

第七章 嵌入式 Android 实验

7.1 编译 U-Boot 实验

7.1.1 实验目的

掌握 U-BOOT 编译方法。

7.1.2 实验内容

编译 ARM Android 的 u-boot

7.1.3 实验设备

硬件：配置完开发环境 ubuntu 虚拟机的 PC 机 1 台

软件：U-boot 源码包

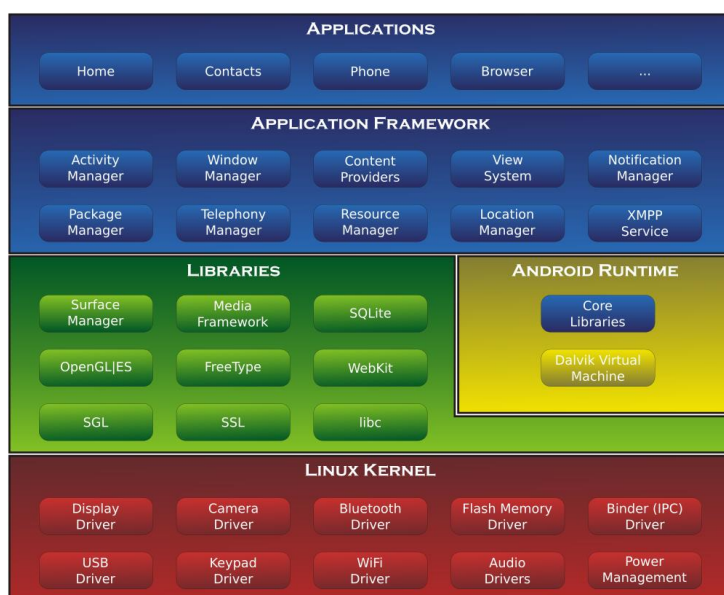
7.1.4 实验原理

一、Android 系统架构

Android 是一种基于 Linux 的自由及开放源代码的操作系统，主要适用于移动设备，如智能手机和平板电脑，由 Google 公司和开放手机联盟领导及开发。Android 并不是传统的 linux 风格的一个规范或分发版本，也不是一系列可重用的组件集成，Android 是一个用于连接设备的软件块。

Android 系统只用了 Linux kernel，别的东西都没用。这就是说，与 Ubuntu、Debian、Redhat 这样的传统 Linux 发行版相比，只有系统的底层结构是一样的，其他东西在 Android 里都不一样，尤其是程序员的编程接口是完全不同的。由于商业原因 google 特意修改 Kernel，使得原本应该包括在 kernel 中的某些功能，都被转移到了 userspace 之中，已达到绕过 GPL 许可。目前 Android 已被 LINUX 开源社区删除，也就是说无法从 kernel 官网下载 android 源码，需从 google 下载。

Android 系统相当于是一个应用系统或者说是一个应用程序，其具备完整的架构并且包含各种复杂的应用。Android 采用层次化系统架构，官方公布的标准架构如下图所示



Android 由底层往上分为 4 个主要功能层，分别是 linux 内核层(Linux Kernel)，系统运行时库层(Libraries

和 Android Runtime)，应用程序架构层（Application Framework）和应用程序层（Applications）。

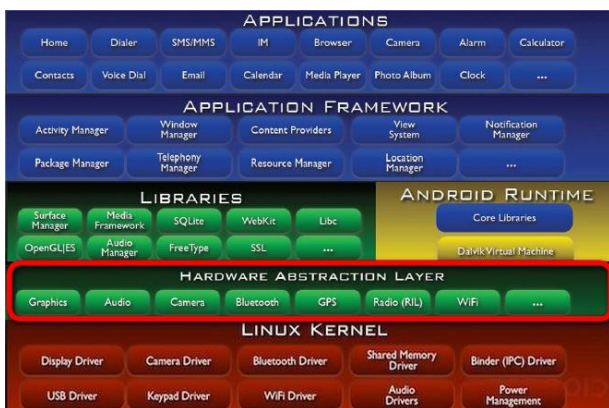
1、Linux 内核层

Android 以 Linux 操作系统内核为基础，借助 Linux 内核服务实现硬件设备驱动，进程和内存管理，网络协议栈，电源管理，无线通信等核心功能。Android4.0 版本之前基于 Linux2.6 系列内核，4.0 及之后的版本使用更新的 Linux3.X 内核，并且两个开源项目开始有了互通。Linux3.3 内核中正式包括一些 Android 代码，可以直接引导进入 Android。Linux3.4 增添了电源管理等更多功能，以增加与 Android 的硬件兼容性，使 Android 在更多设备上得到支持。直到现在最新的 android6.0 仍然继续延用着 linux3.4.0。

Android 内核对 Linux 内核进行了增强，增加了一些面向移动计算的特有功能。例如，低内存管理器 LMK（Low Memory Keller），匿名共享内存（Ashmem），以及轻量级的进程间通信 Binder 机制等。这些内核的增强使 Android 在继承 Linux 内核安全机制的同时，进一步提升了内存管理，进程间通信等方面的安全性。

2、硬件抽象层

内核驱动和用户软件之间还存在所谓的硬件抽象层（Hardware Abstract Layer,HAL），它是对硬件设备的具体实现加以抽象。HAL 没有在 Android 官方系统架构图中标明，下图标出了硬件抽象层在 android 系统中的位置：



鉴于许多硬件设备厂商不希望公开其设备驱动的源代码，如果能将 android 的应用框架层与 linux 系统内核的设备驱动隔离，使应用程序框架的开发尽量独立于具体的驱动程序，则 android 将减少对 Linux 内核的依赖。HAL 由此而生，它是对 Linux 内核驱动程序进行的封装，将硬件抽象化，屏蔽掉了底层的实现细节。HAL 规定了一套应用层对硬件层读写和配置的统一接口，本质上就是将硬件的驱动分为用户空间和内核空间两个层面；Linux 内核驱动程序运行于内核空间，硬件抽象层运行于用户空间。

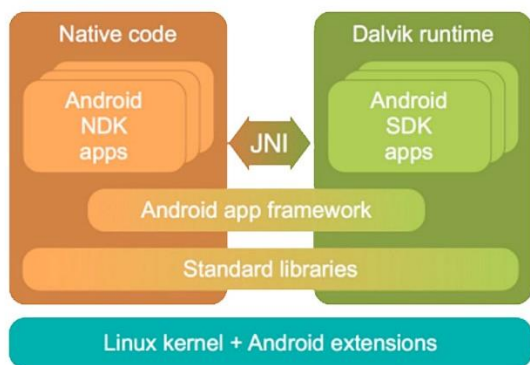
3、系统运行库层

● 系统类库

官方的系统架构图中，位于 Linux 内核层之上的系统运行库层是应用程序框架的支撑，为 Android 系统中的各个组件提供服务。系统运行库层由系统类库和 Android 运行时构成。系统类库大部分由 C/C++编写，所提供的功能通过 Android 应用程序框架为开发者所使用。

除了一些常见的系统类库外，Android 还提供了 Android NDK（Native Development Kit），即 Android 原生库。NDK 为开发者提供了直接使用 Android 系统资源，并采用 C 或 C++语言编写程序的接口。因此，第三方应用程序可以不依赖于 Dalvik 虚拟机进行开发。实际上，NDK 提供了一系列从 C 或 C++生成原生代码所需要的工具，为开发者快速开发 C 或 C++的动态库提供方便，并能自动将生成的动态库和 java 应用程序一起打包成应用程序包文件，即.apk 文件。

NDK 和 SDK 的关系如下图所示，ndk 可以通过 native code 跨过使用 dalvik runtime,直接调用到 android 内核资源，而 sdk 则需要在 dalvik runtime 环境下才能调用到内核资源。然而两者并不是各司其职，各不相关。android 提供了 JNI(java native interface)使两者可以进行相互调用和通信。



● 运行时

Android 运行时包含核心库和 Dalvik 虚拟机两部分。

- **核心库：**核心库提供了 Java5 se API 的多数功能，并提供 Android 的核心 API，如 android.os, android.net, android.media 等。
- **Dalvik 虚拟机：**Dalvik 虚拟机是基于 apache 的 java 虚拟机，并被改进以适应低内存，低处理器速度的移动设备环境。Dalvik 虚拟机依赖于 Linux 内核，实现进程隔离与线程调试管理，安全和异常管理，垃圾回收等重要功能。

本质而言，Dalvik 虚拟机并非传统意义上的 java 虚拟机（JVM）。Dalvik 虚拟机不仅不按照 Java 虚拟机的规范来实现，而且两者不兼容。

Dalvik 和标准 Java 虚拟机有以下主要区别：

- Dalvik 基于寄存器，而 JVM 基于栈。一般认为，基于寄存器的实现虽然更多依赖于具体的 CPU 结构，硬件通用性稍差，但其使用等长指令，在效率速度上较传统 JVM 更有优势。
- Dalvik 经过优化，允许在有限的内存中同时高效地运行多个虚拟机的实例，并且每一个 Dalvik 应用作为一个独立的 Linux 进程执行，都拥有一个独立的 Dalvik 虚拟机实例。Android 这种基于 Linux 的进程“沙箱”机制，是整个安全设计的基础之一。
- Dalvik 虚拟机从 DEX（Dalvik Executable）格式的文件中读取指令与数据，进行解释运行。DEX 文件由传统的，编译产生的 CLASS 文件，经 dx 工具软件处理后生成。
- Dalvik 的 DEX 文件还可以进一步优化，提高运行性能。通常，OEM 的应用程序可以在系统编译后，直接生成优化文件（.ODEX）；第三方的应用程序则可在运行时在缓存中优化与保存，优化后的格式为 DEY（.dey 文件）。

这部分内容，即从 android4.4 开始就出现了 ART（android runtime），这个 ART 是一种用来代替 Dalvik 的新型运行环境。当然在 4.4 的正式环境中用的还是 Dalvik，真正开始用 ART 取代 Dalvik 是从 android5.0 开始的。

4、应用程序框架层

应用程序框架层提供开发 Android 应用程序所需的一系列类库，使开发人员可以进行快速的应用程序开发，方便重用组件，也可以通过继承实现个性化的扩展。具体包括的模块如下表：

应用程序框架层类库名称	功能
活动管理器(Activity Manager)	管理各个应用程序生命周期并提供常用的导航回退功能，为所有程序的窗口提供交互的接口
窗口管理器(Window Manager)	对所有开启的窗口程序进行管理
内容提供者(Content Provider)	提供一个应用程序访问另一个应用程序数据的功能，或者实现应用程序之间的数据共享
视图系统(View System)	创建应用程序的基本组件，包括列表（lists），网格（grids），文本框（text boxes），按钮（buttons），还有可嵌入的 web 浏览器。

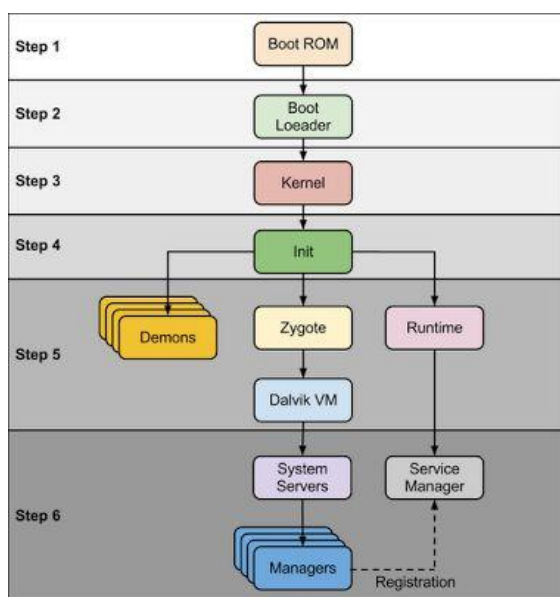
通知管理器(Notification Manager)	使应用程序可以在状态栏中显示自定义的客户提示信息
包管理器(Package Manager)	对应用程序进行管理，提供的功能诸如安装应用程序，卸载应用程序，查询相关权限信息等。
资源管理器(Resource Manager)	提供各种非代码资源供应用程序使用，如本地化字符串，图片，音频等
位置管理器(Location Manager)	提供位置服务
电话管理器(Telephony Manager)	管理所有的移动设备功能
XMPP 服务	是 Google 在线即时交流软件中一个通用的进程，提供后台推送服务

5、应用层

Android 平台的应用层上包括各类与用户直接交互的应用程序，或由 java 语言编写的运行于后台的服务程序。例如，智能手机上实现的常见基本功能程序，诸如 SMS 短信，电话拨号，图片浏览器，日历，游戏，地图，web 浏览器等程序，以及开发人员开发的其他应用程序。

二、Android 启动流程

了解了 Android 的整体架构，我们知道 Android 并不是 linux。它没有本地窗口系统，它没有 glibc 支持，它不包括一整套标准的 Linux 使用程序，他有自己专有的驱动程序。虽然 Android 不是 linux,但是它是基于 linux 内核的“操作系统”。因此我们移植 Android 也和 linux 的关系比较大。Android 的启动流程如下图



第一步：系统启动

当电源按下，引导芯片代码开始从预定义的地方（固化在 ROM）开始执行。加载引导程序到 RAM，然后执行。

第二步：引导程序

引导程序是在 Android 操作系统开始运行前的小程序。引导程序是运行的第一个程序，因此它是针对特定的主板与芯片的。一般情况下使用通用的引导程序比如 uboot，它不是 Android 操作系统的一部分。

引导程序分两个阶段执行。第一个阶段，检测外部的 RAM 以及加载对第二阶段有用的程序；第二阶段，引导程序设置网络、内存等等。这些对于运行内核是必要的，为了达到特殊的目标，引导程序可以根据配置参数或者输入数据设置内核。

默认情况原生 Android 引导程序可以在\bootable\bootloader\legacy\usbloader 找到。由于我们使用的 U-Boot 引导，为避免混淆，我们删除了原生的代码。一般情况下 bootloader 中 init.s 函数初始化堆栈，清零 BBS 段，调用 main.c 的 _main() 函数；main.c 初始化硬件，创建 linux 标签。

第三步：内核

Android 内核与桌面 linux 内核启动的方式差不多。内核启动时，设置缓存、被保护存储器、计划列表，加载驱动。当内核完成系统设置，它首先在系统文件中寻找”init”文件，然后启动 root 进程或者系统的第一个进程。

第四步：init 进程

init 是第一个进程，我们可以说它是 root 进程或者说有进程的父进程。init 进程有两个责任，一是挂载目录，比如/sys、/dev、/proc 目录，二是运行 init.rc 脚本。init 进程可以在/system/core/init 找到。init.rc 文件可以在/system/core/rootdir/init.rc 找到。对于 init.rc 文件，Android 中有特定的格式以及规则。在 Android 中，我们叫做 Android 初始化语言。

Android 初始化语言由四大类型的声明组成，即 Actions（动作）、Commands（命令）、Services（服务）、以及 Options（选项）。

- Action（动作）：动作是以命令流程命名的，有一个触发器决定动作是否发生。

语法：on <trigger>
 <command>
 <command>

- Service（服务）：服务是 init 进程启动的程序、当服务退出时 init 进程会视情况重启服务。

语法：service <name> <pathname> [<argument>]*
 <option>
 <option>

- Options（选项）：选项是对服务的描述。它们影响 init 进程如何以及何时启动服务。

默认的 init.rc 文件如下所示

Action/Service	描述
on early-init	设置 init 进程以及它创建的子进程的优先级，设置 init 进程的安全环境
on init	设置全局环境，为 cpu accounting 创建 cgroup(资源控制)挂载点
on fs	挂载 mtd 分区
on post-fs	改变系统目录的访问权限
on post-fs-data	改变/data 目录以及它的子目录的访问权限
on boot	基本网络的初始化，内存管理等
service servicemanager	启动系统管理器管理所有的本地服务，比如位置、音频、Shared preference 等等...
service zygote	启动 zygote 作为应用进程

在这个阶段你可以在设备的屏幕上看到“Android”logo 了。

第五步：Zygote 进程

在 Java 中，我们知道不同的虚拟机实例会为不同的应用分配不同的内存。假如 Android 应用想要尽可能快地启动，Android 系统需要为每一个应用启动不同的 Dalvik 虚拟机实例，就会消耗大量的内存以及时间。因此，为了克服这个问题，Android 系统创造了”Zygote”。

Zygote 让 Dalvik 虚拟机共享代码、低内存占用以及最小的启动时间成为可能。Zygote 是一个虚拟机进程，正如我们在前一个步骤所说的在系统引导的时候启动。Zygote 预加载以及初始化核心库类。通常，这些核心类一般是只读的，也是 Android SDK 或者核心框架的一部分。在 Java 虚拟机中，每一个实例都有它自己的核心库类文件和堆对象的拷贝。

Zygote 加载进程

- 加载 ZygoteInit 类，源代码：/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
- registerZygoteSocket() 为 zygote 命令连接注册一个服务器套接字。
- preloadClassed “preloaded-classes” 是一个简单的包含一系列需要预加载类的文本文件，你可以在 <Android Source>/frameworks/base 找到 “preloaded-classes” 文件。

➤ `preloadResources()` `preloadResources` 也意味着本地主题、布局以及 `android.R` 文件中包含的所有东西都会用这个方法加载。

在这个阶段，你可以看到启动动画。

第六步：系统服务或服务

完成了上面几步之后，运行环境请求 `Zygote` 运行系统服务。系统服务同时使用 `native` 以及 `java` 编写，系统服务可以认为是一个进程。同一个系统服务在 `Android SDK` 可以以 `System Services` 形式获得。系统服务包含了所有的 `System Services`。

`Zygote` 创建新的进程去启动系统服务。你可以在 `ZygoteInit` 类的 “`startSystemServer`” 方法中找到源代码。

核心服务：启动电源管理器；创建 `Activity` 管理器；启动电话注册；启动包管理器；设置 `Activity` 管理服务为系统进程；启动上下文管理器；启动系统 `Context Providers`；启动电池服务；启动定时管理器；启动传感服务；启动窗口管理器；启动蓝牙服务；启动挂载服务。

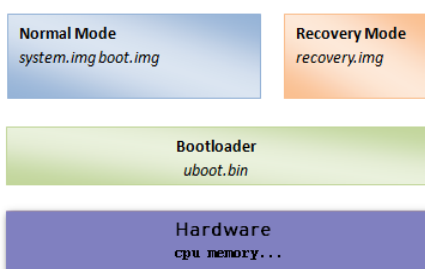
其他服务：启动状态栏服务；启动硬件服务；启动网络状态服务；启动网络连接服务；启动通知管理器；启动设备存储监视服务；启动定位管理器；启动搜索服务；启动剪切板服务；启动登记服务；启动壁纸服务；启动音频服务；启动耳机监听；启动 `AdbSettingsObserver`（处理 `adb` 命令）。

第七步：引导完成

一旦系统服务在内存中跑起来了，`Android` 就完成了引导过程。在这个时候 “`ACTION_BOOT_COMPLETED`” 开机启动广播就会发出去。

三、Bootloader 启动流程

`Android` 设备从硬件到系统的结构如下图



最底层的是各种硬件设备，往上一层是 `Bootloader`，`Bootloader` 代码是芯片复位后进入操作系统之前执行的一段代码，主要用于完成由硬件启动到操作系统启动的过渡，从而为操作系统提供基本的运行环境，如初始化 `CPU`、堆栈、存储器系统等，它可以将系统的软硬件环境配置到一个合适状态，为运行操作系统做好准备。它的任务不多，终极目标就是把 `OS` 引导起来运行。

`Bootloader` 代码与 `CPU` 芯片的内核结构、具体型号、应用系统的配置及使用的操作系统等因素有关，其功能类似于 `PC` 机的 `BIOS` 程序。由于 `bootloader` 和 `CPU` 及电路板的配置情况有关，因此不可能有通用的 `bootloader`，开发时需要用户根据具体情况进行移植。

在嵌入式系统世界里存在各种各样的 `Bootloader`，种类划分也有多种方式。除了按照处理器体系结构不同划分以外，还有功能复杂程度的不同。对于每种体系结构，都有一系列开放源码 `Bootloader` 可以选用。

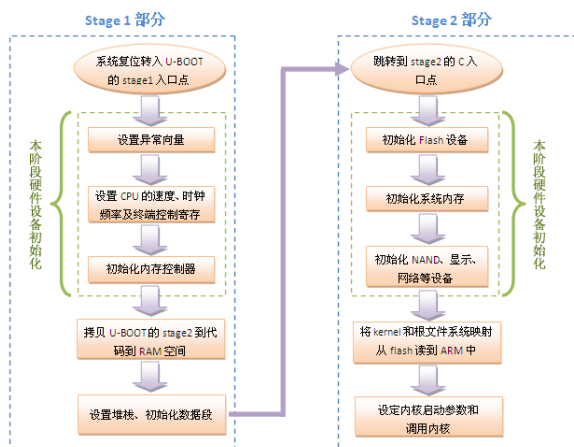
- `X86`： `X86` 的工作站和服务器的上一般使用 `LILO` 和 `GRUB` 引导。
- `ARM`： 最早有为 `ARM720` 处理器开发板所做的固件，又有了 `armboot`， `StrongARM` 平台的 `blob`，还有 `S3C2410` 处理器开发板上的 `vivi` 等。现在 `armboot` 已经并入了 `U-Boot`，所以 `U-Boot` 也支持 `ARM/XSCALE` 平台。 `U-Boot` 已经成为 `ARM` 平台事实上的标准 `Bootloader`。
- `PowerPC`： 最早使用于 `ppcboot`，不过现在大多数直接使用 `U-boot`。
- `MIPS`： 最早都是 `MIPS` 开发商自己写的 `bootloader`，不过现在 `U-boot` 也支持 `MIPS` 架构。

- M68K: Redboot 能够支持 m68k 系列的系统。

嵌入式系统中常用的 Bootloader 有 armboot、redboot、blob、u-boot、Bios-lt、Bootldr 等，其中 U-Boot 是当前比较流行，功能比较强大的 bootloader，可以支持多种体系结构，但相对也比较复杂。bootloader 的实现依赖于 CPU 的体系结构。

由于不同处理器芯片厂商对 ARM Core 的封装差异比较大，所以不同的 ARM 处理器，对于上电引导都是由特定处理器芯片厂商自己开发的程序，这个上电引导程序通常比较简单，会初始化硬件，提供下载模式等，然后才会加载通常的 bootloader。不过也不是所有的处理都是这样，比如三星早起的 S3C24X0 系列，它的 bootROM 直接跳到 U-boot 中执行，首先由 bootROM 将 U-boot 的前 4KB 拷贝到处理器 ISRAM，接着在 U-boot 的前 4KB 中必须保证要完成的两项主要工作：初始化 DDR，nand 和 nand 控制器，接着将 U-boot 剩余的 code 拷贝到 SDRAM 中，然后跳到 SDRAM 的对应地址上去继续跑 U-boot。这种方式比较简单直白，但是太裸了，就好像不穿衣服给比人看得一清二楚，不安全，所以后来三星 exynos 系列搞了 bl1、bl2、2ndboot 等，如 S5P6818 平台的 bootloader 方案为：Samsung (S5P6818)：iROM + 2ndBoot + U-boot，商业化了许多。

U-boot 的启动过程，大致上可以分成 stage1 和 stage2 两大阶段：第一阶段，汇编代码；第二阶段，c 代码。



1、stage1 (start.s 代码结构)--基本的硬件初始化

通过 arch/arm/cpu/slsiap/u-boot.lds 链接脚本可知，第一条指令从 arch/arm/cpu/slsiap/s5p6818/start.S 文件开始

```

u-boot.lds x
10 OUTPUT_FORMAT("elf32-little", "elf32-little", "elf32-little")
11 ENTRY(_start)
12 ENTRY(_start)
13
14 {
15     . = 0x00000000;
16     . = ALIGN(4);
17     .text :
18
  
```

arch/arm/cpu/slsiap/u-boot.lds

```

start.S x
20 .globl _start
21 _start:
22     b reset
23     ldr pc, _undefined_instruction
24     ldr pc, _software_interrupt
25     ldr pc, _prefetch_abort
26     ldr pc, _data_abort
27     ldr pc, _not_used
28     ldr pc, _irq
29     ldr pc, _fiq
30
31 _undefined_instruction: .word undefined_instruction
32 _software_interrupt: .word software_interrupt
33 _prefetch_abort: .word prefetch_abort
34 _data_abort: .word data_abort
35 _not_used: .word not_used
36 _irq: .word irq
37 _fiq:
38
39     .balign 16
  
```

arch/arm/cpu/slsiap/s5p6818/start.S

第一阶段主要做了如下事情：

(1)、首先运行 arch/arm/cpu/slsiap/s5p6818/start.S 里面的 _stext 函数，跳转到 reset 复位处理函数，设置处理器 SVC 模式，关闭 IRQ 和 FIQ 中断。

