma-virtual-machine:~/下载/out/out-edison/target/product/edison$ ls
android-info.txt
boot.img
breakpad
build_fingerprint.txt
cache
cache.img
clean_steps.mk
data
fake_packages
gen
gpt.bin
installed-files.txt
kernel
ledflasher-apps-eng.majiyong.zip
ledflasher-img-eng.majiyong.zip
ledflasher-symbols-eng.majiyong.zip
obj
package-stats.txt
previous_build_config.mk
provision-device
ramdisk.img
root
symbols
system
system.img
userdata.img

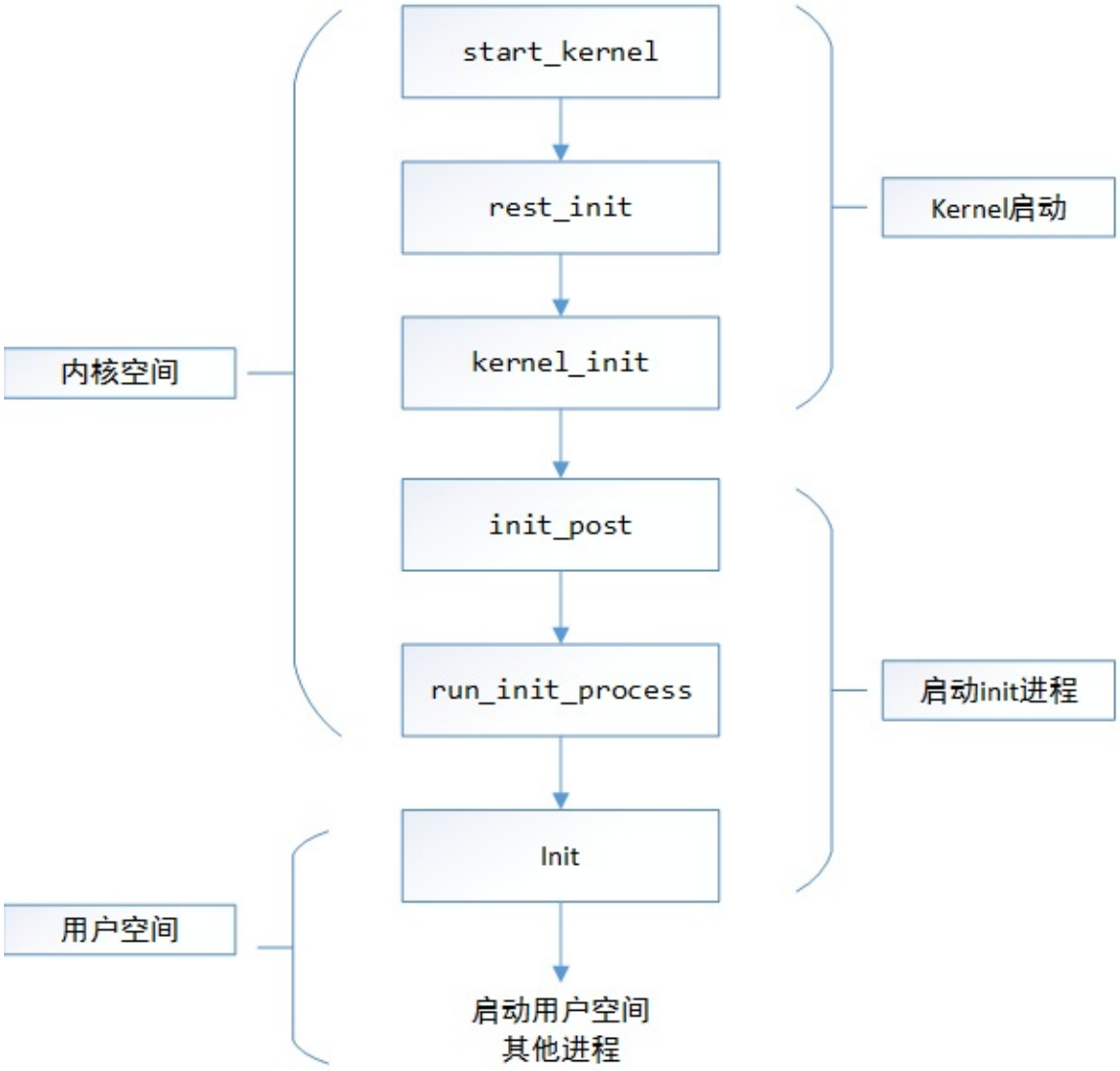
启动进程（BDK/system/core/init）分析

编者：厦门大学信息学院电子工程系2015级研究生 陈琼、陈硕，通信工程系2015级研究生 刘妍

init启动过程

概述

Brillo是一个物联网底层操作系统。它是源于Android，是对Android底层的一个细化，得到了Android的全部支持。而 Android是基于Linux的操作系统，所以init也是Android系统中用户空间的第一个进程，它的进程号是1。下面先简单的看一下init进程的启动过程。



LINUX进程启动图

目前Linux有很多通讯机制可以在用户空间和内核空间之间交互，例如设备驱动文件（位于/dev目录中）、内存文件（/proc、/sys目录等）。了解Linux的同学都应该知道Linux的重要特征之一就是一切都是以文件的形式存在的，例如，一个设备通常与一个或多个设备文件对应。这些与内核空间交互的文件都在用户空间，所以在Linux内核装载完，需要首先建立这些文件所在的目录。而完成这些工作的程序就是本文要介绍的init。Init是一个命令行程序。其主要工作之一就是建立这些与内核空间交互的文件所在的目录。当Linux内核加载完后，要做的第一件事就是调用init程序，也就是说，init是用户空间执行的第一个程序。

在分析init的核心代码之前，还需要初步了解init除了建立一些目录外，还做了如下的工作

1. 初始化属性
2. 处理配置文件的命令（主要是init.rc文件），包括处理各种Action。
3. 性能分析（使用bootchart工具）。
4. 无限循环执行command（启动其他的进程）。

   尽管init完成的工作不算很多，不过代码还是非常复杂的。Init程序并不是由一个源代码文件组成的，而是由一组源代码文件的目标文件链接而成的。这些文件位于如下的目录。

BDK/system/core/init

主函数分析

```
其中init.cpp是init的主文件，现在打开该文件，看看其中的内容。
由于init是命令行程序，所以分析init.cpp首先应从main函数开始，main函数代码如下：
```

```int main(int argc, char* *argv) { /*代码 通过命令行判断argv[0]的字符串内容，来区分当前程序是init，ueventd或是watchdogd。
```

程序的main函数原型为 main(int argc, char* argv[]), ueventd以及watchdogd的启动都在init.rc中描述，
由
init进程解析后执行fork、exec启动，因此其入口参数的构造在init代码中，将在init.rc解析时分析。此时我们只
需要直到argv[0]中将存储可执行文件的名字。
*/
if (!strcmp(basename(argv[0]), "ueventd")) {
    return ueventd_main(argc, argv);
}

if (!strcmp(basename(argv[0]), "watchdogd")) {
    return watchdogd_main(argc, argv);
}