(2)、如果没有定义 CONFIG_SKIP_LOWLEVEL_INIT 则设置 cp15 协处理器，设置异常向量表。跳转到 cpu_init_cp15 初始化协处理器，清除 TLB，关闭 cache，关闭 MMU。如果没有定义 CONFIG_SYS_ICACHE_OFF 则打开 icache。继续执行 cpu_init_crit，跳转到 arch/arm/cpu/slsiap/s5p6818/low_init.S 里面的 lowlevel_init，进行时钟初始化，DDR 初始化。

(3)、如果定义 CONFIG_RELOC_TO_TEXT_BASE 则加载 u-boot 到 SDRAM，执行 board_init_f 函数

(4)、最后跳转到 arch/arm/lib/crt0.S 里面的 _main 进入 C 代码部分。

2、stage2 C 语言代码部分

(1)、执行 arch/arm/lib/board.c 里面的 board_init_f，清空 gd，设置 gd 里面变量 mon_len 为 uboot 大小，执行 init_sequence 里面的初始化序列函数，初始化定时器，串口，环境变量，设置 dram banks 等等。继续 gd 的初始化，如果定义了宏 CONFIG_SYS_MEM_TOP_HIDE 则预留 CONFIG_SYS_MEM_TOP_HIDE ram 隐藏空间，如果定义了 CONFIG_LOGBUFFER，则预留 LOGBUFF_RESERVE 空间给 kernel logbuffer，设置 TLB 大小和空间，如果定义了 CONFIG_LCD 则预留 fb，设置 TOTAL_MALLOC_LEN 大小为 malloc 空间，预留空间给 bd，预留空间给 gd，得到 addr_sp 为栈指针，addr 为 uboot 起始地址。设置波特率，设置 dram 地址地址和大小，设置 uboot 重定位地址为 addr，起始栈指针为 addr_sp，设置重定位偏移，把 gd 数据由当前 r9 处拷贝到新的 gd 空间。

(2)、执行 arch/arm/lib/board.c 里面的 board_init_r 继续初始化，设置 gd 标志为 GD_FLG_RELOC，使能 ceche，调用 board/s5p6818/drone/board.c 里面的 board_init 初始化平台类型，调用 serial_initialize 注册串口设备，调用 mem_malloc_init 初始化 malloc，初始化 nand flash，执行 env_relocate 设置环境变量，设置中断，初始化网卡，最后跳转到 main_loop 死循环，在 main_loop 中会调用 process_boot_delay 检测启动阶段有没有键按下，如果有就进入 uboot 命令行，如果没有就默认加载 kernel，如果加载失败也会进入 uboot 命令行。

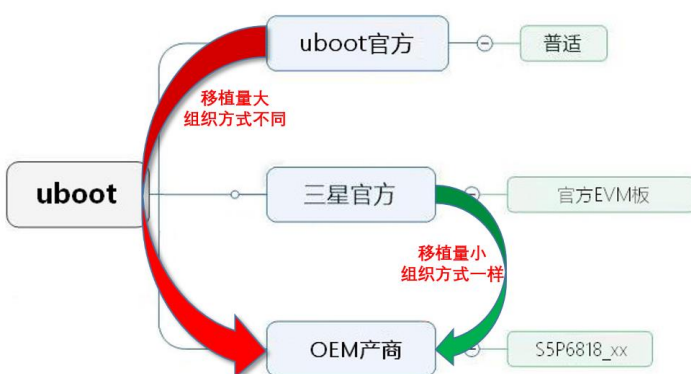
经过编译后，uboot 会生成一个 uboot.bin 镜像，将这个镜像烧到设备上的一个特定分区去，就可以作为 Bootloader 使用了。Bootloader 支持交互式启动，也就是我们可以让 Bootloader 初始化完成硬件之后，不是马上去启动 OS，而是停留在当前状态，等待用户输入命令告诉它接下来该干什么。

四、U-boot 移植

本节仅仅说明 U-boot 移植过程，由于 U-boot 各个版本有所不同，加之各个 OEM 厂家各自修改也不同，其中文件目录等可能与实际情况有所不同。

1、准备源码

移植 U-boot 有两种常规方式，一种是直接基于 U-BOOT 官方源码包进行移植，一种是基于芯片厂家的 EVM 板源码包移植。使用官方包移植量大，但是适用于绝大多数平台。使用芯片厂家的源码包移植量小，但可能与官方的组织结构不同。



我们可以在 U-boot 官方网站上下载 uboot 源码包, 官方网址为 <ftp://ftp.denx.de/pub/u-boot/>。也可以从其他 OEM 厂家下载源码包, OEM 厂家可能只开放给合约厂商, 如三星只开放资料给其合作厂商。

2、添加平台信息

进入 board 目录, 删除所有目录和文件。新建 s5p6818 目录, 进入 s5p6818 目录, 新建 common 目录, 在 common 目录中创建 cfg_type.h 文件, 详细内容见源码。

```
ours@ubuntu:~/ours6818/android_6818/u-boot/board/s5p6818/common$ ls
cfg_type.h
```

在 s5p6818 目录中创建 drone 目录, 在 drone 目录中分别创建 board.c、fastboot_lcd.c、config.mk、Makefile、display.c、eth.c 文件, 文件内容见源码。

```
ours@ubuntu:~/ours6818/android_6818/u-boot/board/s5p6818/drone$ ls
board.c  built-in.o  display.o  fastboot_lcd.c  include
board.o  config.mk  display.su  fastboot_lcd.o  Makefile
board.su  display.c  eth.c      fastboot_lcd.su
```

在 drone 目录下创建 include 目录, 并在其中创建 axp228_cfg.h、cfg_gpio.h、cfg_main.h、cfg_mem.h 文件

```
ours@ubuntu:~/ours6818/android_6818/u-boot/board/s5p6818/drone/include$ ls
axp228_cfg.h  cfg_gpio.h  cfg_main.h  cfg_mem.h
```

修改 board/s5p6818/drone/config.mk 中的交叉编译工具

```
ifneq ($(CONFIG_ARM64), y)
# CROSS_COMPILE ?= arm-eabi-
ifeq ($(origin CROSS_COMPILE), undefined)
CROSS_COMPILE := arm-eabi-
endif
ifeq ($(strip $(CROSS_COMPILE)),)
CROSS_COMPILE := arm-eabi-
endif
else
#CROSS_COMPILE := aarch64-linux-android-
ifeq ($(origin CROSS_COMPILE), undefined)
CROSS_COMPILE := aarch64-linux-android-
endif
ifeq ($(strip $(CROSS_COMPILE)),)
CROSS_COMPILE := aarch64-linux-android-
endif
endif
```

3、增加 config 文件

进入 include/configs 目录, 创建或者修改 s5p6818_drone.h 文件, 若需要编译 64 位版, 创建 s5p6818_arm64_drone.h, 具体参见源码。

4、指定目标 CPU

U-boot 源码中所使用 CPU 的代码位于 arch/arm/cpu/slsiap, 一般所支持的处理器都会放在 arch/目录。与 ARM 相关的在 arch/arm 目录。arch/arm/cpu/slsiap 存放所有与 S5P6818 相关的代码都在此。此目录是移植过程中一个重点需要修改的文件目录, 此目录一般是基于所使用芯片厂的代码修改。

5、配置平台 boards

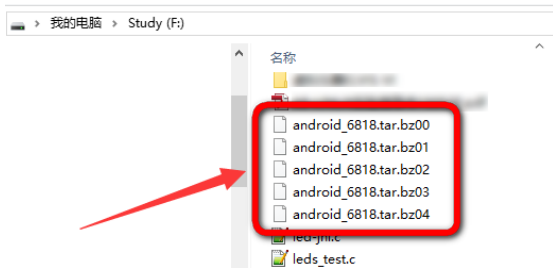
配置 boards.cfg: 打开 uboot 根目录下的 boards.cfg, 在文件相应位置新增 boards 配置

```
Liakhovetski <g.liakhovetski@gmx.de>
Active arm slsiap s5p4418 s5p4418 drone s5p4418_drone -
Active arm slsiap s5p4418 s5p4418 lepus s5p4418_lepus -
Active arm slsiap s5p4418 s5p4418 svt s5p4418_svt -
Active arm slsiap s5p4418 s5p4418 asb s5p4418_asb -
Active arm slsiap s5p4418 s5p4418 smartbox s5p4418_smartbox -
Active arm slsiap s5p6818 s5p6818 drone s5p6818_drone -
Active arm slsiap s5p6818 s5p6818 FDONE s5p6818_FDONE -
Active arm slsiap s5p6818 s5p6818 FDONE s5p6818_FDONE_burning -
Active arm slsiap s5p6818 s5p6818 svt s5p6818_svt -
Active arm slsiap s5p6818 s5p6818 asb s5p6818_asb -
Active aarch64 slsiap s5p6818 s5p6818 svt s5p6818_arm64_svt s5p6818_arm64_svt:ARM64
Active aarch64 slsiap s5p6818 s5p6818 asb s5p6818_arm64_asb s5p6818_arm64_asb:ARM64
Active aarch64 slsiap s5p6818 s5p6818 drone s5p6818_arm64_drone s5p6818_arm64_drone:ARM64
```

7.1.5 实验步骤

一、准备源码包

将 Android 源码包放到虚拟机 Windows 共享目录中（在 windows 系统中），可以使用虚拟机共享文件或者 samba 共享方式，笔者采用虚拟机共享示例：笔者的共享设置是将 windows 系统下的 F 盘共享给了 Ubuntu 系统，所以，将 Android 全部源码包（由于 Android 源码包较大，采用分卷压缩方式，共包含 android_6818.tar.bz00 、 android_6818.tar.bz01 、 android_6818.tar.bz02 、 android_6818.tar.bz03 、 android_6818.tar.bz04 几个文件）存放到 F:盘下，如下图：



二、解压源码包

①请将 Android 源码包从虚拟机共享目录拷贝至工作目录（注意：不要直接在 WIN 系统下复制，在 ubuntu 系统下粘贴；不要直接在共享目录解压。这两种做法都可能造成文件莫名其妙的权限混乱）

```
$ cp -arf /mnt/hgfs/share/android_6818.tar.bz0* /home/ours/ours6818
```

```
$ cd /home/ours/ours6818
```

```
$ ls
```

```
ours@ubuntu:~$ cp -arf /mnt/hgfs/share/android_6818.tar.bz0* /home/ours/ours6818
ours@ubuntu:~$ cd /home/ours/ours6818
ours@ubuntu:~/ours6818$ ls
android_6818.tar.bz00  android_6818.tar.bz03
android_6818.tar.bz01  android_6818.tar.bz04
android_6818.tar.bz02
```

②解压源码包

```
$ cat android_6818.tar.bz0* | tar -jxv
```

```
$ ls
```

```
ours@ubuntu:~/ours6818$ cat android_6818.tar.bz0* | tar -jxv
ours@ubuntu:~/ours6818$ ls
android_6818  android_6818.tar.bz02
android_6818.tar.bz00  android_6818.tar.bz03
android_6818.tar.bz01  android_6818.tar.bz04
```

解压完成后将会生成 android_6818 目录。U-boot、kernel、android 的源码都在此目录下。

```
ours@ubuntu:~/ours6818/android_6818$ ls
abi          dalvik       hardware     make.sh      pdk          u-boot
art          developers   jdk_openJdk17.sh  mnt          prebuilts   vendor
bionic       development  kernel       ndk          README
bootable     device       libcore      ndk.tar.bz2  result
build        docs         libnativehelper  out          sdk
copy_image.sh external     linux        out_image    system
cts          frameworks   Makefile      packages     tools
ours@ubuntu:~/ours6818/android_6818$
```

三、编译 u-boot

由之前的 Android 启动流程可知，引导程序不是 Android 操作系统的一部分，Android 使用 LINUX 内

核，所以我们可以使用所有能够引导 LINUX 内核的引导程序来引导 Android。比如最常用的 U-boot 便可作为引导程序，用来引导 LINUX 及 Android。

① 编译 u-boot

编译 u-boot、kernel、Android 等，需要设置交叉编译工具链，传统的做法是直接修改系统的环境变量，将交叉编译工具链添加到系统环境变量中，这种方式会对所有使用系统的用户产生影响。更为推荐的做法是，针对不同平台，在其源码中配置交叉编译工具，这样不会对系统造成影响。

在系统源码包的 device/nexell/tools 目录下存放有各种脚本，其中 build.sh 是编译系统的脚本，在其中设置了交叉编译工具以及编译规则，bootloader、kernel、android 等都使用此脚本编译，通过传递硬件平台参数和目标文件参数，可编译生成需要的目标文件。在 Android 源码顶层目录执行如下编译指令

```
$ ./device/nexell/tools/build.sh -b s5p6818_drone -t u-boot
```

或者

```
$ ./make.sh u-boot
```

为了减少输入量和出错率，我们创建了 make.sh 脚本，在 make.sh 中调用了 device/nexell/tools/build.sh 脚本，与手动输入效果一致。

```
ours@ubuntu:~/ours6818/android_6818$ ./device/nexell/tools/build.sh -b s5p6818_drone -t u-boot
===== 8 threads compile source code,can be changed in device/nexell/toosl/build.sh =====
PATH setting for android toolchain
Using built-in specs.
COLLECT_GCC=arm-eabi-gcc
COLLECT_LTO_WRAPPER=/home/ours/ours6818/android_6818/prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/bin/../libexec/gcc/arm-eabi/4.8/lto-wrapper
Target: arm-eabi
Creating filesystem with parameters:
  Size: 448790528
  Block size: 4096
  Blocks per group: 32768
  Inodes per group: 6848
  Inode size: 256
  Journal blocks: 1712
  Label:
  Blocks: 109568
  Block groups: 4
  Reserved block group size: 31
Created filesystem with 11/27392 inodes and 3534/109568 blocks
----- End of post processing
ours@ubuntu:~/ours6818/android_6818$
```

成功编译后，会在 u-boot 目录下生成 u-boot.bin 文件，此文件便是目标板的 bootloader 程序。同时脚本会将生成的目标文件拷贝至 result 目录。

```
ours@ubuntu:~/ours6818/android_6818$ cd result/
ours@ubuntu:~/ours6818/android_6818/result$ ls
2ndboot.bin  boot.img  partmap.txt  system      userdata
arm_arch     cache     root         system.img  userdata.img
boot         cache.img root.img.gz  u-boot.bin
ours@ubuntu:~/ours6818/android_6818/result$
```

至此 u-boot 编译完成。

7.2 编译 Kernel 实验

7.2.1 实验目的

掌握 Android kernel 编译方法。

7.2.2 实验内容

编译 Android 的 kernel

7.2.3 实验设备

硬件：配置完开发环境 ubuntu 虚拟机的 PC 机 1 台

软件：Android kernel 源码包

7.2.4 实验原理

一、Android 内核和 Linux 内核的区别

Android 系统层面的底层是 Linux，并且在中间加上了一个叫做 Dalvik 的 Java 虚拟机，从表面层看是 Android 运行库。每个 Android 应用都运行在自己的进程上，享有 Dalvik 虚拟机为它分配的专有实例。为了支持多个虚拟机在同一设备上高效运行，dalvik 被改写过。Dalvik 虚拟机执行的是 Dalvik 格式的可执行文件（.dex）-该格式经过优化，以将内存占用降到最低。

Android 内核和 Linux 内核的差别主要体现在如下 11 个方面：

1.Android Binder

Android Binder 是基于 Openbinder 框架的一个驱动，用于提供 Android 平台的进程间的通信(IPC)。原来的 Linux 系统上层应用的进程间通信主要是 D-bus，采用消息总线的方式来进行 IPC。其源代码位于 drivers/staging/android/binder.c

2.Android 电源管理（PM）

Android 电源管理是一个基于标准 Linux 电源管理系统的轻量级 Android 电源管理驱动，针对嵌入式设备做了很多优化。利用锁和定时器来切换系统状态，控制设备在不同状态下的功耗，以达到节能的目的。其源码位于 kernel/power/earlysuspend.c、kernel/power/consoleearlysuspend.c、kernel/power/fbearlysuspend.c、kernel/power/wakelock.c、kernel/power/userwakelock.c

3. 低内存管理器（Low memory Killer）

Android 中低内存管理器和 linux 标准的 OOM 相比，其机制更加灵活，可以根据需要杀死进程来释放需要的内存。Low memory Killer 的代码非常简单，里面关键是函数 Lowmem_shrinker()。作为一个模块在初始化时调用 register_shrinker 注册一个 Lowmem_shrinker，它会被 vm 在内存紧张的情况下调用。源码位于 drivers/staging/android/lowmemorykiller.c

4.匿名共享内存(Ashmem)

匿名共享内存为进程间提供大块共享内存，同时为内核提供回收和管理这个内存的机制。如果一个程序尝试访问 Kernel 释放的一个共享内存块，它将会受到一个错误提示，然后重新分配内存并重载数据。其源码位于 mm/ashmem.c

5. Android PMEM(Physical)

PMEM 用于向用户空间提供连续的物理内存区域，DSP 和某些设备只能工作在连续的物理内存上。驱动中提供了 mmap、open/release 和 ioctl 等接口。

6.Android Logger

Android Logger 是一个轻量级的日志设备，用于抓取 Android 系统的各种日志，是 Linux 没有的

7.Android Alarm

Android Alarm 提供了一个定时器用于把设备从睡眠状态唤醒，同时它也提供了一个即使在设备睡眠是也会运行的时钟基准。其源码位于 `driver/rtc/alarm.c` `drivers/rtc/alarm-dev.c`

8.USB Gadget 驱动

此驱动是一个具有标准 Linux USB gadget 驱动框架的设备驱动，Android 的 USB 驱动是基于 gadget 框架的。其源码位于如下文件：`drivers/usb/gadget/android.c`、`drivers/usb/gadget/f_adb.c`、`drivers/usb/gadget/f_mass_storage.c`

9.Android Ram Console

为了提供调试功能，Android 允许将调试日志信息写入一个被称为 RAM Console 的设备里，它是一个基于 RAM 的 Buffer 其源码位于 `drviers/staging/android/ram_console.c`

10.Android timed device

Android timed device 提供了对设备进行定时控制功能，目前仅仅支持 vibrator 和 LED 设备。其源码为 `drviers/staging/adnroid/timed_output.c`

11.Yaffs2 文件系统

在 Android 系统中，采用 Yaffs2 作为 MTD NAND FLASH 文件系统。Yaffs2 是一个快速稳定的应用于 NAND 和 NOR Flash 的跨平台的嵌入式设备文件性，同其他 Flash 文件系统相比，Yaffs2 使用更小的内存来保存运行状态，因此它占用内存小；Yaffs2 的垃圾回收非常简单而且快速，因此能够达到更好的性能；其源代码位于 `fs/yaffs2` 目录

二、Android 5.1 源码结构

1、顶层目录

- |-- Makefile （make 文件）
- |-- abi （applicationbinary interface，应用程序二进制接口，生成 libgabi++.so 相关库文件）
- |-- art （google 在 4.4 后加入用来代替 Dalvik 的运行时）
- |-- bionic (Android 的 C library，即 C 库文件)
- |-- bootable （启动引导相关代码）
- |-- build （存放系统编译规则及基础开发配置包）
- |-- cts （Android 兼容性测试套件标准）
- |-- dalvik （dalvik JAVA 虚拟机）
- |-- developers (开发者用，存放几个例子)
- |-- development （开发者需要的一些例程及工具）
- |-- device (设备相关代码，这是各厂商需要配置和修改的代码)
- |-- docs (介绍开源相关文档)
- |-- external （android 使用的一些开源的模组）
- |-- frameworks （核心框架—java 及 C++语言）
- |-- hardware （部分厂家开源的硬件适配层 HAL 代码）
- |-- kernel (驱动内核相关代码)
- |-- libcore (核心库相关)
- |-- libnativehelper (JNI 用到的库)
- |-- ndk (ndk 相关)
- |-- out （编译完成后的代码输出目录）
- |-- packages （应用程序包）
- |-- pdk (google 用来减少碎片化的东西)
- |-- prebuilt （x86 和 arm 架构下预编译的一些资源）
- |-- sdk （sdk 及模拟器）
- |-- tools （工具）

- |-- system （底层文件系统库、应用及组件——C 语言）
- |-- vendor （厂商定制代码，历史遗留问题，目前都放在 device）

2、bionic 目录

- |-- libc （C 库）
 - ||-- arch-arm （ARM 架构，包含系统调用汇编实现）
 - ||-- arch-x86 （x86 架构，包含系统调用汇编实现）
 - ||-- bionic （由 C 实现的功能，架构无关）
 - ||-- docs （文档）
 - ||-- include （头文件）
 - ||-- kernel （Linux 内核中的一些头文件）
 - ||-- private （一些私有的头文件）
 - ||-- stdio （stdio 实现）
 - ||-- tools （几个工具）
 - ||-- tzcode （时区相关代码）
 - ||-- zoneinfo （时区信息）
- |-- libdl （libdl 实现，dl 是动态链接，提供访问动态链接库的功能）
- |-- libm （libm 数学库的实现，）
 - ||-- amd64 （amd64 架构）
 - ||-- arm （arm 架构）
 - ||-- i387 （i387 架构）
 - ||-- include （头文件）
- |-- libstdc++ （libstdc++ C++实现库）
 - ||-- include （头文件）
- |-- linker （动态链接器）
- |-- tests

3、build 目录

- |-- core （核心编译规则）
- |-- libs
 - ||-- host （主机端库，有 android “cp” 功能替换）
- |-- target （目标机编译对象）
 - ||-- board （开发平台）
 - |||-- generic （通用）
 - ||-- product （开发平台对应的编译规则）
 - ||-- security （密钥相关）
- |-- tools （编译中主机使用的工具及脚本）
 - ||-- acp （Android "acp" Command）
 - ||-- apicheck （api 检查工具）
 - ||-- atree （tree 工具）
 - ||-- check_prereq （检查编译时间戳工具）
 - ||-- droiddoc （java 语言，和 JDK5 有关）
 - ||-- fs_config （This program takes a list of files and directories）
 - ||-- fs_get_stats （获取文件系统状态）

- ||-- releasetools （生成镜像的工具及脚本）
- ||-- rgb2565 （rgb 转换为 565）
- ||-- signapk （apk 签名工具）
- ||-- zipalign (zip archive alignment tool)

4、dalvik 目录 dalvik 虚拟机

- |-- dexdump （dex 反汇编）
- |-- dexgen
- |-- dexlist （List all methods in all concrete classes in a DEX file.）
- |-- docs （文档）
- |-- dx （dx 工具，将多个 java 转换为 dex）
- |-- hit （java 语言写成）
- |-- opcode-gen
- |-- tools （工具）
- |-- vm （虚拟机实现）

5、development 目录 （开发者需要的一些例程及工具）

- |-- apps （一些核心应用程序）
 - ||-- BluetoothDebug （蓝牙调试程序）
 - ||-- BuildWidget （版本号相关）
 - ||-- CustomLocale （自定义区域设置）
 - ||-- Development （开发）
 - ||-- DevelopmentSettings (开发设置)
 - ||-- Fallback (回调)
 - ||-- GestureBuilder （手势动作）
 - ||-- launchperf
 - ||-- NinePatchLab （点九图片相关）
 - ||-- OBJViewer （OBJ 查看器）
 - ||-- SdkSetup （SDK 安装器）
 - ||-- SettingInjectorSample （高级设置）
 - ||-- WidgetPreview （预览小部件）
- |-- build （编译脚本模板）
- |-- cmds （有个 monkey 工具）
- |-- docs （文档）
- |-- host （主机端 USB 驱动等）
- |-- ide （集成开发环境）
- |-- libraries (库)
- |-- ndk （本地开发套件——c 语言开发套件）
- |-- perftests
- |-- samples （例程）
 - ||-- AliasActivity
 - ||-- ApiDemos （API 演示程序）
 - ||-- BluetoothChat （蓝牙聊天）
 - ||-- BrowserPlugin （浏览器插件）

```

||-- BusinessCard （商业卡）
||-- Compass （指南针）
||-- ContactManager （联系人管理器）
||-- CubeLiveWallpaper （动态壁纸的一个简单例程）
||-- FixedGridLayout （固定网格布局）
||-- GlobalTime （全球时间）
||-- HelloActivity （Hello）
||-- Home （Home）
||-- JetBoy （jetBoy 游戏）
||-- LunarLander （貌似又是一个游戏）
||-- MailSync （邮件同步）
||-- MultiResolution （多分辨率）
||-- MySampleRss （RSS）
||-- NotePad （记事本）
||-- RSSReader （RSS 阅读器）
||-- SearchableDictionary （目录搜索）
||-- SimpleJNI （JNI 例程）
||-- SkeletonApp （空壳 APP）
||-- Snake （snake 程序）
||-- SoftKeyboard （软键盘）
||-- Wiktionary （维基）
||-- WiktionarySimple （维基例程）
|-- scripts （脚本）
|-- sdk （sdk 配置）
|-- sdk_overlay
|-- sys-img
|-- testrunner （测试用）
|-- tools （一些工具）
|-- tutorials

```

6、external 目录

```

|-- aes （AES 加密）
|-- apache-http （网页服务器）
|-- astl （ASTL (Android STL) is a slimmed-down version of the regular C++ STL.）
|-- bison （自动生成语法分析器，将无关文法转换成 C、C++）
|-- blktrace （blktrace is a block layer IO tracing mechanism）
|-- bluetooth （蓝牙相关、协议栈）
|-- bsdiff （diff 工具）
|-- bzip2 （压缩工具）
|-- clearsilver （html 模板系统）
|-- dbus （低延时、低开销、高可用性的 IPC 机制）
|-- dhcpcd （DHCP 服务）
|-- dosfstools （DOS 文件系统工具）
|-- dropbear （SSH2 的 server）

```

```
|-- e2fsprogs (EXT2 文件系统工具)
|-- elfcopy (复制 ELF 的工具)
|-- elfutils (ELF 工具)
|-- embunit (Embedded Unit Project)
|-- emma (java 代码覆盖率统计工具)
|-- esd (Enlightened Sound Daemon, 将多种音频流混合在一个设备上播放)
|-- expat (Expat is a stream-oriented XML parser.)
|-- fdlibm (FDLIBM (Freely Distributable LIBM))
|-- freetype (字体)
|-- fsck_msdos (dos 文件系统检查工具)
|-- gdata (google 的无线数据相关)
|-- genext2fs (genext2fs generates an ext2 filesystem as a normal (non-root) user)
|-- giflib (gif 库)
|-- googleclient (google 用户库)
|-- grub (This is GNU GRUB, the GRand Unified Bootloader.)
|-- gtest (Google C++ Testing Framework)
|-- icu4c (ICU(International Component for Unicode)在 C/C++下的版本)
|-- ipsec-tools (This package provides a way to use the native IPsec functionality )
|-- iptables (防火墙)
|-- jdiff (generate a report describing the difference between two public Java APIs.)
|-- jhead (jpeg 头部信息工具)
|-- jpeg (jpeg 库)
|-- junit (JUnit 是一个 Java 语言的单元测试框架)
|-- kernel-headers (内核的一些头文件)
|-- libffi (libffi is a foreign function interface library.)
|-- libpcap (网络数据包捕获函数)
|-- libpng (png 库)
|-- libxml2 (xml 解析库)
|-- mtpd (一个命令)
|-- netcat (simple Unix utility which reads and writes data across network connections)
|-- netperf (网络性能测量工具)
|-- neven (看代码和 JNI 相关)
|-- opencore (多媒体框架)
|-- openssl (SSL 加密相关)
|-- openvpn (VPN 开源库)
|-- oprofile (OProfile 是 Linux 内核支持的一种性能分析机制。)
|-- ping (ping 命令)
|-- ppp (pppd 拨号命令, 好像还没有 chat)
|-- proguard (Java class file shrinker, optimizer, obfuscator, and preverifier)
|-- protobuf (a flexible, efficient, automated mechanism for serializing structured data)
|-- qemu (arm 模拟器)
|-- safe-iop (functions for performing safe integer operations )
|-- skia (skia 图形引擎)
|-- sonivox (sole MIDI solution for Google Android Mobile Phone Platform)
```

- |-- speex （Speex 编/解码 API 的使用(libspeex)）
- |-- sqlite （数据库）
- |-- srec （Nuance 公司提供的开源连续非特定人语音识别）
- |-- strace （trace 工具）
- |-- svox （Embedded Text-to-Speech）
- |-- tagsoup （TagSoup 是一个 Java 开发符合 SAX 的 HTML 解析器）
- |-- tcpdump （抓 TCP 包的软件）
- |-- tesseract （Tesseract Open Source OCR Engine.）
- |-- tinyxml （TinyXml is a simple, small, C++ XML parser）
- |-- tremor （I stream and file decoder provides an embeddable,integer-only library）
- |-- webkit （浏览器核心）
- |-- wpa_supplicant （无线网卡管理）
- |-- xmlwriter （XML 编辑工具）
- |-- yaffs2 （yaffs 文件系统）
- |-- zlib （a general purpose data compression library）

7、frameworks 目录（核心框架——java 及 C++语言）

- |-- av （音视频相关）
 - ||-- camera （摄像头服务程序库）
- |-- base （基本内容）
 - ||-- api （都是 xml 文件，定义了 java 的 api）
 - ||-- cmds （重要命令： am、app_proce 等）
 - ||-- core （核心库）
 - ||-- data （字体和声音等数据文件）
 - ||-- docs （文档）
 - ||-- graphics （图形相关）
 - ||-- include （头文件）
 - ||-- keystore （和数据签名证书相关）
 - ||-- libs （库）
 - ||-- location （地区库）
 - ||-- media （媒体相关库）
 - ||-- obex （蓝牙传输库）
 - ||-- opengl （2D-3D 加速库）
 - ||-- packages （设置、TTS、VPN 程序）
 - ||-- sax （XML 解析器）
 - ||-- services （各种服务程序）
 - ||-- telephony （电话通讯管理）
 - ||-- test-runner （测试工具相关）
 - ||-- tests （各种测试）
 - ||-- tools （一些叫不上名的工具）
 - ||-- wifi （无线网络）

8、hardware 目录（部分厂家开源的硬解适配层 HAL 代码）

- |-- broadcom （博通公司）


```

    |-- wlan （无线网卡）
|-- libhardware （硬件库）
    |-- include （头文件）
    |-- modules （Default (and possibly architecture dependents) HAL modules）
|-- libhardware_legacy （旧的硬件库）
    |-- include （头文件）
    |-- power （电源）
    |-- qemu （模拟器）
    |-- qemu_tracing （模拟器跟踪）
    |-- uevent （uevent）
    |-- vibrator （震动）
    |-- wifi （无线）
|-- realtek （realtek 公司硬件）
    |-- bt （realtek 蓝牙）
    |-- wlan （realtek WIFI 设备）
|-- ril （无线电抽象层）
    |-- include （头文件）
    |-- libril （库）
    |-- reference-cdma-sms （cdma 短信参考）
    |-- reference-ril （ril 参考）
    |-- rild （ril 后台服务程序）
|-- samsung_slsi （三星 slsi HAL）
    |-- drone 平台 HAL
    |-- exynos5 平台 HAL
    |-- slsiap 平台 HAL
|-- ti （ti 公司开源 HAL）
    |-- omap3 （omap3 处理器）
        |||-- dspbridge （DSP 桥）
    |||-- libstagefrighthw （stagefright 硬件库）
    |||-- omx （omx 组件）

```

9、packages 目录

```

|-- apps （应用程序库）
    |-- BasicSmsReceiver （简单消息接收器）
    |-- Bluetooth （蓝牙）
    |-- Browser （浏览器，5.1 以后默认是没有的，得找高通申请才会有源码，且源码很大，得自己编译）
    |-- Calculator （计算器）
    |-- Calendar （日历）
    |-- Camera （相机）
    |-- Camera2 (相机，4.4 以后都用这个)
    |-- CellBroadcastReceiver （小区广播）
    |-- CertInstaller （证书安装程序）
    |-- Contacts （联系人）
    |-- ContactsCommon (联系人)

```

```
||-- DeskClock （时钟）
||-- Dialer (拨号)
||-- Email （邮箱）
||-- Exchange （邮箱里的 exchange 账号）
||-- FMRadio (调频广播)
||-- Gallery （相册，和 Camera 类似，多了列表）
||-- Gallery2 （相册，4.4 以后都用这个）
||-- HdmiDualVideo（HDMI 双屏显示）
||-- HTMLViewer （浏览器附属界面，被浏览器应用调用，同时提供存储记录功能）
||-- InCallUI (来电界面)
||-- KeyChain (密钥链)
||-- Launcher2 （登陆启动项，负责应用的调用）
||-- LegacyCamera (相机，一般不用)
||-- ManagedProvisioning (设备配置器)
||-- Mms （短彩信业务）
||-- Music （音乐播放器）
||-- MusicFX (音效)
||-- Nfc (NFC 服务)
||-- OneTimeInitializer （一键清理）
||-- PackageInstaller （安装、卸载程序的响应）
||-- Phone （电话）
||-- Protips (主菜单提示)
||-- Provision （预设应用的状态，使能应用）
||-- QuickSearchBox （快速查找）
||-- Settings （设置：开机设定，包括电量、蓝牙、设备信息、界面、wifi 等）
||-- SmartCardService
||-- SoundRecorder （录音机）
||-- SpareParts
||-- SpeechRecorder
||-- Stk （SIM 卡应用）
||-- Tag
||-- Terminal
||-- TvSettings (Tv 设置)
||-- UnifiedEmail (邮箱相关)
||-- VoiceDialer （语音识别通话）
|-- experimental
|-- inputmethods （输入法）
    |-- LatinIME （拉丁文输入法）
    |-- OpenWnn （OpenWnn 日语输入法）
|-- providers （提供器，提供应用程序、界面所需的数据）
    |-- ApplicationsProvider （应用程序提供器，提供应用程序启动项、更新等）
    |-- CalendarProvider （日历提供器）
    |-- ContactsProvider （联系人提供器）
    |-- DownloadProvider （下载管理提供器）
```

```
    |-- MediaPlayer （媒体提供器，提供存储数据）
    |-- PartnerBookmarksProvider (书签提供器)
    |-- TelephonyProvider （彩信提供器）
    |-- UserDictionaryProvider （用户字典提供器，提供用户常用字字典）
    |-- TvProvider
|-- screensavers (屏保)
    |-- Basic (基本互动屏保)
    |-- PhotoTable （照片屏保）
    |-- WebView （）
|-- services
    |-- Mms (短彩信服务)
    |-- Telecomm （通话服务）
    |-- Telephony （通话服务）
|-- wallpapers （墙纸）
    |-- Basic （基本墙纸，系统内置墙纸）
    |-- Galaxy4
    |-- HoloSpiral
    |-- LivePicker （选择动态壁纸）
    |-- MagicSmoke （壁纸特殊效果）
    |-- MusicVisualization （音乐可视化，图形随音乐而变化）
    |-- NoiseField
    |-- PhaseBeam
```

10、prebuilt 目录（x86 和 arm 架构下预编译的一些资源）

```
-- android-emulator （android 模拟器）
    |-- darwin-x86_64 （darwin x86 64bit 平台）
    |-- linux-x86_64 （linux x86 64bit 平台）
    |-- windows （windows 平台）
```

11、system 目录（底层文件系统库、应用及组件——C 语言）

```
-- core （系统核心工具箱接口）
    |-- adb （adb 调试工具）
    |-- cpio （cpio 工具，创建 img）
    |-- debuggerd （调试工具）
    |-- fastboot （快速启动相关）
    |-- include （系统接口头文件）
    |-- init （init 程序源代码）
    |-- libcutils （libc 工具）
    |-- liblog （log 库）
    |-- libmincrypt （加密库）
    |-- libnetutils （网络工具库）
    |-- libpixelflinger （图形处理库）
    |-- libsysutils （系统工具库）
    |-- libzipfile （zip 库）
```

```

|-- logcat （查看 log 工具）
|-- logwrapper （log 封装工具）
|-- mkbootimg （制作启动 boot.img 的工具盒脚本）
|-- netcfg （网络配置 netcfg 源码）
|-- rootdir （rootfs，包含一些 etc 下的脚本和配置）
|-- toolbox （toolbox，类似 busybox 的工具集）
|-- extras （额外工具）
    |-- latencytop （a tool for software developers ， identifying system latency happen）
    |-- libpagemap （pagemap 库）
    |-- librank （Java Library Ranking System 库）
    |-- procmem （pagemap 相关）
    |-- procrank （Java Library Ranking System 相关）
    |-- showmap （showmap 工具）
    |-- showslab （showslab 工具）
    |-- sound （声音相关）
    |-- su （su 命令源码）
    |-- tests （一些测试工具）
    |-- timeinfo （时区相关）

```

12、vendor 目录（厂家定制内容）

```

--nexell （nexell 公司定制内容）
    |-- apps （nexell 公司定制应用）
    |-- security （加密安全相关）

```

三、Android 目录及文件

- **bootable:** 这个目录下存放 android 部分启动相关代码，包括 android 的 recovery 模式，一般用于进行 OTA 升级，由 C++编写，可以看到用于显示的 ui.cpp 和安装的 install.cpp,模式入口为 recovery.cpp 的 main。
- **build:** 这是 android 源码中编译核心所在地，把这个目录下的所有 make 搞清楚，android 的编译体系就基本了如指掌了。
- **build/envsetup.sh:** 编译初始化 shell 脚本，编译配置命令 lunch mm-mmm 等发源地，所以 android 在编译的初始阶段需要 source *，其最终目的都会执行到这个脚本，把这个脚本中的变量以及函数设置到当前终端的临时变量中，供后续使用。由此脚本中的 lunch 选取 product_name 引入到 core 中的 make 等一系列的初始配置，最后会打印出 TARGET 变量等.供源码中编译使用！
- **build/core/main.mk:** make -j*时的 makefile 入口文件，会对编译体系中的变量进行一些校对，编译类型之类的，并且加载整个源码下的 Android.mk 文件，整体的编译框架，终极目标.PHONY:droid
- **build/core/Makefile:** 由上面的 main.mk 引入，算作 android 真正的主 makefile，由它再依赖到各个子编译体系。
- **build/core/base_rules.mk:** android 整体编译时，会加载根目录下所有的 Android.mk 文件，并且根据文件中的 MODULE 依次分析相关属性，生成编译规则，其中不同的 MODULE 类型就需要在 Android.mk 中 include\$(**)加载对应 mk，分别对应 core 目录下的 mk。比如编译 apk 的 Android.mk 需要末尾 include\$(BUILD_PACKAGE),此时解析到这里的时候就会加载 core 下的 package.mk，其中会加载进 java.mk 在这里会加载到 base_rules.mk 中计算一些变量值，创建一些基本的依赖规则，再又 java.mk 中调用函数\$(transform-**)编译，类似\$(transform-java-to-classes.jar)把 java 文件编译成 classes.jar 其它模块

类型类似。

- **build/core/definitions.mk**: 这个文件下都是一些编译的函数宏定义, 比如上面调用 `transform-java-to-classes.jar` 以及常常出现在 `Android.mk` 中的 `all-java-files-under` 等等...都可在这里找到具体实现!
- **build/target/product/security**: 这个目录下面存放的就是当前编译系统使用的签名密钥对, 用于系统不同组件在编译的时候进行数字签名, `android` 原生默认使用 `testkey`, 这目录下有 `README` 以及密钥对制作脚本 `make_key`, 可以用来制作属于自己的签名密钥, 使整个系统签名独一无二, 更具安全性!
- **build/tools/releasetools**: `Build` 的 `tools` 目录, 全是一些用于编译的工具脚本和可执行工具, 其中 `releasetools` 下是用于编译制作 `androidOTA` 刷机包时的 `Python` 脚本集合, 由上面说到的主 `Makefile` 中调用进 `ota_from_target_files` 中的 `defmain(argv)`:
- **cts**: `google` 提供的 `CompatibilityTest Suite (CTS)` 兼容性测试组, 用于测试 `android` 系统的兼容性以及稳定性, 发测试 report 给 `google` 过了这个认证, 算是得到 `google` 的认可. 一般的 `android` 源码都是有这个组件源码的, 但是不在主编译流程中, 需要使用 `makects` 编译出 `android-cts` 目录供使用。
- **cts/tests/tests**: 存放 CTS 测试用例的地方, 全是 `androidapk`, 添加自己的测试用例也在此。
- **cts/CtsTestCaseList.mk**: 这是 `cts` 模块组件的编译选项配置 `make`, 由上面说到的 `build` 中的 `cts.mk` 调用, 对于自己添加的测试用例需要添加进这里的 `cts_test_packages` 变量中。
- **cts/tools/tradefed-host/src/com/android/cts/tradefed/build/CtsBuildProvider.java** : 可以这里看 `CTS_BUILD_VERSION` 确定你当前源码中的 `cts` 版本。
- **cts/tools/tradefed-host/README**: `google` 提供的 `readme`, 有介绍如何配置 `cts` 环境以及使用的常用命令
- **device**: 这个作为 `android` 源码中对产品的描述文件夹, 各个平台的差异还是比较大的, 但是怎么改动, 本意是不变的, 只是作为要编译的产品的配置文件夹, 这里简单以 `nexell` 为例。
- **device/nexell/s5p6818_drone/AndroidProducts.mk**: 一般的存放规则是 `/device/厂商目录/产品目录`, 这个 `make` 里面一般是定义当前产品的主配置 `make`, 对于这个 `AndroidProducts.mk` 什么时候被加载, 具体可去看 `android` 编译初始化阶段, `lunch` 选取产品之后的一系列 `make` 初始化操作。
- **device/nexell/s5p6818_drone/BoardConfig.mk**: 这个配置文件, 看名字就知道了, 定义的都是跟硬件配置相关的. 这个 `make` 依赖级别在产品角度算是最高的了, 如果想添加一些控制宏, 可以考虑加在这里。
- **device/nexell/s5p6818_drone/device.mk**: 这里配置最多的就是产品编译需要的组件了, 一般配置最多的 `PRODUCT_COPY_FILES` 以及 `PRODUCT_PACKAGES`, 这两个变量在编译体系中的作用不多做介绍。
- **device/nexell/s5p6818_drone/recovery.fstab**: 熟悉 `linux` 的对这个 `fstab` 应该比较熟悉了, 这里配置的就是 `recovery` 模式下的分区, 会用于制作 `OTA` 刷机包时对分区的配置参数。
- **vendorsetup.sh**: 一般会将产品的编译信息存在这个文件中, 类似 `add_lunch_combo aosp_s5p6818_drone-userdebug`。在编译初始阶段由 `lunch` 加载供编译者选择, 这其中 `aosp` 代表编译, `s5p6818_drone` 代表产品名, `userdebug` 代表编译类型, `android` 的产品编译类型可另行参考。
- **external**: 这是 `android` 存放外部工具组件的地方, 以文件夹为单一模块, 最终编译出来的有可执行文件, `jar` 包, 动静态库, 东西比较混杂, `google` 已经移植了很多工具在这里面, 如果自己移植一些模块进 `android` 系统, 可以加在这里, 写好 `Android.mk`, 在上面提到的 `device.mk` 中加入 `PRODUCT_PACKAGES` 变量中。
- **frameworks**: `android` 的运行框架集合, 包含系统运行的各种服务框架, 向 `app` 层提供 `api`, 根据 `JNI` 机制或者 `socket` 往下层调用, 也可使用 `hw_get_module` 调用到 `hardware` 层的 `module`。
- **frameworks/base/core**: 字面意思, 核心所在, 包含 `java` 以及 `jni`, 核心组件的 `java` 类以及 `native` 方法的 `jni` 映射, 其中内容太多, 比如 `java` 中 `app` 相关的 `ActivityManager.java`, 启动相关的 `ZygoteInit.java`, 其中的 `jni` 目录会被编译成 `libandroid_runtime.so` 作为 `android` 运行时的动态库供相关的 `java` 加载。
- **frameworks/base/services**: 框架层中的系统服务存放目录, 包含系统时间服务以及输入子系统服务, 同上 `java` 目录下就是服务的 `java` 类了, 可以看到各种子服务模块, 比如 `pm`, `net`, `display`, 如果想具体了解当前系统启动了多少服务, 可以参考 `SystemService.java`。

- **frameworks/opt/telephony**: android 电话子系统存放目录, 向上提供 api 接口, 向下通过 RIL.javascript 通信。
- **hardware**: 硬件抽象层, 描述对 linuxkernel 中的相关驱动模块的具体操作, 而在 kernel 中的驱动模块只拥有通用操作接口, 比如设置寄存器值, IO 拉高拉低, 但是具体设置什么值, 拉高拉低的时序都写在 hardware 层相对应的 module 中, 这就是 google 对于硬件驱动的商业保护。
- **hardware/libhardware/hardware.c**: hardware 机制核心所在, 定义了相关规则, 比如 load 打开 modules 编译生成的.so, 抽象成一个 module, 向上层提供 hw_get_module 接口以及 module 配置宏。
- **hardware/libhardware/modules**: 这里就是与 kernel 相对应的 module 存放的地方, 头文件存在同级目录的 include 中, 在其中定义 module 结构, 接口方法以及唯一的 moduleID。其中像 gralloc 就是控制 kernel 中显示屏驱动的 module, 用于管理控制 fb 缓冲区, 将来自上层 display 的 SurfaceFlinger 服务传下来的图像推送到 lcd/led 显示屏上.其它类似。
- **hardware/ril**: android 电话系统的 ril 驱动文件目录, 其中包含:
 - rild— ril 主体控制机制
 - libril— ril 与上层 socket 通信
 - reference-ril— ril 与 serial 设备 AT 指令通信
 这三个文件夹, 其中 reference-ril 是第三方驱动, 根据不同的设备选择不同。
- **system**: android 系统底层的文件系统, 应用组件, 包含一些系统库, 以及启动的配置文件。
- **system/core/init/init.c**: 作为系统启动到 android 层的第一个进程, 也将一直作为守护进程, 解析 init.rc 配置文件, 启动相关服务, 其中就有比较常用的属性服务, 之后一直运行于 init 进程中, 具体可参考 property_service.c, android 层系统启动从这里开始。
- **system/core/rootdir**: 存放配置文件, 其中 init.rc 作为启动配置, ueventd.rc 作为 linux 文件系统中文件事件配置, 还包含磁盘挂载所需要的 vold.fstab 配置文件等。
- **system/core/include/private/android_filesystem_config.h**: 这个头文件定义了, android 文件系统中文件的权限配置。
- **out**: 作为 android 源码编译结果存放目录, 其中包含各种中间文件以及目标文件.像 obj 中存放的中间件以及 hostlinux-x86 存放的本地编译项。
- **out/target/product/s5p6818_drone/system.img**: android 系统编译出来的镜像文件, 也是整个源码的最终目标文件。
- **out/target/product/s5p6818_drone/system**: 编译之后的系统文件夹, 也是 system.img 的主要构成, 其中 app 目录下都是 apk 文件, android 中规定此目录下的 apk 作为系统内置应用, 在文件系统中拥有系统权限, 普通用户没有权限删除更改, 详情可参考 PackageManager.其中的 bin 代表可执行文件, etc 下存放的都是系统配置文件, lib 中都是些动态库, 分别对应到文件系统中。
- **out/target/product/s5p6818_drone/system/build.prop**: 这个文件中收集了编译中的所有属性, 包括编译的主机环境, 编译目标的各种配置信息等等...生成过程可参考主 Makefile,在初始化阶段会被 property_service 服务加载, 作为系统属性。
- **out/target/product/s5p6818_drone/data/**: 此目录作为 user 的 data 存储目录, 对应文件系统中的/data 目录, 平时用户安装的 apk 就会被 copy 到这个该目录的 app 目录下, android 系统中 apk 所产生的数据, 比如数据库等都会存放在/data/data 中, 以包名区分。

7.2.5 实验步骤

一、配置 Android 内核

在先前的说明中已经了解到 Android 的底层采用的是 LINUX 的内核, 其内核的配置和编译与 Linux 一样, 没有什么区别。

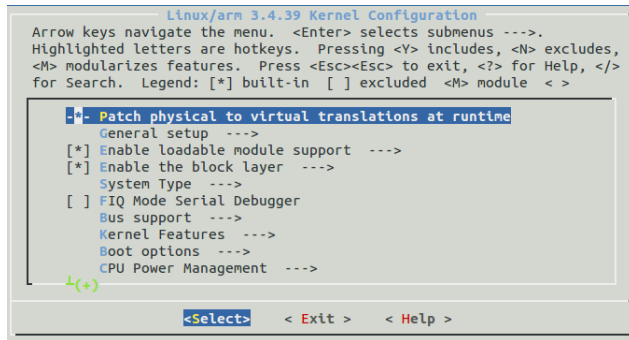
进入 Android 内核源码顶层目录 (/home/ours/ours6818/android_6818/kernel), 使用内核配置工具配置内

核，与 Linux 配置方法一致。

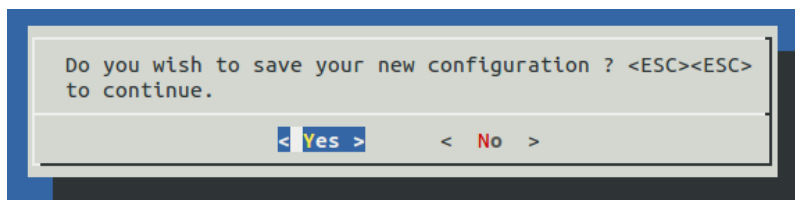
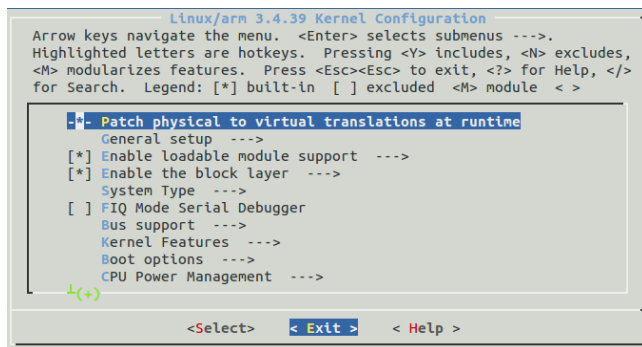
```
$ cd /home/ours/ours6818/android_6818/kernel
```

```
$ make ARCH=arm menuconfig
```

```
ours@ubuntu:~$ cd /home/ours/ours6818/android_6818/kernel
ours@ubuntu:~/ours6818/android_6818/kernel$ make ARCH=arm menuconfig
```