// Clear the umask.
/*
代码<part2>

umaks(0)用于设定当前进程（即/init）的文件模型创建掩码（file mode creation mask），注意这里的文件是广泛意义上的文件，包括普通文件、目录、链接文件、设备节点等。

PS. 以上解释摘自umask的mannual，可在linux系统中执行man 3 umask查看。

Linux C库中mkdir与open的函数运行如下。

int mkdir(const char *pathname, mode_t mode);

int open(const char *pathname, int flags, mode_t mode);

Linux内核给每一个
进程都设定了一个掩码，当程序调用open、mkdir等函数创建文件或目录时，传入open的mode会现

在掩码做运算，得到的文件mode，才是文件真正的mode。

譬如要创建一个目录，并设定它的文件权限

为0777，mkdir("testdir", 0777)

但实际上写入的文件权限却未必是777，因为mkdir系统调用在创建testdir时，

会将0777与当前进程的掩码（称为umask）运算，具体运算方法为 0777&~umask作为testdir的真正权限。
因此上述init中首先调用umask(0)将进程掩码清0，这样调用open/mkdir等函数创建文件或目录时
，传入的mode就会作为实际的值写入文件系统。

umask(0);

add_environment("PATH", _PATH_DEFPATH);

bool is_first_stage = (argc == 1) || (strcmp(argv[1], "--second-stage") != 0);

```
/* 接下来创建目录，并挂载内核文件系统，它们是

*tmpfs，虚拟内存文件系统，该文件系统被挂载到/dev目录下，

主要存放设备节点文件，用户进程通过访问/dev目录下的设备节点文件可以

与硬件驱动程序交互。
*devpts，一种虚拟终端文件系统

*proc，虚拟文件系统，被挂载到/proc目录下，通过该文件系统可与内核数据结构交互
，查看以及设定内核参数。
*sysfs，虚拟文件系统，被挂载到/sys目录下，它与proc类似，是2.6内核在吸收
了proc文件系统的设计经验和教训的基础上所实现的一种较新的文件系统，为内核提供了统一的设
备驱动模型。（引用：http://www.ibm.com/developerworks/cn/linux/l-cn-sysfs/index.html）*/

if (is_first_stage) {
    mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
    mkdir("/dev/pts", 0755);
    mkdir("/dev/socket", 0755);
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    #define MAKE_STR(x) __STRING(x)
    mount("proc", "/proc", "proc", 0, "hidepid=2,gid=" MAKE_STR(AID_READPROC));
    mount("sysfs", "/sys", "sysfs", 0, NULL);
}

// We must have some place other than / to create the device nodes for
// kmsg and null, otherwise we won't be able to remount / read-only
// later on. Now that tmpfs is mounted on /dev, we can actually talk
// to the outside world.
/*该函数名字暗示将init进程的stido，包括stdin（标准输入，文件描述符为0）、stdout
（标准输出，文件描述符为1）以及stderr（标准错误，文件描述符号为2），全部重定向/dev/null
设备，但是细心的读者可能会有疑问，在代码<part2>中虽然挂载了tmpfs文件系统到/dev目录下，但
是并未创建任何设备节点文件，/dev/null此时并不存在啊，如何才能将stdio重定向到null设备中
呢？此时Anrdoid系统上处于启动的早期阶段，可用于接收init进程标准输出、标准错误的设备节点还
不存在。因此init进程一不做二不休，直接把它们重定向到/dev/__nulll__了。*/
open_devnull_stdio();
klog_init();
klog_set_level(KLOG_NOTICE_LEVEL);

NOTICE("init %s started!\n", is_first_stage ? "first stage" : "second stage");

if (!is_first_stage) {
    // Indicate that booting is in progress to background fw loaders, etc.
    close(open("/dev/.booting", O_WRONLY | O_CREAT | O_CLOEXEC, 0000));

//这一句用来初始化Android的属性系统，将在init之属性系统中专门介绍。
    property_init();

    // If arguments are passed both on the command line and in DT,
    // 接下来init程序调用函数process_kernel_cmdline解析内核启动参数。
    内核通常由bootloader（启动引导程序）加载启动，目前广泛使用的bootloader大都基
    于u-boot定制。
```

```
    内核允许bootloader启动自己时传递参数。在内核启动完毕之后，启动参数可
    通过/proc/cmdline查看。
    process_kernel_dt();
    process_kernel_cmdline();

    // Propagate the kernel variables to internal variables
    // used by init as well as the current required properties.
    export_kernel_boot_props();
}

/*这部分代码是在Android4.1之后添加的，随后伴随Android系统更新不停迭代。
这段代码主要涉及SELinux初始化。由于SELinux与Android系统启动关闭不大，暂不分析。*/
// Set up SELinux, including loading the SELinux policy if we're in the kernel domain.
selinux_initialize(is_first_stage);

// If we're in the kernel domain, re-exec init to transition to the init domain now
// that the SELinux policy has been loaded.
if (is_first_stage) {
    if (restorecon("/init") == -1) {
        ERROR("restorecon failed: %s\n", strerror(errno));
        security_failure();
    }
    char* path = argv[0];
    char* args[] = { path, const_cast<char*>("--second-stage"), nullptr };
    if (execv(path, args) == -1) {
        ERROR("execv(\"%s\") failed: %s\n", path, strerror(errno));
        security_failure();
    }
}

// These directories were necessarily created before initial policy load
// and therefore need their security context restored to the proper value.
// This must happen before /dev is populated by ueventd.
NOTICE("Running restorecon...\n");
restorecon("/dev");
restorecon("/dev/socket");
restorecon("/dev/__properties__");
restorecon("/property_contexts");
restorecon_recursive("/sys");

epoll_fd = epoll_create1(EPOLL_CLOEXEC);
if (epoll_fd == -1) {
    ERROR("epoll_create1 failed: %s\n", strerror(errno));
    exit(1);
}

signal_handler_init();

property_load_boot_defaults();
export_oem_lock_status();
start_property_service();

const BuiltinFunctionMap function_map;
```

```
Action::set_function_map(&function_map);

Parser& parser = Parser::GetInstance();
parser.AddSectionParser("service",std::make_unique<ServiceParser>());
parser.AddSectionParser("on", std::make_unique<ActionParser>());
parser.AddSectionParser("import", std::make_unique<ImportParser>());
parser.ParseConfig("/init.rc");
/*
 * 解析完init.rc后会得到一系列的action等，下面的代码将执行处于early-init阶段的action。
 * init将action按照执行时间段的不同分为early-init、init、early-boot、boot。
 * 进行这样的划分是由于有些动作之间具有依赖关系，某些动作只有在其他动作完成后才能执行，
 * 所以就有了先后的        区别。
 * 具体哪些动作属于哪个阶段是在init.rc中的配置决定的
 */
ActionManager& am = ActionManager::GetInstance();

am.QueueEventTrigger("early-init");

// Queue an action that waits for coldboot done so we know ueventd has set up all of /
dev...
am.QueueBuiltinAction(wait_for_coldboot_done_action, "wait_for_coldboot_done");
// ... so that we can start queuing up actions that require stuff from /dev.
am.QueueBuiltinAction(mix_hwrng_into_linux_rng_action, "mix_hwrng_into_linux_rng");
am.QueueBuiltinAction(keychord_init_action, "keychord_init");
am.QueueBuiltinAction(console_init_action, "console_init");

// Trigger all the boot actions to get us started.
am.QueueEventTrigger("init");

// Repeat mix_hwrng_into_linux_rng in case /dev/hw_random or /dev/random
// wasn't ready immediately after wait_for_coldboot_done
am.QueueBuiltinAction(mix_hwrng_into_linux_rng_action, "mix_hwrng_into_linux_rng");
```