当配置完成后，使用键盘【Tab】键将焦点移到【Exit】，回车，如果更改过任何配置项，将弹出保存对话框（否则直接退回到终端）



焦点移到<Yes>,回车，将保存配置到.config 文件，之后便可编译内核。

说明：以上步骤可进入内核配置，与 LINUX 内核配置类似。源码中已经完成内核配置，用户自行更改配置可能导致无法正产编译，或者生成的镜像系统无法正常启动；源码中 /arch/arm/configs/s5p6818_drone_android_lollipop_defconfig 文件是目前使用的内核的配置文件。

二、编译 Android 内核

上一小节已说明，Android 系统使用同一个编译脚本编译所需的引导文件、内核文件、Android 系统。进入 Android 源码目录，执行内核编译命令（与 Linux 一样，只是脚本和命令不同）

```
$ cd /home/ours/ours6818/android_6818
```

```
$ ./device/nexell/tools/build.sh -b s5p6818_drone -t kernel
```

或者

```
$ ./make.sh kernel
```

```
ours@ubuntu:~/ours6818/android_6818$ cd /home/ours/ours6818/android_6818/
ours@ubuntu:~/ours6818/android_6818$ ./device/nexell/tools/build.sh -b s5p6818_drone -t kernel
```

说明：与常规修改 Makefile 指定交叉编译工具不同，编译工具链均在 device/nexell/tools/build.sh 编译脚

本中设置，可阅读编译脚本了解编译过程。

编译需要一段时间，其编译过程会打印在终端。编译成功后在将在 kernel/arch/arm/boot 目录下生成 Image、zImage、uImage 目标文件，其中 Image 是内核编译生成的内核映像文件，zImage 是将 Image 利用 gzip 压缩工具压缩后加上 gzip 自解压工具形成的内核镜像文件。uImage 是 uboot 专用的映像文件，它是在 zImage 之前加上一个长度为 0x40 (64) 字节的“头”，说明这个映像文件的类型、加载位置、生成时间、大小等信息。换句话说，如果直接从 uImage 的 0x40 位置开始执行，zImage 和 uImage 没有任何区别，它是利用 u-boot 中的 mkimage 工具生成的。uImage 是供 u-boot 引导的内核映像。

```
ours@ubuntu:~/ours6818/android_6818/kernel/arch/arm/boot$ ls
bootp compressed dts Image install.sh Makefile uImage zImage
ours@ubuntu:~/ours6818/android_6818/kernel/arch/arm/boot$
```

同时脚本会将生成的目标文件拷贝至/home/ours/ours6818/android_6818/result/boot 目录，并将 boot 目录打包成 boot.img 存放在 result 目录。当 result 目录成功生成 boot.img，说明内核编译成功。

```
ours@ubuntu:~/ours6818/android_6818/result/boot$ ls
battery.bmp logo.bmp ramdisk-recovery.img ramdisk_update.gz root.img.gz uImage update.bmp
ours@ubuntu:~/ours6818/android_6818/result/boot$
```

```
ours@ubuntu:~/ours6818/android_6818/result$ ls
2ndboot.bin boot cache partmap.txt root.img.gz system.img userdata
arm_arch boot.img cache.img root system u-boot.bin userdata.img
ours@ubuntu:~/ours6818/android_6818/result$
```

至此内核编译已完成，烧写需要使用 boot.img 文件。

7.3 编译 Android

7.3.1 实验目的

了解掌握 Android 系统编译方法。

7.3.2 实验内容

编译 Android 系统。

7.3.3 实验设备

硬件：配置完开发环境 ubuntu 虚拟机的 PC 机 1 台

软件：Android 源码包

7.3.4 实验原理

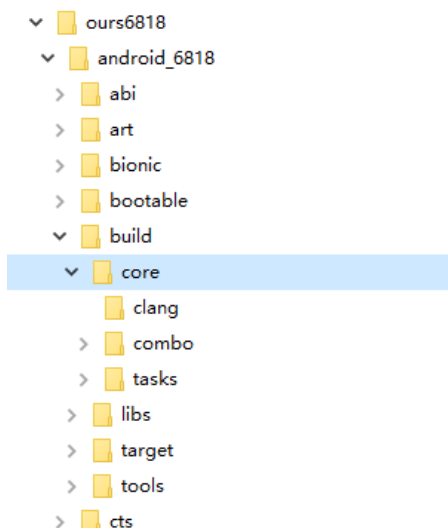
一、Android 编译系统简介

Android 编译系统是用来编译 Android 系统、Android SDK 以及相关文档的一套框架。Android 是一个开源的操作系统。Android 的源码中包含了许许多多的模块。不同产商的不同设备对于 Android 系统的定制都是不一样的。如何将这些模块统一管理起来，如何能够在不同的操作系统上进行编译，如何在编译时能够支持面向不同的硬件设备，不同的编译类型，且还要提供面向各个产商的定制扩展，是非常有难度的。但 Android 编译系统很好的解决了这些问题。

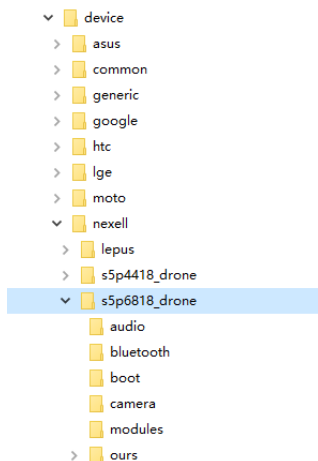
二、Android 编译系统概述

编译系统中最主要的处理逻辑都在 Make 文件中，而其他的脚本文件只是起到一些辅助作用。整个 Build 编译系统中的 Make 文件可以分为三类：

- 第一类是 Build 系统核心文件：此类文件定义了整个 Build 系统的框架，而其他所有 Make 文件都是在这个框架的基础上编写出来的。Android 源码树的目录结构如下图，Build 系统核心文件全部位于 /build/core 目录下。



- 第二类是针对某个产品的 Make 文件：这些文件通常位于 device 目录下，该目录下又以公司名以及产品名分为两级目录，下图是 device 目录下子目录的结构。对于一个产品的定义通常需要一组文件，这些文件共同构成了对于这个产品的定义。例如，device/nexell/s5p6818_drone 目录下的文件共同构成了对于 S5P6818_drone 平台的定义。



- 第三类是针对某个模块的 Make 文件：整个系统中，包含了大量的模块，每个模块都有一个专门的 Make 文件，这类文件的名称统一为“Android.mk”，该文件中定义了如何编译当前模块。Build 系统会在整个源码树中扫描名称为“Android.mk”的文件并根据其中的内容执行模块的编译。

三、执行编译

Android 系统的编译环境目前只支持 Ubuntu 以及 Mac OS 两种操作系统。在完成编译环境的准备工作以及获取到完整的 Android 源码之后，可以使用如下三条指令编译 Android

```
$ source build/envsetup.sh
```

```
$ lunch aosp_s5p6818_drone-userdebug
```

```
$ make -j8
```

第一行命令“source build/envsetup.sh”引入了 build/envsetup.sh 脚本。该脚本的作用是初始化编译环境，并引入一些辅助的 Shell 函数，这其中就包括第二步使用 lunch 函数。除此之外，该文件中还定义了一些其他的函数。build/envsetup.sh 中定义的常用函数如下。

名称	说明
croot	切换到源码树的根目录
m	在源码树的根目录执行 make
mm	Build 当前目录下的模块
mmm	Build 指定目录下的模块
cgrep	在所有 C/C++ 文件上执行 grep
jgrep	在所有 Java 文件上执行 grep
resgrep	在所有 res/*.xml 文件上执行 grep
godir	转到包含某个文件的目录路径
printconfig	显示当前 Build 的配置信息
add_lunch_combo	在 lunch 函数的菜单中添加一个条目

第二行命令“lunch aosp_s5p6818_drone-userdebug”是调用 lunch 函数，并指定参数为“aosp_s5p6818_drone-userdebug”。lunch 函数的参数用来指定此次编译的目标设备以及编译类型。在这里，这两个值分别是“aosp_s5p6818_drone”和“userdebug”。“aosp_s5p6818_drone”是 Android 源码中已经定义好的一种产品而设置的。而编译类型会影响最终系统中包含的模块，关于编译类型将在后续详细讲解。

如果调用 lunch 函数的时候没有指定参数，那么该函数将输出列表以供选择，该列表的内容会根据当前 Build 系统中包含的产品配置而不同，此时可以通过输入编号或者名称进行选择。


```
Lunch menu... pick a combo:
1. aosp_arm-eng
2. aosp_arm64-eng
3. aosp_mips-eng
4. aosp_mips64-eng
5. aosp_x86-eng
6. aosp_x86_64-eng
7. aosp_floounder-userdebug
8. aosp_lepus-userdebug
9. aosp_s5p4418_drone-userdebug
10. aosp_s5p6818_drone-userdebug
11. aosp_s5p6818_drone64-userdebug
12. aosp_hammerhead-userdebug
13. aosp_mako-userdebug
14. aosp_manta-userdebug
```

第三行命令“make”才真正开始执行编译。make 的参数“-j”指定了同时编译的 Job 数量，这是个整数，该值通常是编译主机 CPU 支持的并发线程总数的 1 倍或 2 倍（例如：在一个 4 核，每个核支持两个线程的 CPU 上，可以使用 make -j8 或 make -j16）。在调用 make 命令时，如果没有指定任何目标，则将使用默认的名称为“droid”目标，该目标会编译出完整的 Android 系统镜像。

四、编译结果

所有的编译生成文件都将位于 out 目录下，该目录下主要有以下几个子目录：

- out/host/：该目录下包含了针对主机的 Android 开发工具的产物。即 SDK 中的各种工具，例如：emulator，adb，aapt 等。
- out/target/common/：该目录下包含了针对设备的共通的编译产物，主要是 Java 应用代码和 Java 库。
- out/target/product/s5p6818_drone/：包含了针对特定设备的编译结果以及平台相关的 C/C++库和二进制文件。其中，s5p6818_drone 是具体目标设备的名称。
- /out/dist/：包含了为多种分发而准备的包，通过“make disttarget”将文件拷贝到该目录，默认的编译目标不会产生该目录。

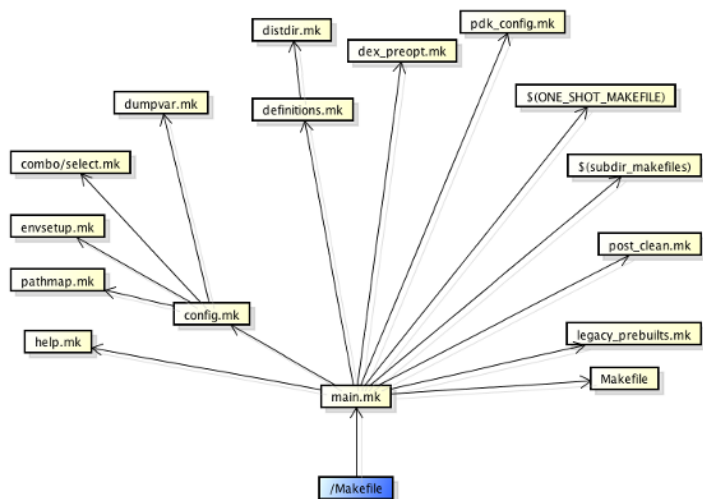
Build 的产物中最重要的是三个镜像文件，它们都位于 /out/target/product/<product_name>/ 目录下。主要包含如下文件：

- system.img：包含了 Android OS 的系统文件，库，可执行文件以及预置的应用程序，将被挂载为根分区。
- ramdisk.img：在启动时将被 Linux 内核挂载为只读分区，它包含了/init 文件和一些配置文件。它用来挂载其他系统镜像并启动 init 进程。
- userdata.img：将被挂载为/data，包含了应用程序相关的数据以及和用户相关的数据。

五、Android 系统 Make 文件

整个 Build 系统的入口文件是源码树根目录下名称为“Makefile”的文件，当在源代码根目录上调用 make 命令时，make 命令首先将读取该文件。Makefile 文件的内容只有一行：“include build/core/main.mk”。该行代码的作用很明显：包含 build/core/main.mk 文件。在 main.mk 文件中又会包含其他的文件，其他文件中又会包含更多的文件，这样就引入了整个 Build 系统。

这些 Make 文件间的包含关系是相当复杂的，下图描述了这种关系，该图中黄色标记的文件（且除了 \$开头的文件）都位于 build/core/目录下。



主要的 make 文件及其说明

文件名	说明
main.mk	最主要的 Make 文件，该文件中首先将对编译环境进行检查，同时引入其他的 Make 文件。另外，该文件中还定义了几个最主要的 Make 目标，例如 droid，sdk，等（参见后文“Make 目标说明”）。
help.mk	包含了名称为 help 的 Make 目标的定义，该目标将列出主要的 Make 目标及其说明。
pathmap.mk	将许多头文件的路径通过名值对的方式定义为映射表，并提供 include-path-for 函数来获取。例如，通过 \$(call include-path-for, frameworks-native)便可以获取到 framework 本地代码需要的头文件路径。
envsetup.mk	配置 Build 系统需要的环境变量，例如：TARGET_PRODUCT，TARGET_BUILD_VARIANT，HOST_OS，HOST_ARCH 等。当前编译的主机平台信息（例如操作系统，CPU 类型等信息）就是在这个文件中确定的。 另外，该文件中还指定了各种编译结果的输出路径。
combo/select.mk	根据当前编译器的平台选择平台相关的 Make 文件。
dumpvar.mk	在 Build 开始之前，显示此次 Build 的配置信息。
config.mk	整个 Build 系统的配置文件，最重要的 Make 文件之一。该文件中主要包含以下内容： 定义了许多的常量来负责不同类型模块的编译。 定义编译器参数以及常见文件后缀，例如 .zip, .jar, .apk。 根据 BoardConfig.mk 文件，配置产品相关的参数。 设置一些常用工具的路径，例如 flex, e2fsck, dx。
definitions.mk	最重要的 Make 文件之一，在其中定义了大量的函数。这些函数都是 Build 系统的其他文件将用到的。例如：my-dir，all-subdir-makefiles，find-subdir-files，sign-package 等，关于这些函数的说明请参见每个函数的代码注释。
distdir.mk	针对 dist 目标的定义。dist 目标用来拷贝文件到指定路径。
dex_preopt.mk	针对启动 jar 包的预先优化。
pdk_config.mk	顾名思义，针对 pdk（Platform Developement Kit）的配置文件。
\$(ONE_SHOT_MAKEFILE)	ONE_SHOT_MAKEFILE 是一个变量，当使用“mm”编译某个目录下的模块时，此变量的值即为当前指定路径下的 Make 文件的路径。
\$(subdir_makefiles)	各个模块的 Android.mk 文件的集合，这个集合是通过 Python 脚本扫描得到

	的。
post_clean.mk	在前一次 Build 的基础上检查当前 Build 的配置，并执行必要清理工作。
legacy_prebuilts.mk	该文件中只定义了 GRANDFATHERED_ALL_PREBUILT 变量。
Makefile	被 main.mk 包含，该文件中的内容是辅助 main.mk 的一些额外内容。

Android 源码中包含了许多的模块，模块的类型有很多种，例如：Java 库，C/C++ 库，APK 应用，以及可执行文件等。并且，Java 或者 C/C++ 库还可以分为静态的或者动态的，库或可执行文件既可能是针对设备（本文的“设备”指的是 Android 系统将被安装的设备，例如某个平台目标板）的也可能是针对主机（本文的“主机”指的是开发 Android 系统的机器，例如装有 Ubuntu 操作系统的 PC 机或装有 MacOS 的 iMac 或 Macbook）的。不同类型的模块的编译步骤和方法是不一样的，为了能够一致且方便的执行各种类型模块的编译，在 config.mk 中定义了许多的常量，这其中的每个常量描述了一种类型模块的编译方式，这些常量有：

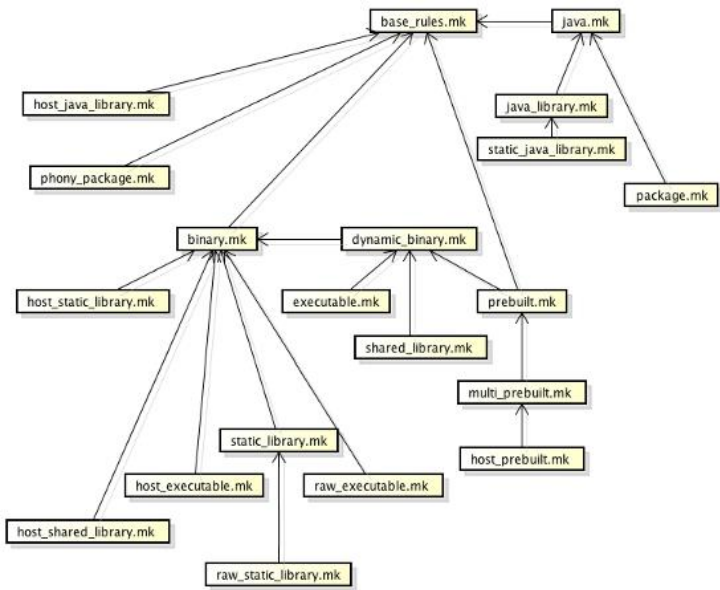
- BUILD_HOST_STATIC_LIBRARY
- BUILD_HOST_SHARED_LIBRARY
- BUILD_STATIC_LIBRARY
- BUILD_SHARED_LIBRARY
- BUILD_EXECUTABLE
- BUILD_HOST_EXECUTABLE
- BUILD_PACKAGE
- BUILD_PREBUILT
- BUILD_MULTI_PREBUILT
- BUILD_HOST_PREBUILT
- BUILD_JAVA_LIBRARY
- BUILD_STATIC_JAVA_LIBRARY
- BUILD_HOST_JAVA_LIBRARY

通过名称大概就可以猜出每个变量所对应的模块类型。（在模块的 Android.mk 文件中，只要包含进这里对应的常量便可以执行相应类型模块的编译。这些常量的值都是另外一个 Make 文件的路径，详细的编译方式都是在对应的 Make 文件中定义的。这些常量和 Make 文件的是一一对应的，对应规则也很简单：常量的名称是 Make 文件的文件名除去后缀全部改为大写然后加上“BUILD_”作为前缀。例如常量 BUILD_HOST_PREBUILT 的值对应的文件就是 host_prebuilt.mk。这些 Make 文件的说明如下表：

文件名	说明
host_static_library.mk	定义了如何编译主机上的静态库。
host_shared_library.mk	定义了如何编译主机上的共享库。
static_library.mk	定义了如何编译设备上的静态库。
shared_library.mk	定义了如何编译设备上的共享库。
executable.mk	定义了如何编译设备上的可执行文件。
host_executable.mk	定义了如何编译主机上的可执行文件。
package.mk	定义了如何编译 APK 文件。
prebuilt.mk	定义了如何处理一个已经编译好的文件（例如 Jar 包）。
multi_prebuilt.mk	定义了如何处理一个或多个已编译文件，该文件的实现依赖 prebuilt.mk。
host_prebuilt.mk	处理一个或多个主机上使用的已编译文件，该文件的实现依赖 multi_prebuilt.mk。
java_library.mk	定义了如何编译设备上的共享 Java 库。
static_java_library.mk	定义了如何编译设备上的静态 Java 库。
host_java_library.mk	定义了如何编译主机上的共享 Java 库。

不同类型的模块的编译过程会有一些相同的步骤，例如：编译一个 Java 库和编译一个 APK 文件都需要

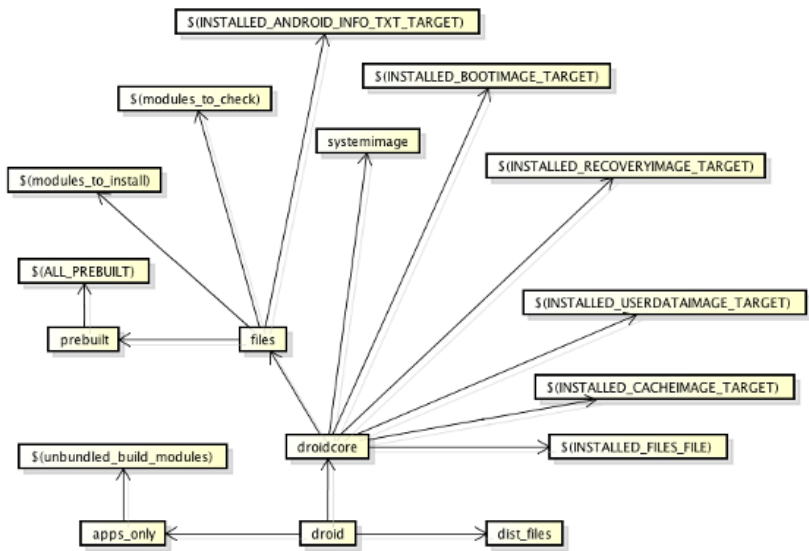
定义如何编译 Java 文件。因此，上面提到的这些 Make 文件的定义中会包含一些共同的代码逻辑。为了减少代码冗余，需要将共同的代码复用起来，复用的方式是将共同代码放到专门的文件中，然后在其他文件中包含这些文件的方式来实现的。这些包含关系如下图



六、make 目标

如果在源码树的根目录直接调用“make”命令而不指定任何目标，则会选择默认目标：“droid”（在 main.mk 中定义）。因此，这和执行“make droid”效果是一样的。

droid 目标将编译出整个系统的镜像。从源代码到编译出系统镜像，整个编译过程非常复杂。这个过程并不是在 droid 一个目标中定义的，而是 droid 目标会依赖许多其他的目标，这些目标的互相配合导致了整个系统的编译。droid 目标所依赖的其他目标如下所示：



名称	说明
apps_only	该目标将编译出当前配置下不包含 user, userdebug, eng 标签的应用程序。
droidcore	该目标仅仅是所依赖的几个目标的组合，其本身不做更多的处理。
dist_files	该目标用来拷贝文件到/out/dist 目录。

files	该目标仅仅是所依赖的几个目标的组合，其本身不做更多的处理。
prebuilt	该目标依赖于 \$(ALL_PREBUILT)，\$(ALL_PREBUILT) 的作用就是处理所有已编译好的文件。
\$(modules_to_install)	modules_to_install 变量包含了当前配置下所有会被安装的模块（一个模块是否会被安装依赖于该产品的配置文件，模块的标签等信息），因此该目标将导致所有会被安装的模块的编译。
\$(modules_to_check)	该目标用来确保我们定义的构建模块是没有冗余的。
\$(INSTALLED_ANDROID_INFO_TXT_TARGET)	该目标会生成一个关于当前 Build 配置的设备信息的文件，该文件的生成路径是： out/target/product/<product_name>/android-info.txt
systemimage	生成 system.img。
\$(INSTALLED_BOOTIMAGE_TARGET)	生成 boot.img。
\$(INSTALLED_RECOVERYIMAGE_TARGET)	生成 recovery.img。
\$(INSTALLED_USERDATAIMAGE_TARGET)	生成 userdata.img。
\$(INSTALLED_CACHEIMAGE_TARGET)	生成 cache.img。
\$(INSTALLED_FILES_FILE)	该目标会生成 out/target/product/<product_name>/installed-files.txt 文件，该文件中内容是当前系统镜像中已经安装的文件列表。

Build 系统中包含的其他一些 Make 目标说明如下表：

Make 目标	说明
make clean	执行清理，等同于 rm -rf out/。
make sdk	编译出 Android 的 SDK。
make clean-sdk	清理 SDK 的编译产物。
make update-api	更新 API。在 framework API 改动之后，需要首先执行该命令来更新 API，公开的 API 记录在 frameworks/base/api 目录下。
make dist	执行 Build，并将 MAKECMDGOALS 变量定义的输出文件拷贝到 /out/dist 目录。
make all	编译所有内容，不管当前产品的定义中是否会包含。
make help	帮助信息，显示主要的 make 目标。
make snod	从已经编译出的包快速重建系统镜像。
make libandroid_runtime	编译所有 JNI framework 内容。
make framework	编译所有 Java framework 内容。
make services	编译系统服务和相关内容。
make <local_target>	编译一个指定的模块，local_target 为模块的名称。
make clean <local_target>	清理一个指定模块的编译结果。
make dump-products	显示所有产品的编译配置信息，例如：产品名，产品支持的地区语言，产品中会包含的模块等信息。
make PRODUCT-xxx-yyy	编译某个指定的产品。
Make bootimage	生成 boot.img
make recoveryimage	生成 recovery.img
make userdataimage	生成 userdata.img

makecacheimage	生成 cache.img
----------------	--------------

七、添加新的产品

当我们要开发一款新的 Android 产品的时候，我们首先就需要在 Build 系统中添加对于该产品的定义。在 Android Build 系统中对产品定义的文件通常位于 device 目录下（另外还有一个可以定义产品的目录是 vender 目录，这是个历史遗留目录，Google 已经建议不要在该目录中进行定义，而应当选择 device 目录）。device 目录下根据公司名以及产品名分为二级目录，这一点我们在概述中已经提到过。

通常，对于一个产品的定义通常至少会包括四个文件：AndroidProducts.mk，产品版本定义文件，BoardConfig.mk 以及 vendorsetup.sh。下面我们来详细说明这几个文件。

- AndroidProducts.mk：该文文件中的内容很简单，其中只需要定义一个变量，名称为“PRODUCT_MAKEFILES”，该变量的值为产品版本定义文件名的列表，例如：

```
PRODUCT_MAKEFILES := \
    $(LOCAL_DIR)/aosp_s5p6818_drone.mk \
    $(LOCAL_DIR)/aosp_s5p6818_drone64.mk
```

产品版本定义文件：顾名思义，该文件中包含了对于特定产品版本的定义。该文件可能不只一个，因为同一个产品可能会有多种版本（例如，面向中国地区一个版本，面向美国地区一个版本）。该文件中可以定义的变量以及含义说明如下表所示

常量	说明
PRODUCT_NAME	最终用户将看到的完整产品名，会出现在“关于手机”信息中。
PRODUCT_MODEL	产品的型号，这也是最终用户将看到的。
PRODUCT_LOCALES	该产品支持的地区，以空格分格，例如：en_GB de_DE es_ES fr_CA。
PRODUCT_PACKAGES	该产品版本中包含的 APK 应用程序，以空格分格，例如:Calendar Contacts。
PRODUCT_DEVICE	该产品的工业设计的名称。
PRODUCT_MANUFACTURER	制造商的名称。
PRODUCT_BRAND	该产品专门定义的商标（如果有的话）。
PRODUCT_PROPERTY_OVERRIDES	对于商品属性的定义。
PRODUCT_COPY_FILES	编译该产品时需要拷贝的文件，以“源路径：目标路径”的形式。
PRODUCT_OTA_PUBLIC_KEYS	对于该产品的 OTA 公开 key 的列表。
PRODUCT_POLICY	产品使用的策略。
PRODUCT_PACKAGE_OVERLAYS	指出是否要使用默认的资源或添加产品特定定义来覆盖。
PRODUCT_CONTRIBUTORS_FILE	HTML 文件，其中包含项目的贡献者。
PRODUCT_TAGS	该产品的标签，以空格分格。

通常情况下，我们并不需要定义所有这些变量。Build 系统的已经预先定义好了一些组合，它们都位于 /build/target/product 下，每个文件定义了一个组合，我们只要继承这些预置的定义，然后再覆盖自己想要的变量定义即可。

- BoardConfig.mk：该文件用来配置硬件主板，它其中定义的都是设备底层的硬件特性。例如：该设备的主板相关信息，Wifi 相关信息，还有 bootloader，内核，radioimage 等信息。对于该文件的示例，请参看 Android 源码树已经有的文件。
- vendorsetup.sh：该文件中作用是通过 add_lunch_combo 函数在 lunch 函数中添加一个菜单选项。该函数的参数是产品名称加上编译类型，中间以“-”连接，例如：add_lunch_combo aosp_s5p6818_drone-userdebug。/build/envsetup.sh 会扫描所有 device 和 vender 二级目录下的名称为"vendorsetup.sh"文件，并根据其中的内容来确定 lunch 函数的菜单选项。

在配置了以上的文件之后，便可以编译出我们新添加的设备的系统镜像了。首先，调用“source build/envsetup.sh”该命令的输出中会看到 Build 系统已经引入了刚刚添加的 vendorsetup.sh 文件。然后再调用“lunch”函数，该函数输出的列表中将包含新添加的 vendorsetup.sh 中添加的条目。然后通过编号或名称选择即可。最后，调用“make -j8”来执行编译即可。

八、添加新的模块

在源码树中，一个模块的所有文件通常都位于同一个文件夹中。为了将当前模块添加到整个 Build 系统中，每个模块都需要一个专门的 Make 文件，该文件的名称为“Android.mk”。Build 系统会扫描名称为“Android.mk”的文件，并根据该文件中内容编译出相应的产物。

在 Android Build 系统中，编译是以模块（而不是文件）作为单位的，每个模块都有一个唯一的名称，一个模块的依赖对象只能是另外一个模块，而不能是其他类型的对象。对于已经编译好的二进制库，如果要用被当作是依赖对象，那么应当将这些已经编译好的库作为单独的模块。对于这些已经编译好的库使用 BUILD_PREBUILT 或 BUILD_MULTI_PREBUILT。例如：当编译某个 Java 库需要依赖一些 Jar 包时，并不能直接指定 Jar 包的路径作为依赖，而必须首先将这些 Jar 包定义为一个模块，然后在编译 Java 库的时候通过模块的名称来依赖这些 Jar 包。下面，我们就来讲解 Android.mk 文件的编写：

Android.mk 文件通常以下两行代码作为开头：

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

这两行代码的作用是：设置当前模块的编译路径为当前文件夹路径。清理（可能由其他模块设置过的）编译环境中用到的变量。

为了方便模块的编译，Build 系统设置了很多的编译环境变量。要编译一个模块，只要在编译之前根据需要设置这些变量然后执行编译即可。它们包括：

- LOCAL_SRC_FILES：当前模块包含的所有源代码文件。
- LOCAL_MODULE：当前模块的名称，这个名称应当是唯一的，模块间的依赖关系就是通过这个名称来引用的。
- LOCAL_C_INCLUDES：C 或 C++ 语言需要的头文件的路径。
- LOCAL_STATIC_LIBRARIES：当前模块在静态链接时需要的库的名称。
- LOCAL_SHARED_LIBRARIES：当前模块在运行时依赖的动态库的名称。
- LOCAL_CFLAGS：提供给 C/C++ 编译器的额外编译参数。
- LOCAL_JAVA_LIBRARIES：当前模块依赖的 Java 共享库。
- LOCAL_STATIC_JAVA_LIBRARIES：当前模块依赖的 Java 静态库。
- LOCAL_PACKAGE_NAME：当前 APK 应用的名称。
- LOCAL_CERTIFICATE：签署当前应用的证书名称。
- LOCAL_MODULE_TAGS：当前模块所包含的标签，一个模块可以包含多个标签。标签的值可能是 debug, eng, user, development 或者 optional。其中，optional 是默认标签。标签是提供给编译类型使用的。不同的编译类型会安装包含不同标签的模块，关于编译类型的说明如下表所示：

名称	说明
eng	默认类型，该编译类型适用于开发阶段。当选择这种类型时，编译结果将： <ul style="list-style-type: none">➢ 安装包含 eng, debug, user, development 标签的模块➢ 安装所有没有标签的非 APK 模块➢ 安装所有产品定义文件中指定的 APK 模块
user	该编译类型适合用于最终发布阶段。当选择这种类型时，编译结果将： <ul style="list-style-type: none">➢ 安装所有带有 user 标签的模块➢ 安装所有没有标签的非 APK 模块➢ 安装所有产品定义文件中指定的 APK 模块，APK 模块的标签将被忽略

userdebug	该编译类型适合用于 debug 阶段。该类型和 user 一样，除了： <ul style="list-style-type: none"> ➤ 会安装包含 debug 标签的模块 ➤ 编译出的系统具有 root 访问权限
------------------	---

Android 源码文件已经定义好了各种类型模块的编译方式。所以要执行编译，只需要引入对应的 Make 文件即可（通过常量的方式）。例如，要编译一个 APK 文件，只需要在 Android.mk 文件中，加入 “include \$(BUILD_PACKAGE)”。除此以外，Build 系统中还定义了一些便捷的函数以便在 Android.mk 中使用，包括：

- \$(call my-dir)：获取当前文件夹路径。
- \$(call all-java-files-under, <src>)：获取指定目录下的所有 Java 文件。
- \$(call all-c-files-under, <src>)：获取指定目录下的所有 C 语言文件。
- \$(call all-aidl-files-under, <src>)：获取指定目录下的所有 AIDL 文件。
- \$(call all-makefiles-under, <folder>)：获取指定目录下的所有 Make 文件。
- \$(call intermediates-dir-for, <class>, <app_name>, <host or target>, <common?>)：获取 Build 输出的目标文件夹路径。

编译 APK 文件的 Make 文件示例：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
# 获取所有子目录中的 Java 文件
LOCAL_SRC_FILES := $(call all-subdir-java-files)
# 当前模块依赖的静态 Java 库，如果有多个以空格分隔
LOCAL_STATIC_JAVA_LIBRARIES := static-library
# 当前模块的名称
LOCAL_PACKAGE_NAME := LocalPackage
# 编译 APK 文件
include $(BUILD_PACKAGE)
```

编译 Java 静态库的 Make 文件示例：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
# 获取所有子目录中的 Java 文件
LOCAL_SRC_FILES := $(call all-subdir-java-files)
# 当前模块依赖的动态 Java 库名称
LOCAL_JAVA_LIBRARIES := android.test.runner
# 当前模块的名称
LOCAL_MODULE := sample
# 将当前模块编译成一个静态的 Java 库
include $(BUILD_STATIC_JAVA_LIBRARY)
```

7.3.5 实验步骤

① 初始化编译环境

进入 Android 源码目录，其中的 build 下存放了配置编译环境的脚本 envsetup.sh，执行环境配置脚本，必须用 source 命令执行。

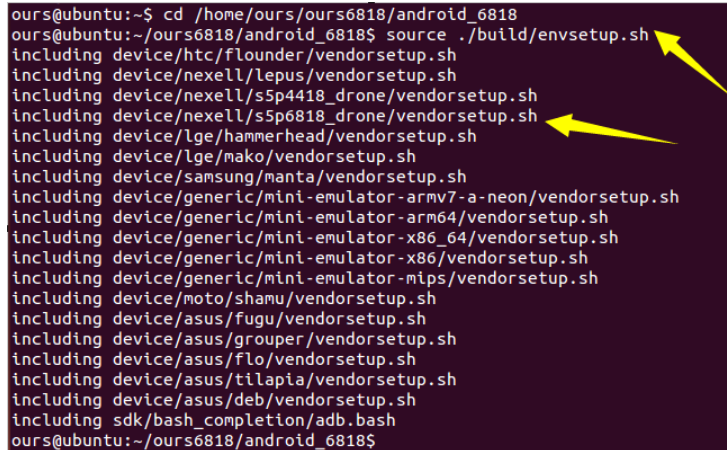
envsetup.sh 编译环境配置脚本主要的作用如下：

- 加载编译时使用到的函数命令
- 添加编译选项

- 查找<厂商目录>下的 vendorsetup.sh 如果存在的话，就加载执行，具体到本系统，加载 device/<厂商目录>下不同平台 vendorsetup.sh 脚本，对于本平台加载脚本且有意义的是 device/nexell/s5p6818_drone/vendorsetup.sh 文件
- 添加厂商自己定义的编译选项

```
$ cd /home/ours/ours6818/android_6818
```

```
$ source ./build/envsetup.sh
```



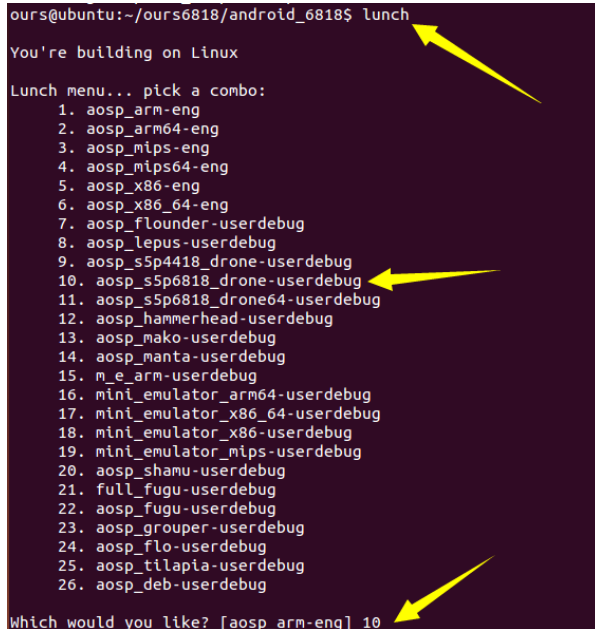
```
ours@ubuntu:~$ cd /home/ours/ours6818/android_6818
ours@ubuntu:~/ours6818/android_6818$ source ./build/envsetup.sh
including device/htc/flounder/vendorsetup.sh
including device/nexell/lepus/vendorsetup.sh
including device/nexell/s5p4418_drone/vendorsetup.sh
including device/nexell/s5p6818_drone/vendorsetup.sh
including device/lge/hammerhead/vendorsetup.sh
including device/lge/mako/vendorsetup.sh
including device/samsung/manta/vendorsetup.sh
including device/generic/mini-emulator-armv7-a-neon/vendorsetup.sh
including device/generic/mini-emulator-arm64/vendorsetup.sh
including device/generic/mini-emulator-x86_64/vendorsetup.sh
including device/generic/mini-emulator-x86/vendorsetup.sh
including device/generic/mini-emulator-mips/vendorsetup.sh
including device/moto/shamu/vendorsetup.sh
including device/asus/fugu/vendorsetup.sh
including device/asus/grouper/vendorsetup.sh
including device/asus/flo/vendorsetup.sh
including device/asus/tilapia/vendorsetup.sh
including device/asus/deb/vendorsetup.sh
including sdk/bash_completion/adb.bash
ours@ubuntu:~/ours6818/android_6818$
```

在编译环境配置脚本中包含了各个平台的编译环境配置，我们这里主要使用的是 S5P6818 的编译环境配置，如上图箭头所示。

②指定目标设备及编译类型

完成编译环境配置后，选择编译目标包，执行 lunch 命令选择【10】。

```
$ lunch
```



```
ours@ubuntu:~/ours6818/android_6818$ lunch
You're building on Linux

Lunch menu... pick a combo:
 1. aosp_arm-eng
 2. aosp_arm64-eng
 3. aosp_mips-eng
 4. aosp_mips64-eng
 5. aosp_x86-eng
 6. aosp_x86_64-eng
 7. aosp_flounder-userdebug
 8. aosp_lepus-userdebug
 9. aosp_s5p4418_drone-userdebug
10. aosp_s5p6818_drone-userdebug
11. aosp_s5p6818_drone64-userdebug
12. aosp_hammerhead-userdebug
13. aosp_mako-userdebug
14. aosp_manta-userdebug
15. m_e_arm-userdebug
16. mini_emulator_arm64-userdebug
17. mini_emulator_x86_64-userdebug
18. mini_emulator_x86-userdebug
19. mini_emulator_mips-userdebug
20. aosp_shamu-userdebug
21. full_fugu-userdebug
22. aosp_fugu-userdebug
23. aosp_grouper-userdebug
24. aosp_flo-userdebug
25. aosp_tilapia-userdebug
26. aosp_deb-userdebug

Which would you like? [aosp_arm-eng] 10
```

或者直接使用如下命令

```
$ lunch aosp_s5p6818_drone-userdebug
```

```

ours@ubuntu:~/ours6818/android_6818$ lunch aosp_s5p6818_drone-userdebug

=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=5.1.1
TARGET_PRODUCT=aosp_s5p6818_drone
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
TARGET_2ND_ARCH=
TARGET_2ND_ARCH_VARIANT=
TARGET_2ND_CPU_VARIANT=
HOST_ARCH=x86_64
HOST_OS=linux
HOST_OS_EXTRA=Linux-4.4.0-127-generic-x86_64-with-Ubuntu-14.04-trusty
HOST_BUILD_TYPE=release
BUILD_ID=LMY48G
OUT_DIR=out
=====

ours@ubuntu:~/ours6818/android_6818$

```

③编译 Android

在 Android 源码下，执行编译脚本

\$./device/nexell/tools/build.sh -b s5p6818_drone -t android

```

ours@ubuntu:~/ours6818/android_6818$ ./device/nexell/tools/build.sh -b s5p6818_drone -t android

```

一般情况下，如果拿到全新的 Android 系统源码，首次对 Android 系统进行编译需要采用全局编译，全局编译会将 bootloader、kernel、Android、module 等都进行编译。全局编译通过后，如果只针对 bootloader 做了修改，可用 bootloader 编译命令单独编译 bootloader。只针对 kernel 做了修改，可用内核编译命令单独编译内核。如果单独编译出现问题，建议先全局编译。全局编译使用 **\$./device/nexell/tools/build.sh -b s5p6818_drone -t modules**

编译完成后会在 out/target/product/s5p6818_drone 目录生成系统镜像文件，同时会拷贝一份镜像到 result 目录。

7.4 使用 TF 卡烧写 Android

7.4.1 实验目的

掌握 TF 卡刷写 Android 方法。

7.4.2 实验内容

TF 卡格式化分区；使用 TF 卡烧写 Android 镜像。

7.4.3 实验设备

硬件：PC 机 1 台、OURS6818 实验平台 1 套、8GB TF 卡 1 张、TF 读卡器一个、串口线一根

软件：WinImage 软件；U-boot、Kernel、Android 镜像；串口调试软件（minicom 或 putty 均可）

7.4.4 实验原理

由于默认目标板上的 Flash 中没有任何数据，我们需要一种手段将系统镜像文件写入目标板的 Flash，由于 S5P6818 内置了 IROM 程序，其支持多种方式启动引导，所以我们可以使用 TF 卡来引导设备，同时将数据存放至 TF 卡中，在引导时将数据从 TF 卡烧写进板载 Flash 中。烧写完成后再从板级 Flash 运行。

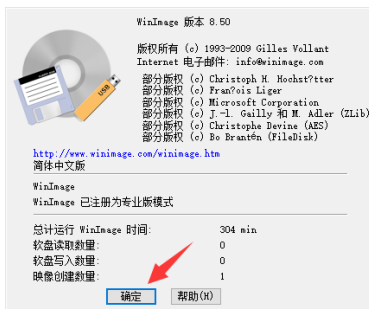
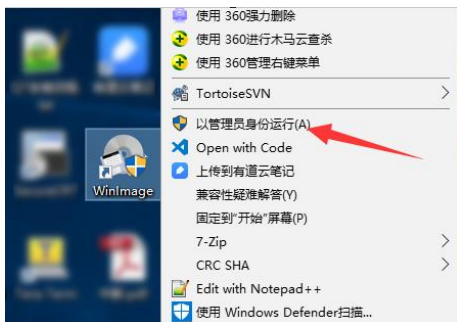
7.4.5 实验步骤

一、格式化 TF 卡

由于 S5P6818 对于引导 TF 卡有一定要求（见裸机中说明），我们需要将 TF 卡进行格式化，为了方便我们使用恢复磁盘镜像的方式可快速的对 TF 卡进行分区。使用 WinImage 软件进行操作（其安装过程比较简单详见裸机文档）。此操作在 windows 系统下进行。

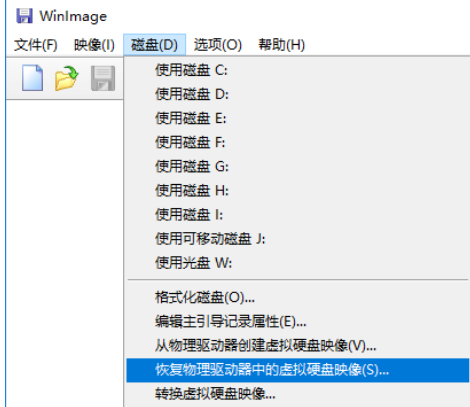
①运行 WinImage 对 TF 分区

将 TF 卡接入读卡器，并将读卡器连接到 windows PC 机上。选中 WinImage，右键以管理员权限运行（WIN7 以上系统必须以管理员权限运行，否则无法获取磁盘列表）

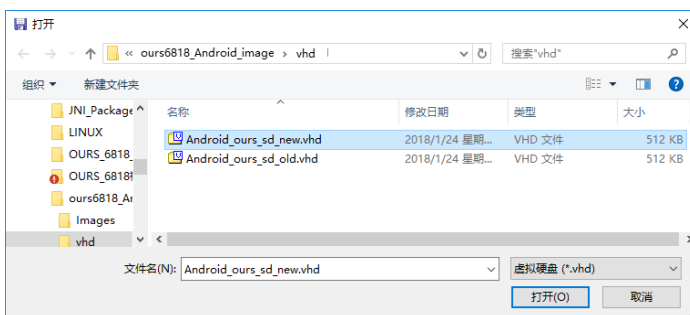
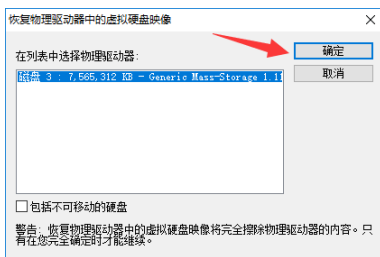


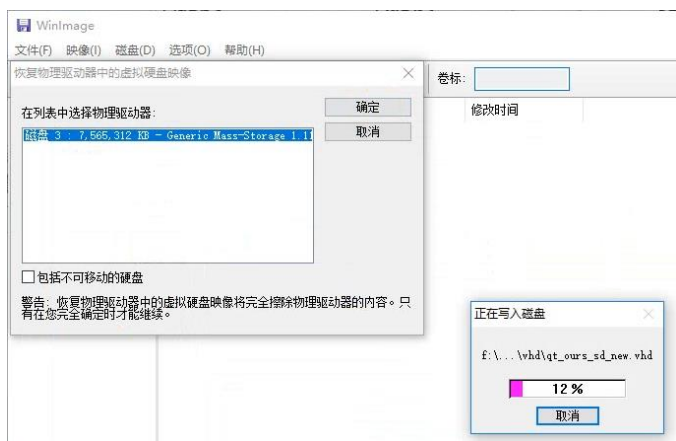


点击【磁盘】->【恢复物理驱动器中的虚拟磁盘映像】



选中 TF 卡对应磁盘（注意不要选错设备），点击【确定】，浏览并选择【Android_ours_sd_new.vhd】文件（在 ours6818_Android_image\vhd 目录中），点击【打开】，之后将自动对 TF 卡分区。

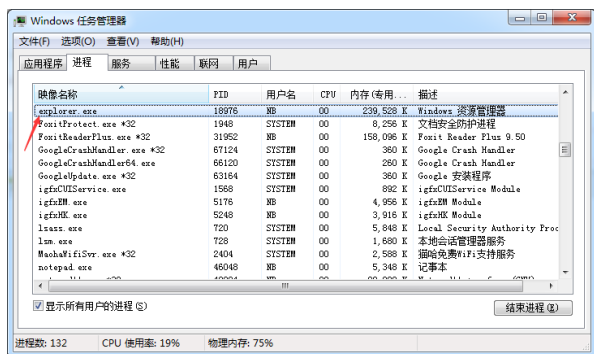




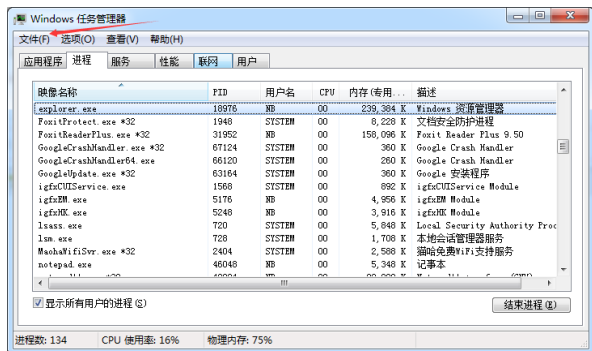
如果不出意外，将弹出如上图所示进度条，完成之后自动消失。完成以上工作后（WinImage 界面中不会有什么变化），关闭 WinImage。