/*这段函数将利用bootmode与字符串"charger"将其保存到is_charger变量中，

is_charger非0表明但前Android是以充电模式启动，否则为正常模式。正常启动

模式与充电模式需要启动的进程不同的，这两种模式启动具体启动的程序差别将在init.rc 解析时介绍。*/

```
// Don't mount filesystems or start core system services in charger mode.

std::string bootmode = property_get("ro.bootmode");
if (bootmode == "charger") {
    am.QueueEventTrigger("charger");
} else {
    am.QueueEventTrigger("late-init");
}

// Run all property triggers based on current state of the properties.
am.QueueBuiltinAction(queue_property_triggers_action, "queue_property_triggers");

while (true) {
    if (!waiting_for_exec) {
        am.ExecuteOneCommand();
        restart_processes();
    }

    int timeout = -1;
    if (process_needs_restart) {
        timeout = (process_needs_restart - gettime()) * 1000;
        if (timeout < 0)
            timeout = 0;
    }

    if (am.HasMoreCommands()) {
        timeout = 0;
    }

    bootchart_sample(&timeout);

    epoll_event ev;
    int nr = TEMP_FAILURE_RETRY(epoll_wait(epoll_fd, &ev, 1, timeout));
    if (nr == -1) {
        ERROR("epoll_wait failed: %s\n", strerror(errno));
    } else if (nr == 1) {
        ((void (*)()) ev.data.ptr)();
    }
}

return 0;}```
```

可以看出在main()函数的最后，init进入了一个无限循环，并等待一些事情的发生。 即：在执行完前面的初始化工作以后，init变为一个守护进程。init所关心的事件 有三类：属性服务事件、keychord事件和SIGNAL，当有这三类事件发生时， init进程会调用相应的handle函数进行处理。

主要函数简介

函数名	所在文件	功能概述
mkdir	system/core/init/init.cpp	建立文件系统的本目录
mount	system/core/init/init.cpp	装载文件系统
open_devnull_stdio	system/core/init/init.cpp	打开基本输入、出设备
log_init	system/core/init/init.cpp	初始化日志功能
init_parse_config_file	system/core/init/init_parser.cpp	读取init.rc文件内容到内存数据区
parse_config	system/core/init/init_parser.cpp	识别init.rc文件中的
parse_new_section	system/core/init/init_parser.cpp	识别section类别
parse_service	system/core/init/init_parser.cpp	对service
parse_line_service	system/core/init/init_parser.cpp	对service sectio的option选项进行分析
parse_action	system/core/init/init_parser.cpp	对action
parse_line_action	system/core/init/init_parser.cpp	对action
action_for_each_trigger	system/core/init/init_parser.cpp	触发某个action的执行
action_add_queue_tail	system/core/init/init_parser.cpp	将某个action的从action_list加到action_queue
execute_one_command	system/core/init/init.cpp	执行当前action的一个command
action_remove_queue_head	system/core/init/init_parser.cpp	从action_queue表上移除头结点(action)
do_class_start	system/core/init/builtins.cpp	class_start
		遍历service_list

		表上的所有结点
service_start_if_not_disabled	system/core/init/builtins.cpp	判断service的fla是否disabled，如果不是，则调用关函数，准备启service
service_start	system/core/init/init.cpp	启动service的主要入口函数，设service数据结构的相关数据结构后，调用fork创建一个新的进行，后调用execve执行新的service
fork Lib	function(ulibc)	进程创建函数
execve	Lib function(ulibc)	调用执行新的service
poll	Lib function(ulibc)	查询property_set_fd signal_fd和keychord_fd文件句柄是否有服务求
handle_property_set_fd	system/core/init/property_service.cpp	处理系统属性服请求，如：service,
handle_keychord	system/core/init/keycords.cpp	处理注册在service
handle_signal	system/core/init/signal_handler.cpp	处理SIGCHLDsignal

认识**init.rc**和了解如何编写**init.c**

在前面分析了init进程的启动过程和main函数，以下内容将着重对配置文件

(init.rc)的解析做一下分析以及如何形成一份init.c。

init.rc脚本语法

init.rc文件不同于init进程，init进程仅当编译完Android后才会生成，而init.rc文 件存在于Android平台源代码中。init.rc在源代码中的位置为：@system/core /rootdir/init.rc。init.rc文件的大致结构如下图所示： init.rc文件并不是普通的配置文件，而是由一种被称为“Android初始

init.rc文件不同于init进程，init进程仅当编译完Android后才会生成，而init.rc文 件存在于Android平台源代码中。init.rc在源代码中的位置为：@system/core /rootdir/init.rc。init.rc文件的大致结构如下图所示： init.rc文件并不是普通的配置文件，而是由一种被称为“Android初始化语言”（Android Init Language，这里简称为AIL）的脚本写成的文件。在了解init如何解析init.rc文件 之前，先了解AIL非常必要，否则机械地分析init.c及其相关文件的源代码毫无意义。

为了学习AIL，读者可以到自己Android手机的根目录寻找init.rc文件，最好下载到本地以 便查看，如果有编译好的Android源代码，在out/target/product /geneic/root目录也可找到init.rc文件。

AIL由如下4部分组成。

```
1.  动作（Actions）
2.  命令（Commands）
3·服务（Services）
4.  选项（Options）
 这4部分都是面向行的代码，也就是说用回车换行符作为每一条语句的分隔符。
 而每一行的代码由多个符号（Tokens）表示。可以使用反斜杠转义符在Token中插入空格。
 双引号可以将多个由空格分隔的Tokens合成一个Tokens。如果一行写不下，可以在行尾
 加上反斜杠，来连接下一行。也就是说，可以用反斜杠将多行代码连接成一行代码。
 AIL的注释与很多Shell脚本一行，以#开头。
 AIL在编写时需要分成多个部分（Section），而每一部分的开头需要指定Actions或
 Services。也就是说，每一个Actions或Services确定一个Section。而所有的
 Commands和Options只能属于最近定义的Section。如果Commands和Options在
 第一个Section之前被定义，它们将被忽略。
```

Actions和Services的名称必须唯一。如果有两个或多个Action或Service拥有同样的 名称，那么init在执行它们时将抛出错误，并忽略这些Action和Service。 如何去写 Android init.rc (Android init language) Android 初始化语言由四大类声明组成 : 行为类 (Actions), 命 令类 (Commands) ， 服务类 (Services), 选项类 (Options).