出错解决：winimage.exe 制作升级卡提示程序使用此文件，进程无法访问

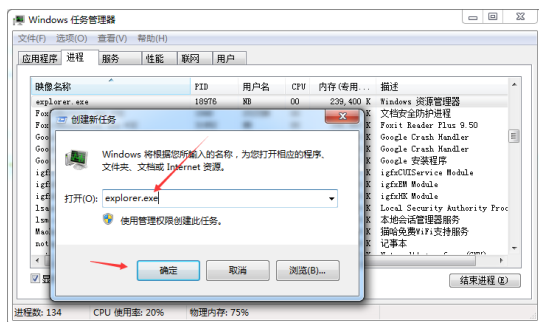
第一步：打开任务管理器，找到 explorer.exe，并且结束掉此进程



第二步：选择：文件 -> 新建任务

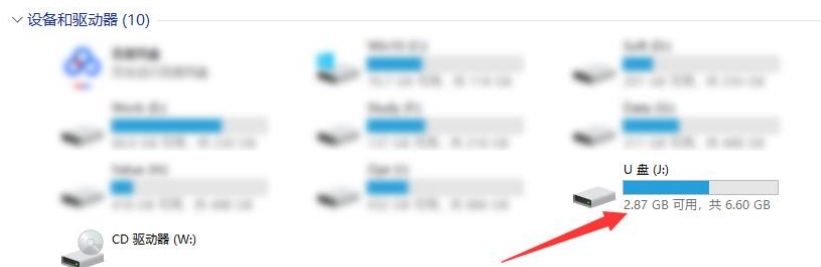


第三步：在新建任务中输入 explorer.exe，点击确定即可解决问题

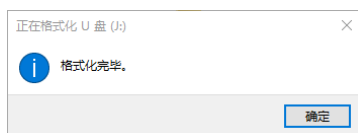
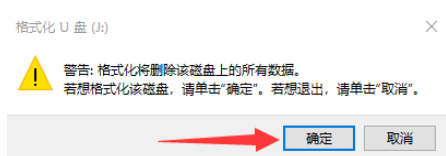
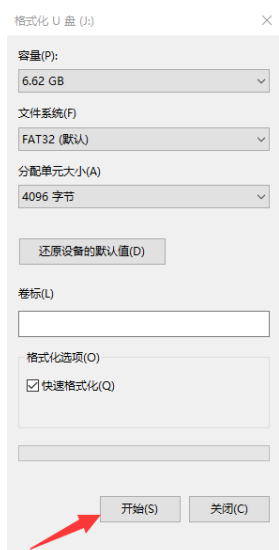
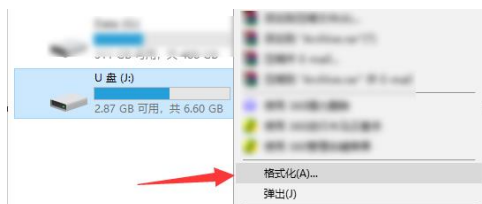


②格式化分区

成功完成上步分区工作后，在 windows 中将看到一个 6.6G 左右的磁盘分区（可能同时会出现额外其他 3 个未格式化分区，不用管它，不同的系统显示可能有别）

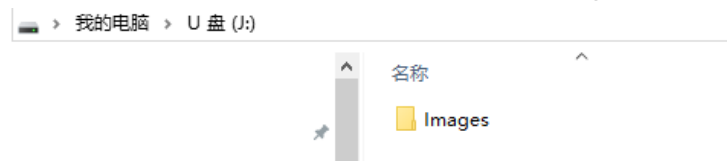


右键此分区，点击【格式化】，将此分区格式化（如果出错，请重新格式化）



③创建镜像

完成格式化工作后，进入此分区，新建一个 Images 文件夹（注意目录名必须是 Images）

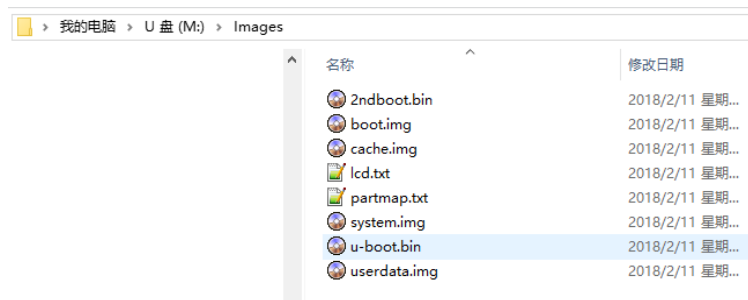


将烧写的镜像文件拷贝至此目录，需要的文件有八个，分别是：2ndboot.bin、u-boot.bin、boot.img、system.img、

userdata.img、cache.img、lcd.txt、partmap.txt。

说明：

- 2ndboot.bin 为一级引导程序，由厂家提供；
- u-boot.bin 是编译 U-boot 生成的引导程序；
- boot.img 是编译 kernel 生成的；
- system.img、userdata.img、cache.img 是编译 Android 系统生成的；
- lcd.txt 是一个专门针对 LCD 的配置文件，由于目标板可连接多种 LCD 屏及多种数据格式，为方便适配 LCD 屏，专门创建此文件，系统启动会读取此文件。
- partmap.txt 文件为 Flash 的分区信息，系统会读取此文件并按照其中的配置对 Flash 进行分区。



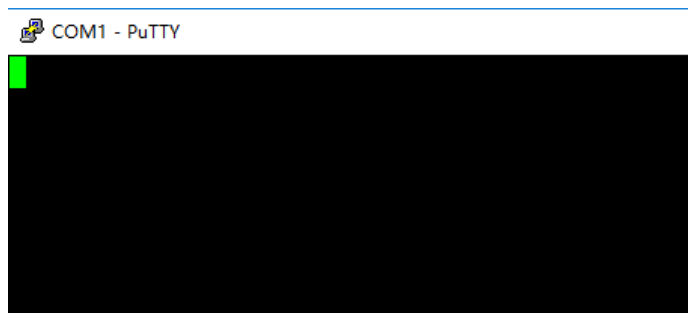
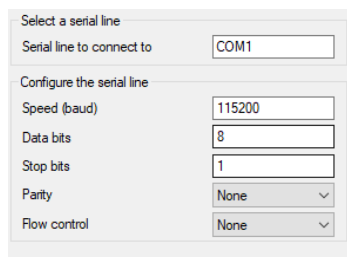
完成后 TF 卡中应具有如上图所示镜像文件，至此，引导烧写卡便制作完成。

二、烧写系统镜像

①连接目标板

将制作好的 TF 卡，插接到目标板正下方的 TF 卡槽中；使用串口线连接目标板左上侧 DB9 串口（debug 调试口）到 PC 机。将目标板电源接口使用 5V 电源适配器连接到电源。

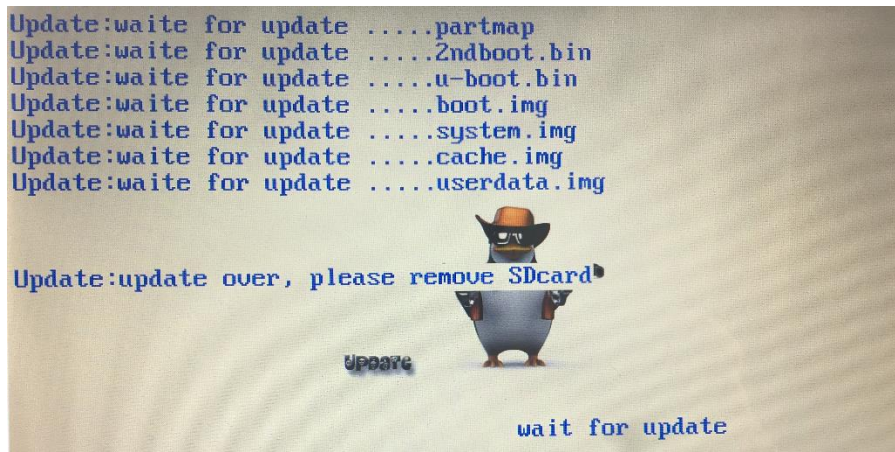
打开串口调试软件，波特率 115200，数据位 8 位，停止位 1 位，无奇偶校验，无流控。



②烧写

完成设备连接后，打开目标板电源，系统将自动将 TF 卡中的镜像文件烧写到 Flash 中，LCD 屏将显示烧写进度，烧写完成 LCD 会显示提示信息并间隔 1 秒闪烁，详细的烧写过程将通过串口输出。

LCD 屏显示信息如下图



串口输出信息如下图，笔者只截取了烧写部分的打印信息，最后打印的“file is not exist”不用理会。

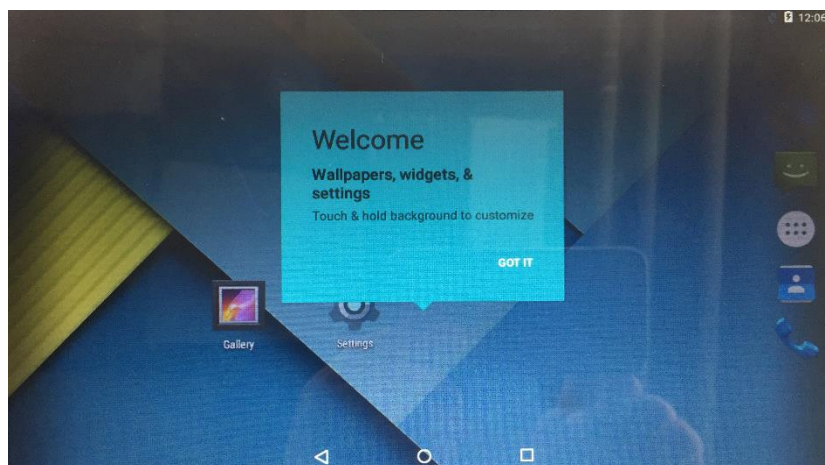
```
Flash : partmap - DONE
Read data 18 f0 9f e5 crc=4b4b01be
Read 30972 (0x000078fc) bytes
  switch to partitions #0, OK
mmc2(part 0) is current device
switch to partitions #0, OK
mmc2(part 0) is current device
head boot dev = 2
update mmc.2 type 2ndboot = 0x200(0x1) ~ 0x78fc(0x3d): Done
Flash : 2ndboot - DONE
Read data 12 0 0 ea crc=b75865ae
Read 268228 (0x000417c4) bytes
  switch to partitions #0, OK
mmc2(part 0) is current device
switch to partitions #0, OK
mmc2(part 0) is current device
head boot dev = 2
head load addr = 0x42c00000
head load size = 0x000417c4
head gignature = 0x4849534e
update mmc.2 type boot = 0x8000(0x40) ~ 0x419c4(0x20d): Done
Flash : bootloader - DONE
Read data 3a ff 26 ed crc=442e39f7
Read 22835432 (0x015c70e8) bytes
  switch to partitions #0, OK
mmc2(part 0) is current device
Flash : boot - DONE
Read data 3a ff 26 ed crc=2b635d5d
Read 527800012 (0x1f7596cc) bytes
  switch to partitions #0, OK
mmc2(part 0) is current device
Flash : system - DONE
Read data 3a ff 26 ed crc=bfd65eaf
Read 8962352 (0x0088c130) bytes
  switch to partitions #0, OK
mmc2(part 0) is current device
Flash : cache - DONE
Read data 3a ff 26 ed crc=a7104f18
Read 140917740 (0x08663bec) bytes
  switch to partitions #0, OK
mmc2(part 0) is current device
Flash : userdata - DONE
```


file is not exist

三、启动系统

当完成上述烧写步骤，未出现错误，同时 LCD 屏提示 “update over, please remove SDcard!” 时，说明系统烧写成功。将 TF 卡从目标板移除，重新给目标板上电（或直接按复位按键），系统将从 Flash 进行引导和启动，Flash 中存放的是我们刚烧写的镜像文件。

如果编译的引导程序、内核镜像、文件系统以及烧写工程没有错误的话，系统将成功引导并启动。



7.5 使用 FAST BOOT 烧写 Android

7.5.1 实验目的

掌握 FAST BOOT 烧写系统镜像方法。

7.5.2 实验内容

利用 FAST Boot 工具烧写系统镜像。

7.5.3 实验设备

硬件：PC 机 1 台、OURS6818 实验平台 1 套、8GB TF 卡 1 张、TF 读卡器一个、串口线一根、miniUSB 数据线 1 根

软件：FAST boot 软件、USB 驱动、U-boot、Kernel、Android 镜像；串口调试软件（minicom 或 putty 均可）

7.5.4 实验原理

一、Fastboot 简介

FastBoot 是 Android 系统的底层的刷机模式（俗称引导模式），是使用 USB 数据线连接目标板的一种刷机模式。用 fastboot 模式来进行目标板中各分区 img 的更新或者获取目标板某些信息非常方便，因为不需要启动内核。这也是目前各种手机、平板、手持设备最常用的一种烧写模式。

理论上，所有支持 Android 的设备都存在着 Fastboot/Bootloader 模式，BootLoader 可以分为两个阶段。在阶段一，做一些初始化，在阶段二，与用户进行交互，比如特定的按键，就会进入交互模式。在交互模式下即可使用 fastboot 更新系统。（一般情况下，设备发布版系统会屏蔽人机交互模式，也就是我们常听说的手机 bootloader 锁定，为的是避免黑客等不正当手段对设备的系统造成污染、修改等）。

进入 fastboot 模式之后，要想通过 PC 跟目标板进行通讯完成刷机等功能，PC 机上必须也要有一个 fastboot 的可执行文件，而 PC 机上的这个可执行文件的生成方式可以通过 android 平台代码编译生成，或者网络上有很多已经编译好的可执行文件。

fastboot 默认支持的命令如下：

usage: fastboot [<option>] <command>

commands:

```
update <filename> : reflash device from update.zip
flashall : flash boot + recovery + system
flash <partition> [ <filename> ] : write a file to a flash partition
erase <partition> : erase a flash partition
format <partition> : format a flash partition
getvar <variable> : display a bootloader variable
boot <kernel> [ <ramdisk> ] : download and boot kernel
flash:raw boot <kernel> [ <ramdisk> ] : create bootimage and flash it
devices : list all connected devices
continue : continue with autoboot
reboot : reboot device normally
reboot-bootloader : reboot device into bootloader
help : show this help message
```

options:

-w : erase userdata and cache (and format if supported by partition type)
-u : do not first erase partition before formatting
-s <specific device> : specify device serial number or path to device port
-l : with "devices", lists device paths
-p <product> : specify product name
-c <cmdline> : override kernel cmdline
-i <vendor id> : specify a custom USB vendor id
-b <base_addr>: specify a custom kernel base address
-n <page size>: specify the nand page size. default: 2048
-S <size>[K|M|G]: automatically sparse files greater than size. 0 to disable

注意：FASTBOOT 烧写需要目标板中已存在 bootloader 程序，也就是说在目标板上没有任何 LINUX 或 Android 引导程序时无法使用 fastboot。Fastboot 主要应用于针对已有系统进行更新升级。

二、Android 常见分区

Android 系统使用几个分区来管理文件和文件夹。每个分区在设备上都有不同的功能，Android 系统的内部存储分区主要包含如下几个：

- bootloader 分区
- boot 分区
- recovery 分区
- system 分区
- userdata 分区
- cache 分区
- misc 分区

①bootloader 分区

Bootloader 分区存放 Android 系统 Bootloader 镜像，其分为主引导和二级引导。主引导代码主要执行硬件检测，确保硬件能正常工作，然后将二级引导代码拷贝到内存(RAM)开始执行。二级引导代码进行一些硬件初始化工作，获取内存大小信息等，然后根据用户的交互进入到某种启动模式。比如说正常启动模式、recovery 模式、fastboot 模式等。fastboot 模式：fastboot 是 android 定义的一种简单的刷机协议，用户可以通过 fastboot 命令行工具来进行刷机。

②boot 分区

Boot 分区存放有 Android 的内核和内存操作程序。没有这个分区设备就不能启动。恢复系统的时候会擦除这个分区，并且必须重新安装 ROM 才能重启系统。当系统直接上电时，会进入正常启动模式，二级引导程序会跳转到 boot 分区开始启动。Boot 分区的格式是固定的，首先是一个头部，然后是 LINUX 内核，最后是根文件系统的 ramdisk。当 LINUX 内核启动完毕后，就开始执行根文件系统中的 init 程序，init 程序会读取启动脚本文件 init.rc 和 init.xxx.rc。

android 内核挂载文件系统后，根据 init.rc、init.s5p6818_drone.rc 来初始化并装载系统库、程序等直到开机完成。init.rc 脚本包括了文件系统初始化、装载的许多过程。init.rc 的工作主要是：

- 1) 设置一些环境变量
- 2) 创建 system、sdcard、data、cache 等目录
- 3) 把一些文件系统 mount 到一些目录去，如，mount tmpfs tmpfs /sqlite_stmt_journals
- 4) 设置一些文件的用户群组、权限
- 5) 设置一些线程参数
- 6) 设置 TCP 缓存大小

init 程序读取启动脚本，执行脚本中指定的动作和命令，脚本中的一部分是运行 system 分区的程序。

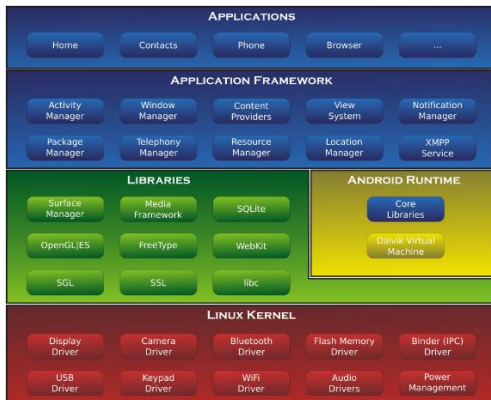
③recovery 分区

recovery 分区即恢复分区，在正常分区被破坏后，仍可以进入这一分区进行备份和恢复。这个分区保存一个简单的 OS 或底层软件，在 Android 的内核被破坏后可以用 bootloader 从这个分区引导进行操作。这个分区可以认为是一个 boot 分区的替代品，可以使你的设备进入 Recovery 程序，进行高级恢复或安卓系统维护工作。

recovery 模式：recovery 是 android 定义的一个标准刷机协议。当进入 recovery 模式时，二级引导程序从 recovery 分区开始启动，recovery 分区实际上是一个简单的 Linux 系统，当内核启动完毕后，开始执行第一个程序 init(init 程序是 Linux 系统所有程序的老祖宗)。init 会启动一个叫做 recovery 的程序（recovery 模式的名称也由此而来）。通过 recovery 程序，用户可以执行清除数据，安装刷机包等操作。一般的手机厂商都提供一个简单的 recovery 刷机，大多数只能进行 upate 的操作。

④system 分区

System 分区存放 Android 系统。这个分区基本包含了整个安卓操作系统，除了内核 (kerne) 和 ramdisk。里面包含了 Android 用户接口、用户界面、和所有预装的系统应用程序。擦除了这个分区，会删除整个安卓系统，你可以通过进入 Recovery 程序或者 bootloader 程序中，安装一个新 ROM，也就是新安卓系统。



除上图中的 linux Kernel 部分位于 boot 分区外，其他的 Library、runtime、framework、core application 都是处于 system 分区。

- /system/priv-app：特权 App，比 system_app 权限还要高，其不仅 System_app 标识是 true，同时还置了 Priv-app 标识。
- /system/app：核心应用程序 (*.apk)，都是放在这。像是 Phone、Alarm Clock, Browser, Contacts 等等。
- /system/framework：这里放 Android 系统的核心程式库，就是上图中 application framework 部分的库。像是 core.jar, framework-res.apk, framework.jar 等等。
- system/lib：上图中 Library 部分，存放的是所有动态链接库(.so 文件)，这些 SO 是 JNI 层，Dalvik 虚拟机，本地库，HAL 层所需要的，因为系统应用/system/app 下的 apk 是不会解压 SO 到程序的目录下，所以其相应用的 SO，都应放在/system/lib 下面。当一个系统 apk 的 SO 加载时，会从此目录下寻找对应用的 SO 文件；
- /system/media/audio/(notification, alarms, ringtones, ui)：这里放系统的声音文件，像是闹铃声，来电铃声等等。这些声音文件，多是 ogg 格式。
- /system/bin: 存放的是一些可执行文件，基本上是由 C/C++编写的。其中有一个重要的命令叫 app_process。一般大家称之为 Zygote。(Zygote 是卵的意思，所有的 Android 进程都是由它生出来的)。Zygote 首先会加载 dalvik 虚拟机，然后产生一个叫做 system_server 的进程。system_server 顾名思义被称作 Android 的系统服务程序，它主要管理整个 android 系统。system_server 启动完成后开始寻找一个叫做启动器的程序，找到之后由 zygote 开始启动执行启动器，这就是我们常见到的桌面程序。
- system/xbin: 存放的是一些扩展的可执行文件，既该目录可以为空。大家常用的 busybox 就放在该目录下。Busybox 所建立的各种符号链接命令都是放在该目录。

- **system/build.prop**: build.prop 和上节说得根文件系统中的 default.prop 文件格式一样，都称为属性配置文件。它们都定义了一些属性值，代码可以读取或者修改这些属性值。属性值有一些命名规范：**ro** 开头的表示只读属性，即这些属性的值代码是无法修改的。**persist** 开头的表示这些属性值会保存在文件中，这样重新启动之后这些值还保留。其它的属性一般以所属的类别开头，这些属性是可读可写的，但是对它们的修改重启之后不会保留。
- **system/etc**: 目录存放一些配置文件，和属性配置文件不一样，这下面的配置文件可能稍微没那么有规律。一般来说，一些脚本程序，还有大家所熟悉 GPS 配置文件(gps.conf)和 APN 配置文件(apns-conf.xml)放在这个目录。

⑤userdata 分区

它将挂载到 /data 目录下，它是由编译出来的 userdata.img 来烧入。这个分区也叫用户数据区，包含了用户的数据：联系人、短信、设置、用户安装的程序。擦除这个分区，本质上等同于手机恢复出厂设置，也就是手机系统第一次启动时的状态，或者是最后一次安装官方或第三方 ROM 后的状态。在 Recovery 程序中进行的“data/factory reset”操作就是在擦除这个分区。

⑥cache 分区

它将挂载到 /cache 目录下。这个分区是安卓系统缓存区，保存系统最常访问的数据和应用程序。擦除这个分区，不会影响个人数据，只是删除了这个分区中已经保存的缓存内容，缓存内容会在后续手机使用过程中重新自动生成。

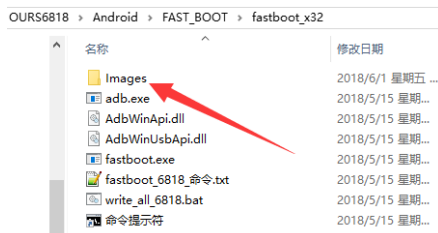
⑦misc 分区

被 recovery 使用的小型分区来保存一些信息，以防止当 OTA 包正在被应用的时候设备重启。

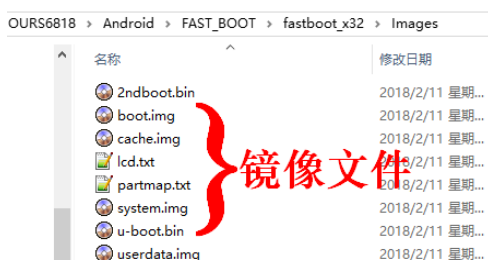
7.5.5 实验步骤

一、准备系统镜像

为了烧写方便，请将需要烧写的镜像文件拷贝到 fastboot 工具目录，在 fastboot 目录下创建 Images 目录，用于存放镜像文件（注意根据自己的系统选择不同的 fastboot 工具（32 位或 64 位））



将镜像文件拷贝至 fastboot 目录下的 Images 目录，需要的文件为 lcd.txt、partmap.txt、2ndboot.bin、u-boot.bin、boot.img、system.img、userdata.img、cache.img。



二、安装 fastboot 和 adb

首先，你需要在你的电脑上安装 ADB 和 Fastboot，只有有了它们你才能使用 Fastboot 命令刷入镜像。Adb 和 fastboot 可以从 Android SDK Tools 中获得，也可以从网络获得，如果情况允许建议使用 Android SDK Tools 获取方式，因为其更新比较快可完全兼容新版设备。笔者提供的工具位于“FASTBOOT”目录，其中

包含了 adb 和 fastboot。

三、安装 USB 驱动

由于 fastboot 使用 USB 数据线与目标板通信,所以需要安装目标板的 USB 驱动。要对目标板进行 fastboot 烧写时, 必须在电脑上安装该芯片的对应 USB 驱动。(目标板需要有 U-boot 程序,可以是已烧写到 Flash 的 u-boot, 也可以从 TF 卡引导)

①连接目标板

使用 MiniUSB 数据线连接 PC 机和目标板, 目标板 miniUSB 接口位于板子正下方 TF 卡座旁; 使用串口线连接目标板左上侧 DB9 接口 (debug 调试口) 到 PC 机; 使用 5V 电源给目标板供电。