- 初始化语言以行为单位，由以空格间隔的语言符号组成。
  1. C 风格的反斜杠转义符可以用来插入空白到语言符号。
  2. 双引号也可以用来防止文本被空格分成多个语言 符号。
  3. 当反斜杠在行末时，作为折行符。
- 以 # 开 始 (前面允许有空格) 的行为注释行。
- Actions 和 Services 隐 含声明一个新的段落。
  1. 所有该段落下 Commands 或 Options 的声明属于该段落。
  2. 第一段落前的 Commands 或 Options 被 忽略。
- Actions 和 Services 拥 有独一无二的命名。在它们之后声明相同命名的类将被当作错误并忽略。

Actions 是一系列命令的命名。 Actions 拥有一个触发器 (trigger) 用 来决定 action 何时 执行。当一个 action 在符合触发条件被执行时，如果它还没被加入到待执行队列中的话，则加入到队列最后。

队列中的 action 依次执行， action 中的命令也依次执行。 Init 在 执行命令的中间处理其 它活动 (设备创建 / 销毁 ,property 设 置，进程重启)。 Actions 表现形式为： on

---

Services

Services 是由 init 启 动 ，在它们退出时重启 (可选)。 Service 表 现形式为：

service [ ]*

Options

Options 是 Services 的 修饰，它们影响 init 何时、如何运行 service .

1. critical 这是一个设备关键服务 (device-critical service) . 如果它在 4 分钟内退出超过 4 次， 设备将重启并进入恢复模式。
2. disabled
3. 这个服务的级别将不会自动启动，它必须被依照服务名指定启动才可以启 动。

setenv

```
设置已启动的进程的环境变量 <name> 的值 <value>
```

socket [[ ]]

```
创建一个名为 /dev/socket/<name> 的 unix domin socket ，并传送它的 fd 到已启动的进程
。 <type> 必 须为 "dgram" 或 "stream". 用户和组默认为 0.
```

user

```
在执行服务前改变用户名。当前默认为 root. 如果你的进程需要 linux 能 * 力，你不能使用这个
命令。你必须在还是 root 时请求能力，并下降到你 需要的 uid.
```

group [ ]*

```
    在执行服务前改变组。在第一个组后的组将设为进程附加组 ( 通过 setgroups()). 当 前默认为
```

```
在执行服务前改变组。在第一个组后的组将设为进程附加组 ( 通过 setgroups()). 当 前默认为
```

root.

oneshot

```
在服务退出后不重启。
```

class

```
为 service 指 定一个类别名。同样类名的所有的服务可以一起启动或停止。如果没有指定类别的服务默认为 "default" 类。
```

onrestart

```
当服务重启时执行一个命令。
```

Triggers

```
Triggers( 触发器 ) 是一个字符串，可以用来匹配某种类型的事件并执行一个 action 。
```

boot

```
这是当 init 开 始后执行的第一个触发器 ( 当 /init.conf 被加载 )
```

=

```
当 property <name> 被设为指定的值 <value> 时 触发。
```

device-added-

device-removed-

```
当设备节点被添加或移除时触发。
```

service-exited-

```
当指定的服务存在时触发
```

Commands

exec [ ]*

```
Fork 并执行一个程序 (<path>). 这将被 block 直 到程序执行完毕。
最好避免执行例如内建命令以外的程序，它可能会导致 init 被 阻塞不动。
```

export

```
设定全局环境变量 <name> 的值 <value> ， 当这个命令执行后所有的进程都可以取得。
```

ifup

```
使网络接口 <interface> 联 机。
```

import

```
解析一个 init 配 置文件，扩展当前配置文件。
```

hostname

```
设置主机名
```

chmod

```
改变文件访问权限
```

chown

```
改变文件所属和组
```

class_start

```
当指定类别的服务没有运行，启动该类别所有的服务。
```

class_stop

```
当指定类别的服务正在运行，停止该类别所有的服务。
```

domainname

```
设置域名。
```

insmod

```
加载该路径 <path> 的 模块
```

mkdir [mode] [owner] [group]

```
在 <path> 创 建一个目录 , 可选选项 :mod,owner,group.
如果没有指定，目录以 755 权限， owner 为 root,group 为
root 创 建 .
```

mount

[ ]*

```
尝试 mount <device> 到目录 <dir>. <device> 可以用 mtd@name
格式以命名指定一个 mtd 块设备。 <mountoption> 包 含 "ro","rw","remount","noatime".
```

setkey

setprop

```
设置系统 property <name> 的值 <value>.
```

setrlimit

```
设置 resource 的 rlimit.
```

start

```
启动一个没有运行的服务。
```

stop

```
停止一个正在运行的服务
。
```

symlink

```
创建一个 <path> 的 符号链接到 <target>
```

sysclktz

```
设置系统时区 (GMT 为 0)
```

trigger

```
触发一个事件。用于调用其它 action 。
```

write [ ]*

```
打开 <path> 的 文件并写入一个或多个字符串。
```

Properties

Init 会更新一些系统 property 以提供查看它正在干嘛。

init.action 当前正在执行的 action, 如果没有则为 ""

init.command

```
被执行的命令，如果没有则为 ""
```

init.svc.

```
命名为 <name> 的 服务的状态 ("stopped", "running", "restarting")
```

init.rc 示例：

```# not complete -- just providing some examples of usage # on boot export PATH /sbin:/system/sbin:/system/bin

export LD_LIBRARY_PATH /system/lib

mkdir /dev

mkdir /proc
```

```
mkdir /sys
mount tmpfs tmpfs /dev
mkdir /dev/pts
mkdir /dev/socket
mount devpts devpts /dev/pts
mount proc proc /proc
mount sysfs sysfs /sys
write /proc/cpu/alignment 4
ifup lo
hostname localhost
domainname localhost
mount yaffs2 mtd@system /system
mount yaffs2 mtd@userdata /data
import /system/etc/init.conf
class_start default
service adbd /sbin/adbd
user adb
group adb
service usbd /system/bin/usbd -r
user usbd
group usbd
socket usbd 666
service zygote /system/bin/app_process -Xzygote /system/bin --zygote
socket zygote 666
service runtime /system/bin/runtime
user system
group system
```

on device-added-/dev/compass

start akmd

on device-removed-/dev/compass

stop akmd

service akmd /sbin/akmd

disabled

user akmd

group akmd```

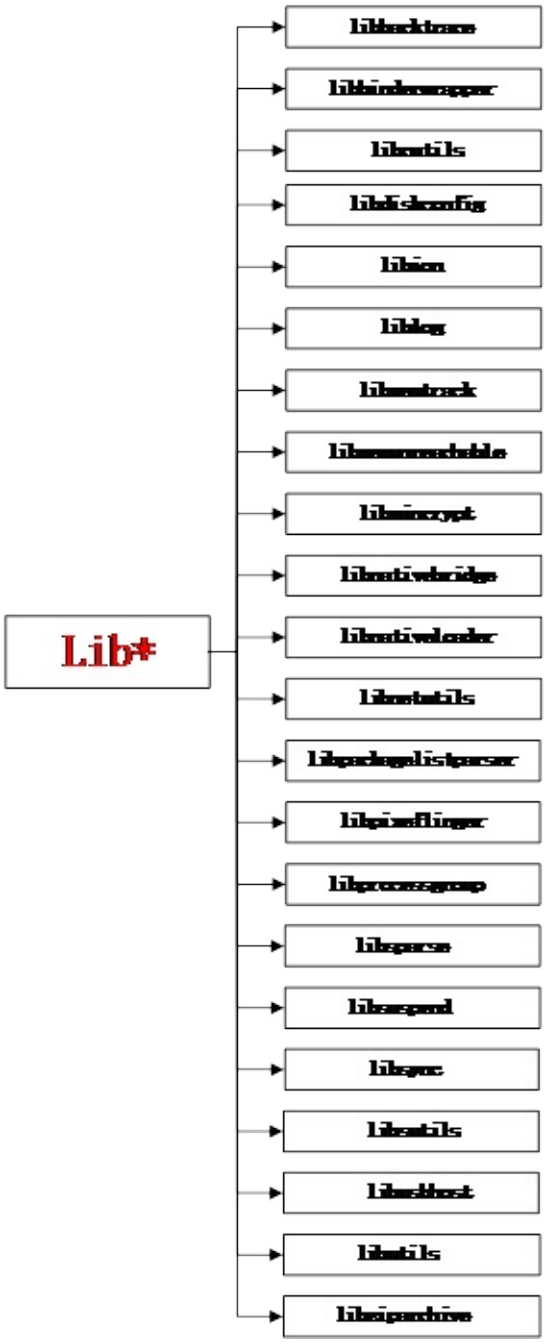
调试

默认情况下， init 执行的程序输出的信息和错误到 /dev/null. 为了 debug ， 你可以通过 Android 程序 logwrapper 执行你的程序。这将复位向输出 / 错误输出到 Android logging 系统 (通过 logcat 访问) 。 例如 service akmd /system/bin/logwrapper /sbin/akmd

system/core/下面的lib*库分析

编者：厦门大学信息学院通信工程系2015级研究生 史星辉， 电子工程系2015级研究生龙志军、江志明

1、lib*库分支



2、 libbacktrace库目录分析

2.1 libbacktrace/Android.mk

LOCAL_PATH := $(call my-dir) Android.mk 开始必须定义变量LOCAL PATH _PATH,它用来指定源文件的位置。 my-dir: 编译系统提供的'my-dir'宏函数，被用来获取当前的目录 The libbacktrace library. libbacktrace_src_files := \ Backtrace.cpp \ BacktraceCurrent.cpp \ BacktraceMap.cpp \ BacktracePtrace.cpp \ thread_utils.c \ ThreadEntry.cpp \ UnwindCurrent.cpp \ UnwindMap.cpp \ UnwindPtrace.cpp \ 定义变量LOCAL PATH _PATH, 它用来指定源文件的位置 include $(LOCAL_PATH)/Android.build.mk build_type := host libbacktrace_multilib := both include $(LOCAL_PATH)/Android.build.mk Special truncated libbacktrace library for mac. ifeq ($(HOST_OS),darwin) include $(CLEAR_VARS) LOCAL_MODULE := libbacktrace LOCAL_MODULE_TAGS := optional LOCAL_SRC_FILES := \ BacktraceMap.cpp \ LOCAL_MULTILIB := both include $(BUILD_HOST_SHARED_LIBRARY) 编译当前目录子目录下的，以及子目录的libc目录下的Android.mk文件。

2.2 libbinderwrapper/Android.mk

LOCAL_PATH := $(call my-dir) binderwrapperCommonCFlags := -Wall -Werror -Wno-unused-parameter binderwrapperCommonCFlags += -Wno-sign-promo # for libchrome binderwrapperCommonExportCIncludeDirs := $(LOCAL_PATH)/../include binderwrapperCommonCIncludes := $(LOCAL_PATH)/../include binderwrapperCommonSharedLibraries := \ libbinder \ libchrome \ libutils \ libbinderwrapper shared library include $(CLEAR_VARS) LOCAL_MODULE := libbinderwrapper LOCAL_CPP_EXTENSION := .cc LOCAL_CFLAGS := $(binderwrapperCommonCFlags) LOCAL_EXPORT_C_INCLUDE_DIRS := $(binderwrapperCommonExportCIncludeDirs) LOCAL_C_INCLUDES := $(binderwrapperCommonCIncludes) LOCAL_SHARED_LIBRARIES := $(binderwrapperCommonSharedLibraries) LOCAL_SRC_FILES := \ binder_wrapper.cc \ real_binder_wrapper.cc \ include $(BUILD_SHARED_LIBRARY) libbinderwrapper_test_support shared library include $(CLEAR_VARS) LOCAL_MODULE := libbinderwrapper_test_support LOCAL_CPP_EXTENSION := .cc LOCAL_CFLAGS := $(binderwrapperCommonCFlags) LOCAL_EXPORT_C_INCLUDE_DIRS := $(binderwrapperCommonExportCIncludeDirs) LOCAL_C_INCLUDES := $(binderwrapperCommonCIncludes) LOCAL_STATIC_LIBRARIES := libgtest LOCAL_SHARED_LIBRARIES := \ $(binderwrapperCommonSharedLibraries) \ libbinderwrapper \ LOCAL_SRC_FILES := \ binder_test_base.cc \ stub_binder_wrapper.cc \ include $(BUILD_SHARED_LIBRARY)

3、参考链接

http://wiki.qemu.org/Documentation/Networking
https://en.wikibooks.org/wiki/QEMU/Networking https://wiki.debian.org/QEMU
https://wiki.archlinux.org/index.php/QEMU#Networking
https://www.brobwind.com/archives/102
http://www.cnblogs.com/leaven/archive/2011/01/25/1944688.html

service分析

编者：厦门大学信息学院通信工程系2015级研究生孙甜、叶林飞

目录位置: frameworks/native/services/sensorservice/

1 service目录Android.mk文件分析

LOCAL_PATH := $(call my-dir) Android.mk 开始必须定义变量LOCAL_PATH，它用来指定源文件的位置。my-dir: 编译系统提供的'my-dir'宏函数，被用来获取当前的目录。

1.1 创建第一个模块

首先创建一个名为libsensorservice的第一个模块，代码如下：

```
include $(CLEAR_VARS)
```

编译系统提供CLEAR_VARS变量，它指向了一个用来清除LOCAL_XXX开头的变量（例如LOCAL_MODULE,LOCAL_SRC_FILES但是LOCAL_PATH除外）的makefile文件，需要它的原因是整个的编译上下文中，所有的变量都是全局的，这样就可以保证这些变量只在局部范围内起作用。

```
LOCAL_SRC_FILES:= \
BatteryService.cpp \
CorrectedGyroSensor.cpp \
Fusion.cpp \
GravitySensor.cpp \
LinearAccelerationSensor.cpp \
OrientationSensor.cpp \
RotationVectorSensor.cpp \
SensorDevice.cpp \
SensorFusion.cpp \
SensorInterface.cpp \
SensorService.cpp
```