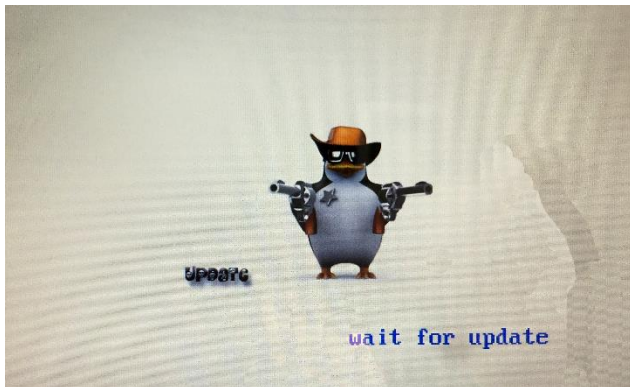
②进入 fastboot 模式

打开串口调试软件: 波特率 115200, 数据位 8, 停止位 1, 无奇偶校验, 无流控。目标板通电后, 在 uboot 启动倒计时内按下回车键, 输入 fastboot 并回车, 此时目标板即进入 fastboot 模式, 同时 LCD 屏将显示升级界面。

S5p6818# fastboot

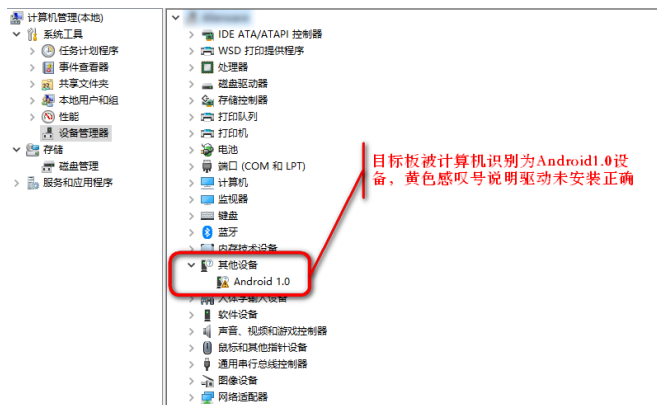
```
Hit Enter key or Space key to stop autoboot: 0
s5p6818# fastboot

Fastboot Partitions:
mmc.2: 2ndboot, img : 0x200, 0x7e00
mmc.2: bootloader, img : 0x8000, 0x77000
mmc.2: boot, fs : 0x100000, 0x4000000
mmc.2: system, fs : 0x4100000, 0x2f200000
mmc.2: cache, fs : 0x33300000, 0x1ac00000
mmc.2: misc, fs : 0x4e000000, 0x800000
mmc.2: recovery, fs : 0x4e900000, 0x1600000
mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part : partmap mem env cmd
DONE: Logo bmp 300 by 270 (3bpp), len=243054
DRAW: 0x47000000 -> 0x46000000
Load USB Driver: android
Core usb device tie configuration done
OTG cable Connected!
-----
```



③更新驱动

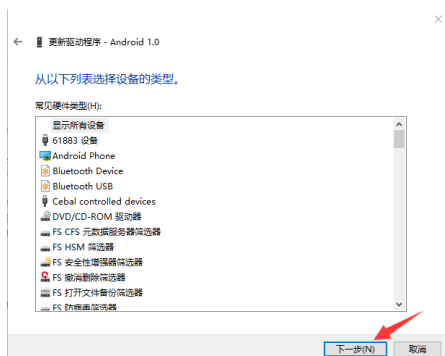
当目标板进入 fastboot 模式后, 打开 PC 的设备管理器, 将会出现新设备, 如下图

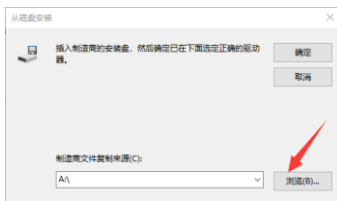


接下来安装驱动，对未知设备点击右键，选择【更新驱动程序软件】，再点击【浏览计算机查找驱动程序软件】（如下图）

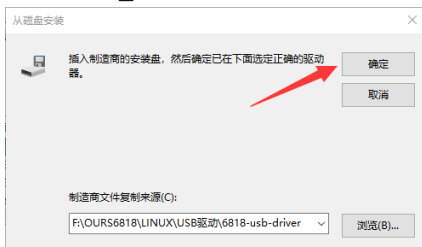


点击【让我从计算机上的可用驱动列表中读取】

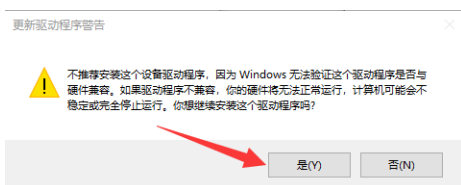




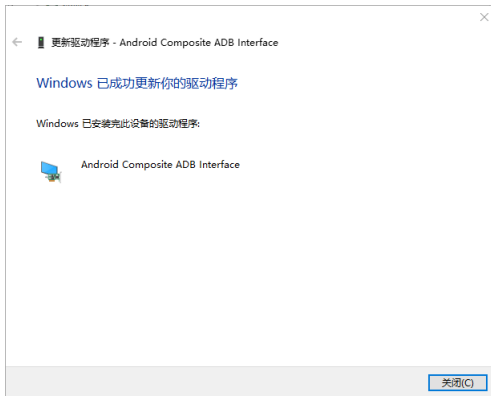
点击【浏览】，浏览到 USB 驱动对应的位置（例如：我的在“USB 驱动\6818-usb-driver 目录”），选择“android_winusb.inf”文件，点击【打开】，之后点击【确定】



选择“Android composite ADB Interface”，点击【下一步】，在弹出的警告窗口中，点击【是】



接下来在弹出警告窗口中选择【始终安装此驱动软件】

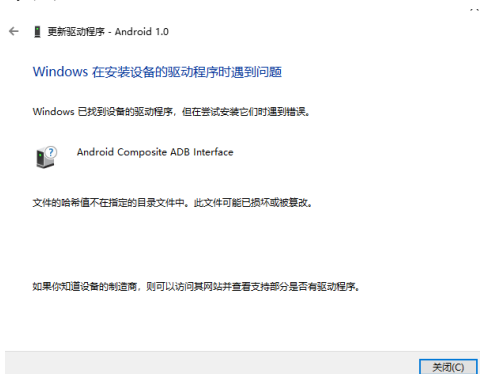


返回如上图所示，说明驱动安装完成，再次到设备管理器中查看设备



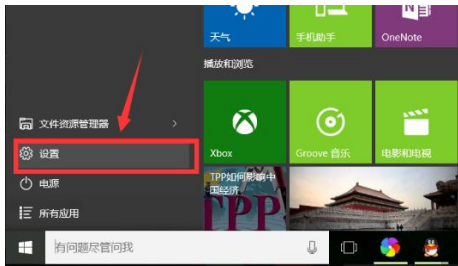
当设备管理器中出现如上图所示设备，说明驱动安装成功。至此，fastboot 方式下的设备 USB 驱动安装完成。

说明：绝大多数系统完成上述步骤即可完成驱动安装，如果使用 WIN10 系统，可能出现如下图中所示的错误信息。如若用户同样使用 WIN10 系统出现如下错误信息，请按照“WIN10 系统禁用驱动强制签名”步骤禁用。



附件：WIN10 系统禁用驱动强制签名：

第一步、点开【开始】菜单，点击里面的【设置】



第二步、在电脑设置界面，点击“更新和安全”。



第三步、在“更新和安全”界面，点击左侧【恢复】，点击右侧窗口高级启动项目下面的立即重新启动。



第四步、电脑重启后，在选择一个选项界面，点击疑难解答



第五步、在疑难解答界面，点击高级选项



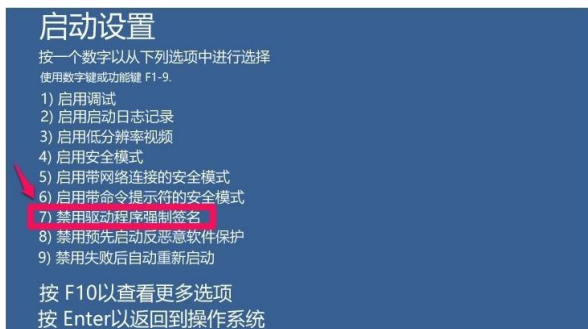
第六步、在高级选项界面，点击启动设置



第七步、在启动设置界面，点击重启。



第八步、重启，重新启动后进入启动设置窗口，按下键盘上的数字键【7】或者【F7】，即选择禁用驱动程序强制签名。



注意：Win8 以上系统务必按照上述说明禁用驱动签名，否则将无法安装驱动。

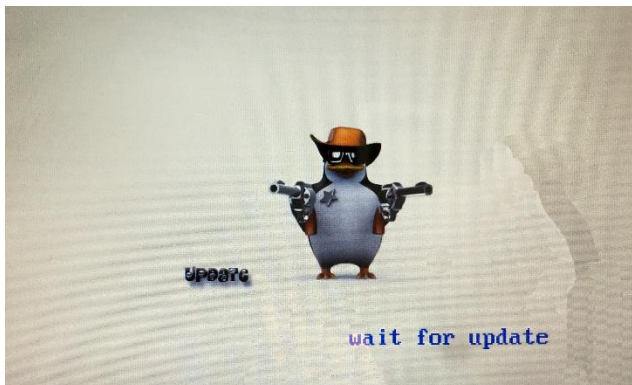
三、烧写镜像

使用 fastboot 烧写，分两种方式，一种为单文件烧写，一种为批量烧写，单文件烧写适用于更新单个文件；批量烧写适用于更新全部镜像；不同的方式适用于不同场景。

①单文件烧写

第一步：进入目标板 fastboot 模式

打开串口调试软件（波特率 115200,数据位 8,停止位 1，无奇偶校验，无流控），目标板通电后，在 uboot 启动倒计时内按下回车键,输入 fastboot 并回车，此时目标板即进入 fastboot 模式，同时 LCD 屏将显示升级界面。



```

Hit Enter key or Space key to stop autoboot: 0
s5p6818# fastboot

Fastboot Partitions:
mmc.2: 2ndboot, img : 0x200, 0x7e00
mmc.2: bootloader, img : 0x8000, 0x77000
mmc.2: boot, fs : 0x100000, 0x4000000
mmc.2: system, fs : 0x4100000, 0x2f200000
mmc.2: cache, fs : 0x33300000, 0x1ac00000
mmc.2: misc, fs : 0x4e000000, 0x800000
mmc.2: recovery, fs : 0x4e900000, 0x1600000
mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part : partmap mem env cmd
DONE: Logo bmp 300 by 270 (3bpp), len=243054
DRAW: 0x47000000 -> 0x46000000
Load USB Driver: android
Core usb device tie configuration done
OTG cable Connected!
-----

```

目标板进入Fastboot模式

第二步：进入 fastboot 工具目录

运行【命令提示符】快捷方式文件（或者先打开命令提示符 cmd，从命令行进入 fastboot 目录），打开命令行窗口。

```

C:\ 命令提示符
Microsoft Windows [版本 10.0.16299.1251]
(c) 2017 Microsoft Corporation。保留所有权利。

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>

```

第三步：烧写 partmap

在命令行中输入：**fastboot flash partmap Images/partmap.txt** 命令烧写分区表，详细烧写细节可查看目标板串口回显信息。（命令中的 Images/partmap.txt 代表烧写镜像是当前目录下的 Images 下的 partmap.txt 文件，如果用户用于存放镜像的目录在其他目录，需要修改路径及目录名，建议使用绝对路径）

```

C:\ 命令提示符
Microsoft Windows [版本 10.0.16299.1251]
(c) 2017 Microsoft Corporation。保留所有权利。

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash partmap Images/partmap.txt
sending 'partmap' (0 KB)... OKAY
writing 'partmap'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32> Windows端Fastboot

```

```

Starting download of 646 bytes
downloading 512 -- 0% complete.downloading of 646 bytes to 48000000 (0xc8000000) finished
Flash : partmap

Fastboot Partitions:
mmc.2: 2ndboot, img : 0x200, 0x7e00
mmc.2: bootloader, img : 0x8000, 0x77000
mmc.2: boot, fs : 0x100000, 0x4000000
mmc.2: system, fs : 0x4100000, 0x2f200000
mmc.2: cache, fs : 0x33300000, 0x1ac00000
mmc.2: misc, fs : 0x4e000000, 0x800000
mmc.2: recovery, fs : 0x4e900000, 0x1600000
mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part : partmap mem env cmd
fdisk 2 6: 0x100000:0x4000000 0x4100000:0x2f200000 0x33300000:0x1ac00000 0x4e000000:0x4e900000 0x4e900000:0x1600000 0x50000000:0x0
Writing to MMC(2)... done
Flash : partmap - DONE

```

目标板串口回显

第四步：烧写 2ndboot

在命令行中输入：**fastboot flash 2ndboot Images/2ndboot.bin** 命令烧写 2ndboot

```

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash 2ndboot Images/2ndboot.bin
sending '2ndboot' (30 KB)... OKAY
writing '2ndboot'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32> Fastboot烧写2ndboot

```

```
Starting download of 30972 bytes
downloading 30720 -- 97% complete.
downloading of 30972 bytes to 48000000 (0xc8000000) finished
Flash : 2ndboot
switch to partitions #0, OK
mmc2(part 0) is current device
switch to partitions #0, OK
mmc2(part 0) is current device
head boot dev = 2
update mmc.2 type 2ndboot = 0x200(0x1) ~ 0x78fc(0x3d): Done
Flash : 2ndboot - DONE
```

烧写2ndboot
目标板回显信息

第五步：烧写 u-boot

在命令行中输入：`fastboot flash bootloader Images/u-boot.bin` 命令烧写 u-boot

```
F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash bootloader Images/u-boot.bin
sending 'bootloader' (261 KB)... OKAY
writing 'bootloader'... OKAY
```

Fastboot烧写u-boot

```
Starting download of 268228 bytes
downloading 266240 -- 99% complete.
downloading of 268228 bytes to 48000000 (0xc8000000) finished
Flash : bootloader
switch to partitions #0, OK
mmc2(part 0) is current device
switch to partitions #0, OK
mmc2(part 0) is current device
head boot dev = 2
head load addr = 0x42c00000
head load size = 0x000417c4
head signature = 0x4849534e
update mmc.2 type boot = 0x8000(0x40) ~ 0x419c4(0x20d): Done
Flash : bootloader - DONE
```

烧写U-boot目
标板回显信息

第六步：烧写 kernel

在命令行中输入：`fastboot flash boot Images/boot.img` 命令烧写 Android 内核镜像

```
F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash boot Images/boot.img
sending 'boot' (22300 KB)... OKAY
writing 'boot'... OKAY
```

Fastboot烧写内核

```
Starting download of 22835432 bytes
downloading 22607872 -- 99% complete.
downloading of 22835432 bytes to 48000000 (0xc8000000) finished
Flash : boot
switch to partitions #0, OK
mmc2(part 0) is current device
Flash : boot - DONE
```

烧写boot目标板回显

第七步：烧写 Android 系统

在命令行中输入：`fastboot flash system Images/system.img` 命令烧写 Android 系统镜像，由于文件系统较大，烧写过程稍微长一点，可在目标板串口回显中查看烧写进度。

```
F:\OURS6818\Linux\FAST_BOOT\fastboot_32bit>fastboot flash system Images/system.img
sending 'system' (2000000 KB)... OKAY
writing 'system'... OKAY
```

Windows端fastboot烧写system执行窗口

```
Starting download of 204800000 bytes
downloading 202752512 -- 99% complete.
downloading of 204800000 bytes to 48000000 (0xc8000000) finished
Flash : system
switch to partitions #0, OK
mmc2(part 0) is current device
write image to 0x4100000(0x20800), 0xc350000(0x20800)
Flash : system - DONE
```

目标板烧写system回显

第八步：烧写 cache 镜像

在命令行中输入：`fastboot flash system Images/cache.img` 命令烧写 cache 镜像。

```
命令提示符
Microsoft Windows [版本 10.0.16299.125]
(c) 2017 Microsoft Corporation. 保留所有权利。

F:\OURS6818\Android\FAST_BOOT\Fastboot_x32>fastboot flash cache Images/cache.img
sending 'cache' (8752 KB)... OKAY
writing 'cache'... OKAY

F:\OURS6818\Android\FAST_BOOT\Fastboot_x32> Fastboot烧写cache
```

```
Starting download of 8962352 bytes
downloading 8873472 -- 99% complete.
downloading of 8962352 bytes to 48000000 (0xc8000000) finished
Flash : cache
switch to partitions #0, OK
mmc2(part 0) is current device
Flash : cache - DONE

} 烧写cache目标板回显
```

第九步：烧写 userdata 系统

在命令行中输入：**fastboot flash system Images/userdata.img** 命令烧写 userdata 镜像。

```
F:\OURS6818\Android\FAST_BOOT\Fastboot_x32>fastboot flash userdata Images/userdata.img
sending 'userdata' (137614 KB)... OKAY
writing 'userdata'... OKAY

F:\OURS6818\Android\FAST_BOOT\Fastboot_x32> Fastboot烧写userdata
```

```
Starting download of 140917740 bytes
downloading 139509248 -- 99% complete.
downloading of 140917740 bytes to 48000000 (0xc8000000) finished
Flash : userdata
switch to partitions #0, OK
mmc2(part 0) is current device
Flash : userdata - DONE

} 烧写userdata目标板回显
```

完成以上步骤即完成 Fastboot 镜像烧写，之后便可重启系统，系统将执行新更新的镜像文件。可以通过在目标板串口终端中按下 CTRL+C 键退出 fastboot 模式，在 u-boot 命令行输入 boot 启动，也可以直接通过电源开关重新启动目标板。

```
Fastboot ended by user
DONE: Logo bmp 1024 by 600 (3bpp), len=1843254
DRAW: 0x47000000 -> 0x46000000
s5p6818# boot
## Booting kernel from Legacy Image at 48000000 ...
Image Name: Linux-3.4.39
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 4579976 Bytes = 4.4 MiB
Load Address: 40008000
Entry Point: 40008000
Verifying Checksum ... OK
Loading Kernel Image ... OK

按下Ctrl+C返回
U-boot交互窗口

输入boot启动系统
```

注意：在使用 fastboot 烧写过程中，如果长时间没有操作，再次烧写可能没有响应，此时需要重启目标板，并重新进入 fastboot 模式。重启不会影响之前已完成的烧写，可以接着上次步骤继续。

②批量烧写

fastboot 批量烧写与单文件烧写没有本质区别，只是方便更新镜像将逐条命令放在一个脚本中。当目标板进入 fastboot 模式后，进入 windows 系统下 fastboot 目录，直接双击运行 write_all_6818.bat 脚本。

```
Hit Enter key or Space key to stop autoboot: 0
s5p6818# fastboot

Fastboot Partitions:
mmc.2: 2ndboot, img : 0x200, 0x7e00
mmc.2: bootloader, img : 0x8000, 0x77000
mmc.2: boot, fs : 0x100000, 0x4000000
mmc.2: system, fs : 0x4100000, 0x2f200000
mmc.2: cache, fs : 0x33300000, 0x1ac00000
mmc.2: misc, fs : 0x4e000000, 0x8000000
mmc.2: recovery, fs : 0x4e900000, 0x1600000
mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part : partmap mem env cmd
DONE: Logo bmp 300 by 270 (3bpp), len=243054
DRAW: 0x47000000 -> 0x46000000
Load USB Driver: android
Core usb device tie configuration done
OTG cable Connected!

目标板进入Fastboot模式
```



```
F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash partmap Images/partmap.txt
sending 'partmap' (0 KB)... OKAY
writing 'partmap'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash 2ndboot Images/2ndboot.bin
sending '2ndboot' (30 KB)... OKAY
writing '2ndboot'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash bootloader Images/u-boot.bin
sending 'bootloader' (261 KB)... OKAY
writing 'bootloader'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash boot Images/boot.img
sending 'boot' (22300 KB)... OKAY
writing 'boot'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash system Images/system.img
sending 'system' (515429 KB)...
```

Fastboot批量烧写Android

升级完成后，批处理文件会自动关闭，升级过程可以到串口工具里可以看到，烧写完成后，系统自动重启。

说明：在批处理最后加入了 fastboot reboot 命令控制系统重启，用户可自行编辑批处理文件，删除此命令或者添加其他命令以完成更多功能。

7.5.6 实验总结

至此 fastboot 烧写已全部完成。本节 fastboot 烧写需要熟练掌握，目前绝大多数的手机平板都是采用此方法，此方法具有很强通用性。

7.6 Android 系统 OTA 升级

7.6.1 实验目的

掌握系统 OTA 升级方法。

7.6.2 实验内容

利用 OTA 进行系统镜像升级。

7.6.3 实验设备

硬件：PC 机 1 台、OURS6818 实验平台 1 套、串口线一根、miniUSB 数据线 1 根

软件：adb 软件、USB 驱动、OTA 镜像；串口调试软件（minicom 或 putty 均可）

7.6.4 实验原理

一、Android OTA 更新

Android 设备可以接受和安装系统和应用程序的空中下载更新。设备有一个特殊的带有软件的 recovery 分区，该分区可以解压下载的更新包并且将他们应用到系统中。

OTA 更新被设计用于设计底层的操作系统和安装在系统分区的只读 app；这些更新不会影响来自应用市场的用户安装的应用程序。

Android 系统进行升级的时候，有两种途径，一种是通过接口传递升级包路径自动升级（Android 系统 SD 卡升级），升级完之后系统自动重启；另一种是手动进入 recovery 模式下，选择升级包进行升级，升级完成之后停留在 recovery 界面，需要手动选择重启。前者多用于手机厂商的客户端在线升级，后者多用于开发和测试人员。但不管哪种，原理都是一样的，都要在 recovery 模式下进行升级。

1、OTA 更新包含的步骤

一个典型的 OTA 更新包含下面的步骤：

- 设备使用 OTA 服务运行常规的检查，并且被可用更新通知，包括更新包的 URL 和展示给用户的一个描述字符串。

更新下载到一个 cache 或者是 data 分区，并且他的加密签名和位于/system/etc/security/otacerts.zip 的证书进行验证。用户被推送来安装更新

设备重启进入 recovery 模式，在这个模式中，位于 recovery 分区的内核和系统被启动，而不是位于 boot 分区内核启动

Recovery 二进制文件被 init 启动。他寻找位于/cache/recovery/command 中的命令行参数，该参数执行下载的包。

Recovery 通过位于/res/keys（位于 recovery 分区的一部分 RAM）下的公共钥匙来验证包的加密签名。

数据从包中提取出来并且用于去更新需要的 boot,system 和 vendor 分区。位于 system 分区的新文件之一包含新的 recovery 分区的内容

设备正常重启。

- a. 新更新的 boot 分区被加载，他挂载和开始执行位于新的更新的 system 分区中的二进制文件
- b. 作为正常启动的一部分，系统检查 recovery 分区的内容和需要的内容做对比（之前以文件的形式保存在 /system 中）。他们是不同的，所以 recovery 分区被需要的内容写入。（在随后的启动中，recovery 分区已经包含新的内容，所以不需要重新写入了。）

这一小节描述的是 Android5.x 发行版的 OTA 系统升级。最后我会将其他版本的 OTA 相关的代码附上。

二、Android 常见分区

Android 系统使用几个分区来管理文件和文件夹。每个分区在设备上都有不同的功能，Android 系统的内部存储分区主要包含如下几个：

- bootloader 分区
- boot 分区
- recovery 分区
- system 分区
- userdata 分区
- cache 分区
- misc 分区

①bootloader 分区

Bootloader 分区存放 Android 系统 Bootloader 镜像，其分为主引导和二级引导。主引导代码主要执行硬件检测，确保硬件能正常工作，然后将二级引导代码拷贝到内存(RAM)开始执行。二级引导代码进行一些硬件初始化工作，获取内存大小信息等，然后根据用户的交互进入到某种启动模式。比如说正常启动模式、recovery 模式、fastboot 模式等。fastboot 模式：fastboot 是 android 定义的一种简单的刷机协议，用户可以通过 fastboot 命令行工具来进行刷机。

②boot 分区

Boot 分区存放有 Android 的内核和内存操作程序。没有这个分区设备就不能启动。恢复系统的时候会擦除这个分区，并且必须重新安装 ROM 才能重启系统。当系统直接上电时，会进入正常启动模式，二级引

导程序会跳转到 boot 分区开始启动。Boot 分区的格式是固定的，首先是一个头部，然后是 LINUX 内核，最后是根文件系统的 ramdisk。当 LINUX 内核启动完毕后，就开始执行根文件系统中的 init 程序，init 程序会读取启动脚本文件 init.rc 和 init.xxx.rc。

android 内核挂载文件系统后，根据 init.rc、init.s5p6818_drone.rc 来初始化并装载系统库、程序等直到开机完成。init.rc 脚本包括了文件系统初始化、装载的许多过程。init.rc 的工作主要是：