LOCAL_SRC_FILES必须包含一系列的C/C++源文件，它将会被建立和装载到模块中。在该代码将以上源文件建立并装载到模块中。

```
LOCAL_CFLAGS:= -DLOG_TAG=\"SensorService\"
```

```
LOCAL_CFLAGS += -fvisibility=hidden
```

LOCAL_CFLAGS变量为C/C++编译器定义额外的标志，当编译C/C++源文件时传递一个可选的编译器标志，这对于指定额外的宏定义或编译选项很有用。相当于在所有源文件中增加一个#define LOG_TAG. 重要提示：尽量不要改变Android.mk中的优化/调试级别，这个可以通过在Application.mk中设置相应的信息来自动为你处理，并且会会让NDK生成在调试过程中使用的有用的数据文件。注意：在Android-ndk-1.5_r1中，只使用于C源文件，而不适用于C++源文件。在匹配所有Android build system的行为已经得到了纠正。（现在你可以为C++源文件使用LOCAL_CPPFLAGS来指定标志）它可以用LOCAL_CFLAGS += -I<path>来指定额外的包含路径，然而，如果使用LOCAL_C_INCLUDES会更好，因为用ndk-gdk进行本地调试的时候，那些路径依然是需要使用的。
在编译动态库时，想把一些不需要导出函数给隐藏起来。因此使用了-fvisibility属性。

```
LOCAL_SHARED_LIBRARIES := \
libcutils \libhardware \
```

LOCAL_SHARED_LIBRARIES变量用来列出模块所需的共享库的列表，不需要加上.so后缀。如上。

```
LOCAL_MODULE:= libsensorservice
```

创建一个LOCAL_MODULE，MODULE名字必须是唯一的并且不包含任何空格，编译系统将自动的修改生成的前缀和后缀，该代码创建了一个名为libsensorservice的LOCAL_MODULE。

```
include $(BUILD_SHARED_LIBRARY)
```

BUILD_SHARED_LIBRARY是编译系统提供的变量，指向一个GUN Makefile脚本，负责收集从上次调用include $(CLEAR_VARS)以来，定义在LOCAL_XXX变量中的所有信息，注意：在include这个脚本文件之前你必须至少已经定义了LOCAL_MODULE和LOCAL_SRC_FILES。注意：这会生成一个名为lib$(LOCAL_MODULE).so的动态库。至此完成模块libsensorservice的创建。

1.2生成可执行文件

include $(CLEAR_VARS)
在每个module中都要设置以LOCAL_开头的变量。它们会被include $(CLEAR_VARS)命令来清除，你会在你的很多module中使用这些LOCAL_开头的变量。
LOCAL_SRC_FILES:= \
main_sensorservice.cpp
LOCAL_SRC_FILES变量必须包含一系列将被构建和组合成模块的C/C++源文件。注意：不需要列出头文件或include文件，因为生成系统会为你自动计算出源文件的依赖关系。默认的C++源文件的扩展名是.cpp，但你可以通过定义LOCAL_DEFAULT_EXTENSION来指定一个扩展名。

LOCAL_SHARED_LIBRARIES := \
libsensorservice \
libbinder \
libutils
LOCAL_SHARED_LIBRARIES变量用来列出模块所需的共享库的列表，不需要加上.so后缀。如上。

LOCAL_MODULE_TAGS := optional
LOCAL_MODULE:= sensorservice
optional:指该模块在所有版本下都编译，
LOCAL_MODULE变量必须定义，用来标识在Android.mk文件中描述的每个模块。名称必须是唯一的，而且不包含任何空格。如果有其它moudle中已经定义了该名称，那么你在编译时就会报类似这样的错误：libgl2jni already defined by frameworks/base/opengl/tests/gl2_jni/jni. Stop. 注意：编译系统会自动产生合适的前缀和后缀

补充：变量命名的规范性

如果LOCAL_MODULE变量定义的值可能会被其它module调用时，就要考虑为其变量命名的规范性了。特别是在使用JNI时，既在LOCAL_JNI_SHARED_LIBRARIES变量中定义的值，最好要和LOCAL_MODULE变量定义的值保存一致(具体请参考LOCAL_JNI_SHARED_LIBRARIES变量的使用说明)。这时的LOCAL_MODULE变量的命名最好以lib开头，既“libxxx”。

include $(BUILD_EXECUTABLE)

include $(BUILD_EXECUTABLE)表示编译成可执行程序

2 参考文献

ledflasher 工程目录分析

编者：厦门大学信息学院通信工程系2015级研究生陈增贤、张恒伟、曾文婷

ledflasher/weaved.conf文件

config文件中定义了设备名字,生产商,设备模式,路径等一系列配置信息.

ledflasher/sepolicy文件

/sepolicy 提供了Android平台中的安全策略源文件。SEAndroid安全机制中的安全策略是在安全上下文的基础上进行描述的，也就是说，它通过主体和客体的安全上下文，定义主体是否有权限访问客体。
/sepolicy目录中的所有以.te为后缀的文件经过编译后，就会生成一个sepolicy文件，这个sepolicy文件会打包在ROM中，并且保存在设备上的根目录下。
文件 ledflasher.te定义了brillo_domain的访问权限：

```
allow_crash_reporter(ledflasher)
allow_call_weave(ledflasher)
allow ledflasher example_led_service:service_manager find;
```

ledflasher/out文件

编译完成后，将在根目录中生成一个out文件夹，所有生成的内容均放置在这个文件夹中。
out文件夹目录结构如下所示：
out/
-- CaseCheck.txt
-- casecheck.txt
-- host
    -- common
    -- linux-x86/darwin-x86
-- target（如果编译时候定义TARGET_STRIP_MODULE=false的话，这个目录为debug/target）
    -- common

-- product
主要的两个目录为host和target，前者表示在主机（x86）生成的工具，后者表示目标机（模认为ARMv5）运行的内容。

host目录的结构如下所示：
out/host/
-- common
-- obj（Java库）
-- linux-x86/darwin-x86
-- bin（二进制程序）
-- framework（JAVA库,＊.jar文件）
-- lib（共享库*.so）
-- obj（中间生成的目标文件）
host目录是一些在主机上用的工具，有一些是二进制程序,有一些是JAVA的程序。

target目录的结构如下所示：
out/target/
-- common
-- R （资源文件）
-- docs
-- obj （目标文件）
-- APPS（包含了JAVA应用程序生成的目标，每个应用程序对应其中一个子目录，将结合每个应用程序的原始文件生成Android应用程序的APK包）
-- JAVA_LIBRARIES（包含了JAVA的库，每个库对应其中一个子目录）
-- product
-- generic
-- android-info.txt
-- clean_steps.mk
-- data（存放数据的文件系统）
-- obj
-- APPS（包含了各种JAVA应用，与common/obj/APPS相对应，但是已经打成了APK包）
-- SHARED_LIBRARIES（存放所有动态库）
-- STATIC_LIBRARIES（存放所有静态库）
-- ramdisk.img（内存盘的根文件系统映像）
-- root
-- symbols
-- system（存放主要的文件系统）
-- system.img（文件系统的映像）

-- userdata-qemu.img（模拟器使用的数据文件）
-- userdata.img（数据内容映像）
其中common目录表示通用的内容，product中则是针对产品的内容。

ledflasher/AndroidProducts.mk

```
PRODUCT_MAKEFILES := $(LOCAL_DIR)/ledflasher.mk
```

文件中定义了Android产品文件ledflasher的路径。

ledflasher/ledflasher.mk

```
CFGTREE_ROOT := $(LOCAL_PATH)     //为当前模块的相对路径，必须出现在所有的编译模块之前。
include device/generic/brillo/brillo_base.mk    //引入该路径下的makefile文件
PRODUCT_NAME := $(call get_product_name_from_file)
ifneq ($(PRODUCT_NAME),$(call cfgtree-get,name))
  $(error config/name, <name>.mk, and AndroidProducts.mk must all be updated together.
)
endif
PRODUCT_BRAND := $(call cfgtree-get,brand)
PRODUCT_DEVICE := $(call cfgtree-get,device)
PRODUCT_MANUFACTURER := $(call cfgtree-get,manufacturer)
PRODUCT_PACKAGES += $(call cfgtree-get-if-exists,packages)
PRODUCT_COPY_FILES += $(addprefix $(LOCAL_PATH)/,$(call cfgtree-get-if-exists,copy_fil
es))
BOARD_SEPOLICY_DIRS := $(BOARD_SEPOLICY_DIRS) $(LOCAL_PATH)/sepolicy
-include $(LOCAL_PATH)/extras.mk
```

ledflasher.mk中包含很多特定的产品变量。产品特定的变量被定义在产品说明文件(definition files)中。产品说明文件可以继承自其它的产品说明文件。这样将便于管理。
产品说明文件中可以包含如下变量：

Parameter	Description
PRODUCT_NAME	End-user-visible name for the overall product. Appears in the "About the phone" info.
PRODUCT_MODEL	End-user-visible name for the end product
PRODUCT_LOCALES	A space-separated list of two-letter language code, two-letter country code pairs that describe several settings for the user, such as the UI language and time, date and currency formatting. The first locale listed in PRODUCT_LOCALES is is used if the locale has never been set before.
PRODUCT_PACKAGES	Lists the APKs to install.
PRODUCT_DEVICE	Name of the industrial design
PRODUCT_MANUFACTURER	Name of the manufacturer
PRODUCT_BRAND	The brand (e.g., brillo) the software is customized for
PRODUCT_PROPERTY_OVERRIDES	List of property assignments in the format "key=value"
PRODUCT_COPY_FILES	List of words like source_path:destination_path. The file at the source path should be copied to the destination path when building this product. The rules for the copy steps are defined in config/Makefile
PRODUCT_OTA_PUBLIC_KEYS	List of OTA public keys for the product
PRODUCT_TAGS	list of space-separated words for a given product

ledflasher/Android.mk

```
LOCAL_PATH := $(call my-dir)   //Android.mk开始必须定义变量 LOCAL_PATH,它用来指定源文件的位置
                                    my-dir: 编译系统提供的'my-dir'宏函数,被用来获取当前的目录
include $(CLEAR_VARS)          //模块开始
LOCAL_MODULE := ledservice
LOCAL_INIT_RC := ledservice.rc
LOCAL_SRC_FILES
LOCAL_SHARED_LIBRARIES
LOCAL_STATIC_LIBRARIES
LOCAL_CLANG := true
LOCAL_C_INCLUDES := external/gtest/include
LOCAL_CFLAGS := -Wall -Werror
include $(BUILD_EXECUTABLE)     //模块结束标志
```

Android.mk基本组成:

1. LOCAL_PATH 定义了当前模块的相对路径，必须出现在所有的编译模块之前。
2. 每个编译模块由include $(CLEAR_VARS) 开始，由include $(BUILD_XXX) 结束。
3. include $(CLEAR_VARS) 是一个编译模块的开始，它会清空除LOCAL_PATH之外的所有LOCA_XXX变量。
4. include $(BUILD_XXX) 描述了编译目标。
5. LOCAL_SRC_FILES 定义了本模块编译使用的源文件，采用的是基于LOCAL_PATH的相对路径。
6. LOCAL_MODULE 定义了本模块的模块名。
7. LOCAL_STATIC_LIBRARIES 表示编译本模块时需要链接的静态库。
8. LOCAL_C_INCLUDES 表示了本模块需要引用的include文件。

编译acp还需要了几个可选的变量：

1. LOCAL_PATH 定义了当前模块的相对路径，必须出现在所有的编译模块之前。
2. LOCAL_C_INCLUDES 表示了本模块需要引用的include文件。
3. LOCAL_ACP_UNAVAILABLE 表示是否支持acp，如果支持acp，则使用acp进行拷贝，否则使用linux cp拷贝，本模块编译acp，当然是不支持acp了。

参考链接

1.Android.mk 常用宏和变量
http://jingyan.baidu.com/article/37bce2be1659c81002f3a2b
2.Android.mk文件语法规范及使用模板
http://www.cnblogs.com/leaven/archive/2011/01/25/1944688.htm
3.Android Makefile 文件分析
http://blog.csdn.net/wh_19910525/article/details/7993266
4.Android 编译 out文件分析

http://blog.csdn.net/guiwang2008/article/details/7353442

5.Application makefile文件编写规则

http://blog.csdn.net/fengbingchun/article/details/38705519

HAL层分析

编者：厦门大学航空航天学院机电系2015级研究生王凯，能源学院能效工程2015级研究生王猛

目录：bdk/hardware/bsp/intel/peripheral/light/mraa

1.open_lights函数

函数目的：打开背光灯

```
    static int open_lights(const struct hw_module_t *module, char const *name,
        struct hw_device_t **device)
    {
    struct light_device_ext_t *dev;
    int rc = 0, type = -1;
```

```
ALOGV("%s: Opening %s lights module", __func__, name);
```

首先定义了一个struct light_device_ext_t *dev;

```
ALOGV("%s: Opening %s lights module", __func__, name);
```

这个是打印相关的信息，首先我们需要定义一个TAG;

比如：

```
#define LOG_TAG "lights"
```

这个LOG_TAG在21行定义过。这样将来logcat出来的东西就是这样的：

```
V/lights(422):hardware/bsp/intel/peripheral/light/mraa/lights.c: Opening ... lights mo
dule
```

这里的就是传进来的name参数。

//

```
if (0 == strcmp(LIGHT_ID_NOTIFICATIONS, name)) {
        type = NOTIFICATIONS_TYPE;
    } else {
        return EINVAL;
    }
```

接着看传进来的name参数是不是LIGHT_ID_NOTIFICATIONS，这个定义为：

```
#define LIGHT_ID_NOTIFICATIONS      "notifications"
```

如果不相等，那么直接返回打开不成功标志EINVAL。

//

然后初始化**dev**变量

```
    dev = (struct light_device_ext_t *)(light_devices + type);

    pthread_mutex_lock(&dev->write_mutex);

    if (dev->refs != 0) {
        /* already opened; nothing to do */
        goto inc_refs;
    }
```

接下来是先将线程互斥锁定，然后在进行操作，这样比较安全，最后记得取消锁定。 之后就判断dev是否已经打开了，判断的方法是判断refs的引用次数，如果引用次数大于0，那么就表示已经打开了，然后直接引用加1返回打开成功标志。

//

接下来是模型初始化

```
    rc = init_module(type);
    if (rc != 0) {
        ALOGE("%s: Failed to initialize lights module", __func__);
        goto mutex_unlock;
    }
```

模型初始化是将全局变量

```
struct light_device_ext_t light_devices[] = {[ 0 ... (LIGHTS_TYPE_NUM - 1) ] = { .writ
e_mutex = PTHREAD_MUTEX_INITIALIZER}};
```

给初始化。如果初始化失败，那么打印出失败信息，同时返回rc失败标志。

//

接下来是背光灯同步源初始化，也就是创建一个新的线程

```
    rc = init_light_sync_resources(&dev->flash_cond,&dev->flash_signal_mutex);
       if (rc != 0)
       {
         goto mutex_unlock;
       }
```

同样如果没有成功就返回rc失败标志。

//

接下来是设备信息更新，给**dev**变量赋值。接下来是引用加**1**，然后线程锁解锁。

```
     dev->base_dev.common.tag = HARDWARE_DEVICE_TAG;
     dev->base_dev.common.version = 0;
     dev->base_dev.common.module = (struct hw_module_t *)module;
     dev->base_dev.common.close = (int (*)(struct hw_device_t *))close_lights;
     dev->base_dev.set_light = set_light_generic;
     inc_refs:
     dev->refs++;
     *device = (struct hw_device_t *)dev;
     mutex_unlock:
     pthread_mutex_unlock(&dev->write_mutex);
     return rc;
```

2.close_lights函数

函数目的：关闭背光灯

//

首先对传进来的参数进行正确性判断，如果是**NULL**就打印失败信息，同时返回关闭失败标志。

```
     static int close_lights(struct light_device_t *base_dev)
       {
     struct light_device_ext_t *dev = (struct light_device_ext_t *)base_dev;
       int rc = 0;

       if (dev == NULL)
       {
           ALOGE("%s: Cannot deallocate a NULL light device", __func__);
           return EINVAL;
       }
```

//

接下来是判断引用.

```
if (dev->refs == 0)
 /* the light device is not open */
  {
          rc = EINVAL;
          goto mutex_unlock;
  }
  else if (dev->refs > 1)
  {
          goto dec_refs;
  }
```

如果refs引用次数为0表示设备根本没打开。这样可以直接返回。 否则如果被引用次数大于1，那么只需要减少一次引用次数，然后直接返回。 否则，也就是如果只被一个程序打开，那就就需要关闭设备了。

//

接下来就关闭设备，如果是闪烁模式，那么则销毁闪烁线程，将**flashMode**设置为未闪烁模式。

```
  if (dev->state.flashMode)
         {
          /* destroy flashing thread */
          pthread_mutex_lock(&dev->flash_signal_mutex);
          dev->state.flashMode = LIGHT_FLASH_NONE;
          pthread_cond_signal(&dev->flash_cond);
          pthread_mutex_unlock(&dev->flash_signal_mutex);
          pthread_join(dev->flash_thread, NULL);
         }
     free_light_sync_resources(&dev->flash_cond,&dev->flash_signal_mutex);
```

然后将相关的设置更新，接着释放背光灯同步源。实质上就是销毁线程。

```
    static void free_light_sync_resources(pthread_cond_t *cond,pthread_mutex_t *signal
_mutex)
      {
          pthread_mutex_destroy(signal_mutex);
          pthread_cond_destroy(cond);
      }
```

//

然后将引用次数减**1**，再取消线程互斥锁，然后返回成功标志。

3.set_light_generic函数

函数目的：设置背光灯状态

//

```
      static int set_light_generic(struct light_device_t *base_dev,struct light_state_
t const *state)
      {
        struct light_device_ext_t *dev = (struct light_device_ext_t *)base_dev;
        struct light_state_t *current_state;
        int rc = 0;
      if (dev == NULL)
        {
          ALOGE("%s: Cannot set state for NULL device", __func__);
          return EINVAL;
        }
      current_state = &dev->state;
      pthread_mutex_lock(&dev->write_mutex);
      if (dev->refs == 0)
        {
          ALOGE("%s: The light device is not opened", __func__);
          pthread_mutex_unlock(&dev->write_mutex);
          return EINVAL;
        }
```

首先还是检查设备信息，如果传入参数是无效参数，或者设备没有打开，那么则进行相关的错误处理。 //

接下来直接打印设备的信息，包括闪烁模式，颜色

```
   ALOGV("%s: flashMode:%x, color:%x", __func__, state->flashMode, state->color);
```

//

接下来如果现在的状态是闪烁模式，那么销毁闪烁线程，更新状态。

```
if (current_state->flashMode)
       {
           /* destroy flashing thread */
           pthread_mutex_lock(&dev->flash_signal_mutex);
           current_state->flashMode = LIGHT_FLASH_NONE;
           pthread_cond_signal(&dev->flash_cond);
           pthread_mutex_unlock(&dev->flash_signal_mutex);
           pthread_join(dev->flash_thread, NULL);
       }
```

//

接着对灯光进行设置。

```
    *current_state = *state;
    if (dev->transform != NULL)
       {
        current_state->color = dev->transform(current_state->color);
        }
    if (current_state->flashMode)
        {
          /* start flashing thread */
          if (check_flash_state(current_state) == 0)
            {
              rc = pthread_create(&dev->flash_thread, NULL,flash_routine, (void *)dev)
;
                if (rc != 0)
                {
                    ALOGE("%s: Cannot create flashing thread", __func__);
                    current_state->flashMode = LIGHT_FLASH_NONE;
                }
            } else
         {
            ALOGE("%s: Flash state is invalid", __func__);
            current_state->flashMode = LIGHT_FLASH_NONE;
        }
        }
    else
        {
            rc = set_gpio_value(dev->pin, current_state->color);
            if (rc != 0)
            {
                ALOGE("%s: Cannot set light color.", __func__);
            }
        }
```

如果需要设置的是闪烁状态，那么先创建一个闪烁的线程。否则如果不需要闪烁，那么直接就设置颜色就好。

//

```
    pthread_mutex_unlock(&dev->write_mutex);

    return rc;
}
```

最后解锁互斥锁，然后返回设置成功标志**rc**。

First Chapter

GitBook allows you to organize your book into chapters, each chapter is stored in a separate file like this one.