- 1) 设置一些环境变量
- 2) 创建 system、sdcard、data、cache 等目录
- 3) 把一些文件系统 mount 到一些目录去，如，mount tmpfs tmpfs /sqlite_stmt_journals
- 4) 设置一些文件的用户群组、权限
- 5) 设置一些线程参数
- 6) 设置 TCP 缓存大小

init 程序读取启动脚本，执行脚本中指定的动作和命令，脚本中的一部分是运行 system 分区的程序。

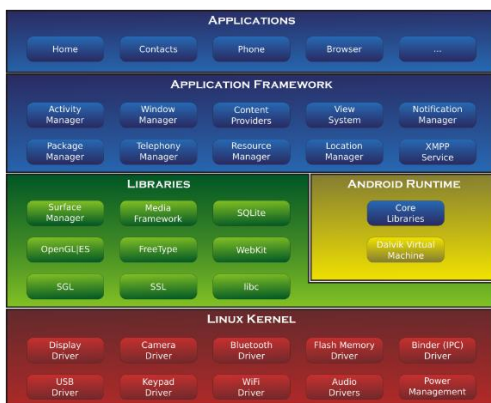
③recovery 分区

recovery 分区即恢复分区，在正常分区被破坏后，仍可以进入这一分区进行备份和恢复。这个分区保存一个简单的 OS 或底层软件，在 Android 的内核被破坏后可以用 bootloader 从这个分区引导进行操作。这个分区可以认为是一个 boot 分区的替代品，可以使你的设备进入 Recovery 程序，进行高级恢复或安卓系统维护工作。

recovery 模式：recovery 是 android 定义的一个标准刷机协议。当进入 recovery 模式时，二级引导程序从 recovery 分区开始启动，recovery 分区实际上是一个简单的 Linux 系统，当内核启动完毕后，开始执行第一个程序 init(init 程序是 Linux 系统所有程序的老祖宗)。init 会启动一个叫做 recovery 的程序（recovery 模式的名称也由此而来）。通过 recovery 程序，用户可以执行清除数据，安装刷机包等操作。一般的手机厂商都提供一个简单的 recovery 刷机，大多数只能进行 upate 的操作。

④system 分区

System 分区存放 Android 系统。这个分区基本包含了整个安卓操作系统，除了内核(kerne)和 ramdisk。里面包含了 Android 用户接口、用户界面、和所有预装的系统应用程序。擦除了这个分区，会删除整个安卓系统，你可以通过进入 Recovery 程序或者 bootloader 程序中，安装一个新 ROM，也就是新安卓系统。



除上图中的 linux Kernel 部分位于 boot 分区外，其他的 Library、runtime、framework、core application 都是处于 system 分区。

- /system/priv-app：特权 App，比 system_app 权限还要高，其不仅 System_app 标识是 true，同时还置了 Priv-app 标识。
- /system/app：核心应用程序 (*.apk)，都是放在这。像是 Phone、Alarm Clock, Browser, Contacts 等等。
- /system/framework：这里放 Android 系统的核心程式库，就是上图中 application framework 部分的库。像是 core.jar, framework-res.apk, framework.jar 等等。
- system/lib：上图中 Library 部分，存放的是所有动态链接库(.so 文件)，这些 SO 是 JNI 层，Dalvik 虚拟

机，本地库，HAL 层所需要的，因为系统应用/system/app 下的 apk 是不会解压 SO 到程序的目录下，所以其相应用的 SO，都应放在/system/lib 下面。当一个系统 apk 的 SO 加载时，会从此目录下寻找对应用的 SO 文件；

- /system/media/audio/(notification, alarms, ringtones, ui)：这里放系统的声音文件，像是闹铃声，来电铃声等等。这些声音文件，多是 ogg 格式。
- /system/bin: 存放的是一些可执行文件，基本上是由 C/C++编写的。其中有一个重要的命令叫 app_process。一般大家称之为 Zygote。(Zygote 是卵的意思，所有的 Android 进程都是由它生出来的)。Zygote 首先会加载 dalvik 虚拟机，然后产生一个叫做 system_server 的进程。system_server 顾名思义被称作 Android 的系统服务程序，它主要管理整个 android 系统。system_server 启动完成后开始寻找一个叫做启动器的程序，找到之后由 zygote 开始启动执行启动器，这就是我们常见到的桌面程序。
- system/sbin: 存放的是一些扩展的可执行文件，该目录可以为空。大家常用的 busybox 就放在该目录下。Busybox 所建立的各种符号链接命令都是放在该目录。
- system/build.prop: build.prop 和上节说得根文件系统中的 default.prop 文件格式一样，都称为属性配置文件。它们都定义了一些属性值，代码可以读取或者修改这些属性值。属性值有一些命名规范：ro 开头的表示只读属性，即这些属性的值代码是无法修改的。persist 开头的表示这些属性值会保存在文件中，这样重新启动之后这些值还保留。其它的属性一般以所属的类别开头，这些属性是可读可写的，但是对它们的修改重启之后不会保留。
- system/etc: 目录存放一些配置文件，和属性配置文件不一样，这下面的配置文件可能稍微没那么有规律。一般来说，一些脚本程序，还有大家所熟悉 GPS 配置文件(gps.conf)和 APN 配置文件(apns-conf.xml)放在这个目录。

⑤userdata 分区

它将挂载到 /data 目录下，它是由编译出来的 userdata.img 来烧入。这个分区也叫用户数据区，包含了用户的数据：联系人、短信、设置、用户安装的程序。擦除这个分区，本质上等同于手机恢复出厂设置，也就是手机系统第一次启动时的状态，或者是最后一次安装官方或第三方 ROM 后的状态。在 Recovery 程序中进行的“data/factory reset”操作就是在擦除这个分区。

⑥cache 分区

它将挂载到 /cache 目录下。这个分区是安卓系统缓存区，保存系统最常访问的数据和应用程序。擦除这个分区，不会影响个人数据，只是删除了这个分区中已经保存的缓存内容，缓存内容会在后续手机使用过程中重新自动生成。

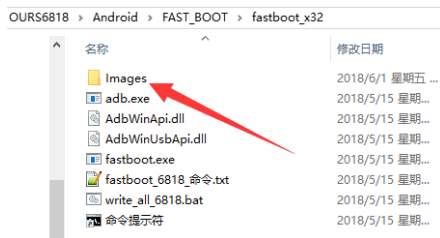
⑦misc 分区

这个分区包括了一些杂项内容：比如一些系统设置和系统功能启用禁用设置。这些设置包括 CID(运营商或区域识别码)、USB 设置和一些硬件设置等等。如果此分区损坏或者部分数据丢失，手机的一些特定功能可能不能正常工作。

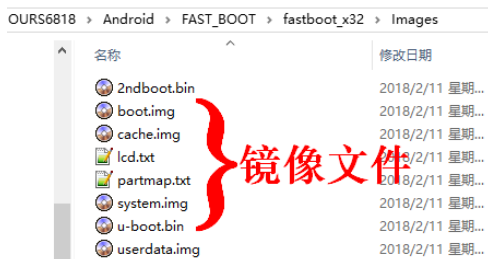
7.6.5 实验步骤

一、准备系统镜像

为了烧写方便，请将需要烧写的镜像文件拷贝到 fastboot 工具目录，在 fastboot 目录下创建 Images 目录，用于存放镜像文件（注意根据自己的系统选择不同的 fastboot 工具（32 位或 64 位））



将镜像文件拷贝至 fastboot 目录下的 Images 目录，需要的文件为 lcd.txt、partmap.txt、2ndboot.bin、u-boot.bin、boot.img、system.img、userdata.img、cache.img。



二、安装 fastboot 和 adb

首先，你需要在你的电脑上安装 ADB 和 Fastboot，只有有了它们你才能使用 Fastboot 命令刷入镜像。Adb 和 fastboot 可以从 Android SDK Tools 中获得，也可以从网络获得，如果情况允许建议使用 Android SDK Tools 获取方式，因为其更新比较快可完全兼容新版设备。笔者提供的工具位于“FASTBOOT”目录，其中包含了 adb 和 fastboot。

三、安装 USB 驱动

由于 fastboot 使用 USB 数据线与目标板通信，所以需要安装目标板的 USB 驱动。要对目标板进行 fastboot 烧写时，必须在电脑上安装该芯片的对应 USB 驱动。（目标板需要有 U-boot 程序，可以是已烧写到 Flash 的 u-boot，也可以从 TF 卡引导）

①连接目标板

使用 MiniUSB 数据线连接 PC 机和目标板，目标板 miniUSB 接口位于板子正下方 TF 卡座旁；使用串口线连接目标板左上侧 DB9 接口（debug 调试口）到 PC 机；使用 5V 电源给目标板供电。

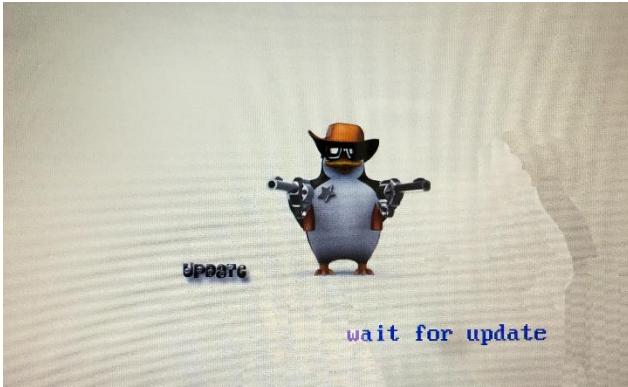
②进入 fastboot 模式

打开串口调试软件：波特率 115200，数据位 8，停止位 1，无奇偶校验，无流控。目标板通电后，在 uboot 启动倒计时内按下回车键，输入 fastboot 并回车，此时目标板即进入 fastboot 模式，同时 LCD 屏将显示升级界面。

S5p6818# fastboot

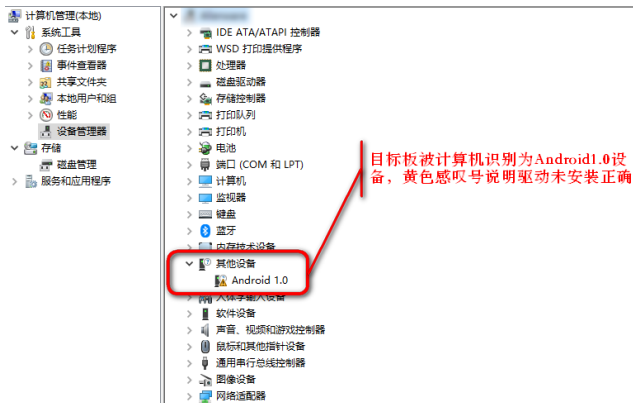
```
Hit Enter key or Space key to stop autoboot: 0
s5p6818# fastboot

Fastboot Partitions:
mmc.2: 2ndboot, img : 0x200, 0x7e00
mmc.2: bootloader, img : 0x8000, 0x77000
mmc.2: boot, fs : 0x100000, 0x4000000
mmc.2: system, fs : 0x4100000, 0x2f200000
mmc.2: cache, fs : 0x33300000, 0x1ac00000
mmc.2: misc, fs : 0x4e000000, 0x800000
mmc.2: recovery, fs : 0x4e900000, 0x1600000
mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part : partmap mem env cmd
DONE: Logo bmp 300 by 270 (3bpp), len=243054
DRAW: 0x47000000 -> 0x46000000
Load USB Driver: android
Core usb device tie configuration done
OTG cable Connected!
```



③更新驱动

当目标板进入 fastboot 模式后，打开 PC 的设备管理器，将会出现新设备，如下图

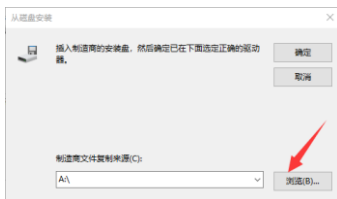
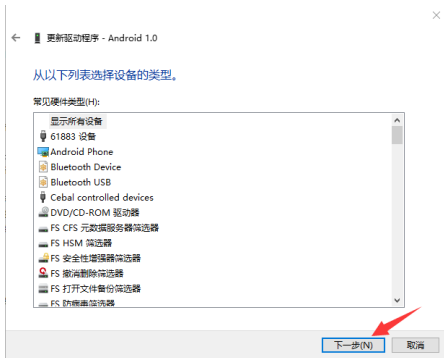


接下来安装驱动，对未知设备点击右键，选择【更新驱动程序软件】，再点击【浏览计算机查找驱动程序软件】（如下图）

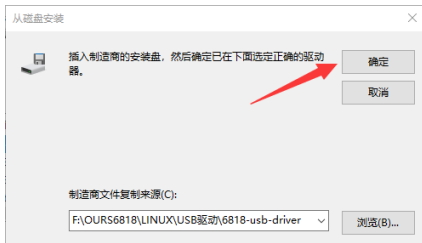


点击【让我从计算机上的可用驱动列表中读取】



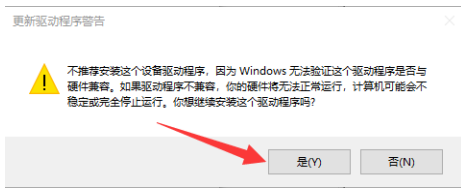


点击【浏览】，浏览到 USB 驱动对应的位置（例如：我的在“USB 驱动\6818-usb-driver 目录”），选择“android_winusb.inf”文件，点击【打开】，之后点击【确定】

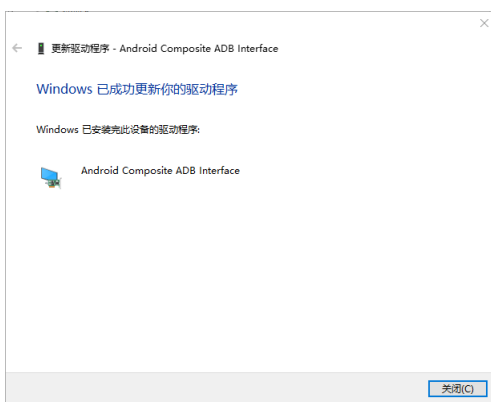


选择“Android composite ADB Interface”，点击【下一步】，在弹出的警告窗口中，点击【是】

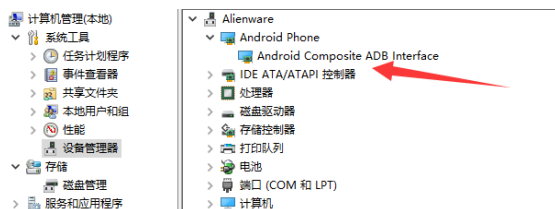




接下来在弹出警告窗口中选择【始终安装此驱动软件】



返回如上图所示，说明驱动安装完成，再次到设备管理器中查看设备



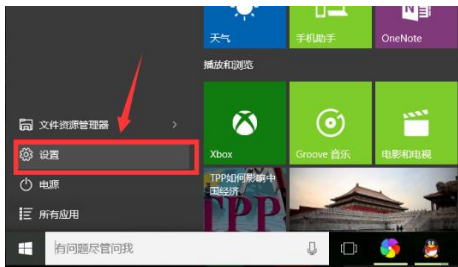
当设备管理器中出现如上图所示设备，说明驱动安装成功。至此，fastboot 方式下的设备 USB 驱动安装完成。

说明：绝大多数系统完成上述步骤即可完成驱动安装，如果使用 WIN10 系统，可能出现如下图中所示的错误信息。如若用户同样使用 WIN10 系统出现如下错误信息，请按照“WIN10 系统禁用驱动强制签名”步骤禁用。



附件：WIN10 系统禁用驱动强制签名：

第一步、点开【开始】菜单，点击里面的【设置】



第二步、在电脑设置界面，点击“更新和安全”。



第三步、在“更新和安全”界面，点击左侧【恢复】，点击右侧窗口高级启动项目下面的立即重新启动。



第四步、电脑重启后，在选择一个选项界面，点击疑难解答



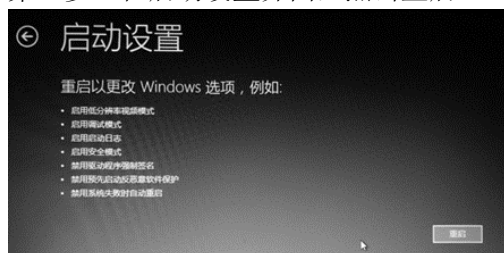
第五步、在疑难解答界面，点击高级选项



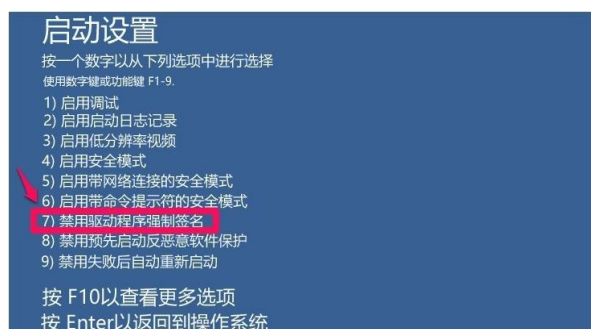
第六步、在高级选项界面，点击启动设置



第七步、在启动设置界面，点击重启。



第八步、重启，重新启动后进入启动设置窗口，按下键盘上的数字键【7】或者【F7】，即选择禁用驱动程序强制签名。



注意：Win8 以上系统务必按照上述说明禁用驱动签名，否则将无法安装驱动。

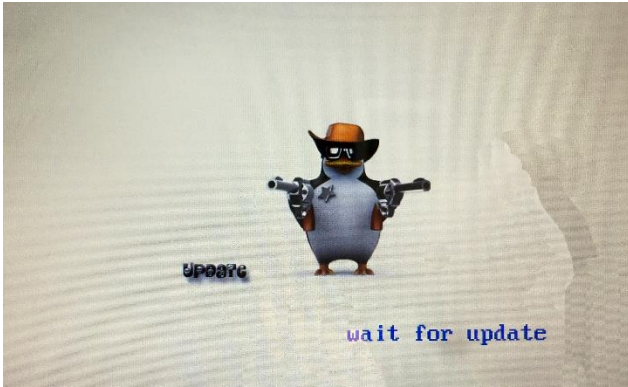
三、烧写镜像

使用 fastboot 烧写，分两种方式，一种为单文件烧写，一种为批量烧写，单文件烧写适用于更新单个文件；批量烧写适用于更新全部镜像；不同的方式适用于不同场景。

①单文件烧写

第一步：进入目标板 fastboot 模式

打开串口调试软件（波特率 115200,数据位 8,停止位 1，无奇偶校验，无流控），目标板通电后，在 uboot 启动倒计时内按下回车键,输入 fastboot 并回车，此时目标板即进入 fastboot 模式，同时 LCD 屏将显示升级界面。



```
Hit Enter key or Space key to stop autoboot: 0
s5p6818# fastboot

Fastboot Partitions:
mmc.2: 2ndboot, img : 0x200, 0x7e00
mmc.2: bootloader, img : 0x8000, 0x77000
mmc.2: boot, fs : 0x100000, 0x4000000
mmc.2: system, fs : 0x4100000, 0x2f200000
mmc.2: cache, fs : 0x33300000, 0x1ac00000
mmc.2: misc, fs : 0x4e000000, 0x800000
mmc.2: recovery, fs : 0x4e900000, 0x1600000
mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part : partmap mem env cmd
DONE: Logo bmp 300 by 270 (3bpp), len=243054
DRAW: 0x47000000 -> 0x46000000
Load USB Driver: android
Core usb device tie configuration done
OTG cable Connected!
=====
```

目标板进入Fastboot模式

第二步：进入 fastboot 工具目录

运行【命令提示符】快捷方式文件（或者先打开命令提示符 cmd，从命令行进入 fastboot 目录），打开命令行窗口。

```
命令提示符
Microsoft Windows [版本 10.0.16299.125]
(c) 2017 Microsoft Corporation。保留所有权利。

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>
```

第三步：烧写 partmap

在命令行中输入：**fastboot flash partmap Images/partmap.txt** 命令烧写分区表，详细烧写细节可查看目标板串口回显信息。（命令中的 Images/partmap.txt 代表烧写镜像是当前目录下的 Images 下的 partmap.txt 文件，如果用户用于存放镜像的目录在其他目录，需要修改路径及目录名，建议使用绝对路径）

```
命令提示符
Microsoft Windows [版本 10.0.16299.125]
(c) 2017 Microsoft Corporation。保留所有权利。

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash partmap Images/partmap.txt
sending 'partmap' (0 KB)... OKAY
writing 'partmap'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32> Windows端Fastboot
```

```

Starting download of 646 bytes
downloading 512 -- 0% complete, downloading of 646 bytes to 48000000 (0xc8000000) finished
Flash : partmap

Fastboot Partitions:
mmc.2: 2ndboot, img : 0x200, 0x7e00
mmc.2: bootloader, img : 0x8000, 0x77000
mmc.2: boot, fs : 0x100000, 0x4000000
mmc.2: system, fs : 0x4100000, 0x2f200000
mmc.2: cache, fs : 0x33300000, 0x1ac00000
mmc.2: misc, fs : 0x4e000000, 0x8000000
mmc.2: recovery, fs : 0x4e900000, 0x16000000
mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part : partmap mem env cmd
disk 2 6: 0x100000:0x4000000 0x4100000:0x2f200000 0x33300000:0x1ac00000 0x4e00000:0x800000 0x4e900000:0x1600000 0x50000000:0x0
Writing to MMC(2)... done
Flash : partmap - DONE

```

} 目标板串口回显

第四步：烧写 2ndboot

在命令行中输入：**fastboot flash 2ndboot Images/2ndboot.bin** 命令烧写 2ndboot

```

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash 2ndboot Images/2ndboot.bin
sending '2ndboot' (30 KB)... OKAY
writing '2ndboot'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>

```

Fastboot烧写2ndboot

```

Starting download of 30972 bytes
downloading 30720 -- 97% complete.
downloading of 30972 bytes to 48000000 (0xc8000000) finished
Flash : 2ndboot
switch to partitions #0, OK
mmc2(part 0) is current device
switch to partitions #0, OK
mmc2(part 0) is current device
head boot dev = 2
update mmc.2 type 2ndboot = 0x200(0x1) ~ 0x78fc(0x3d): Done
Flash : 2ndboot - DONE

```

} 烧写2ndboot
目标板回显信息

第五步：烧写 u-boot

在命令行中输入：**fastboot flash bootloader Images/u-boot.bin** 命令烧写 u-boot

```

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash bootloader Images/u-boot.bin
sending 'bootloader' (261 KB)... OKAY
writing 'bootloader'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>

```

Fastboot烧写u-boot

```

Starting download of 268228 bytes
downloading 266240 -- 99% complete.
downloading of 268228 bytes to 48000000 (0xc8000000) finished
Flash : bootloader
switch to partitions #0, OK
mmc2(part 0) is current device
switch to partitions #0, OK
mmc2(part 0) is current device
head boot dev = 2
head load addr = 0x42c00000
head load size = 0x000417c4
head signature = 0x4849534e
update mmc.2 type boot = 0x8000(0x40) ~ 0x419c4(0x20d): Done
Flash : bootloader - DONE

```

} 烧写U-boot目
标板回显信息

第六步：烧写 kernel

在命令行中输入：**fastboot flash boot Images/boot.img** 命令烧写 Android 内核镜像

```

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash boot Images/boot.img
sending 'boot' (22300 KB)... OKAY
writing 'boot'... OKAY

F:\OURS6818\Android\FAST_BOOT\fastboot_x32>

```

Fastboot烧写内核

```

Starting download of 22835432 bytes
downloading 22607872 -- 99% complete.
downloading of 22835432 bytes to 48000000 (0xc8000000) finished
Flash : boot
switch to partitions #0, OK
mmc2(part 0) is current device
Flash : boot - DONE

```

} 烧写boot目标板回显

第七步：烧写 Android 系统

在命令行中输入：`fastboot flash system Images/system.img` 命令烧写 Android 系统镜像，由于文件系统较大，烧写过程稍微长一点，可在目标板串口回显中查看烧写进度。

```
F:\OURS6818\Linux\Fast_Boot\Fastboot_32bit>fastboot flash system Images/system.img
sending 'system' <2000000 KB>... OKAY
writing 'system'... OKAY
F:\OURS6818\Linux\Fast_Boot\Fastboot_32bit>
```

Windows端fastboot烧写system执行窗口

```
Starting download of 204800000 bytes
downloading 202752512 -- 99% complete.
downloading of 204800000 bytes to 48000000 (0xc8000000) finished
Flash : system
switch to partitions #0, OK
mmc2(part 0) is current device
write image to 0x4100000 (0x20800), 0xc350000 (0x20800)
Flash : system - DONE
```

目标板烧写system回显

第八步：烧写 cache 镜像

在命令行中输入：`fastboot flash system Images/cache.img` 命令烧写 cache 镜像。

```
命令提示符
Microsoft Windows [版本 10.0.16299.125]
(c) 2017 Microsoft Corporation. 保留所有权利。

F:\OURS6818\Android\Fast_Boot\Fastboot_x32>fastboot flash cache Images/cache.img
sending 'cache' <8752 KB>... OKAY
writing 'cache'... OKAY
F:\OURS6818\Android\Fast_Boot\Fastboot_x32>
```

Fastboot烧写cache

```
Starting download of 8962352 bytes
downloading 8873472 -- 99% complete.
downloading of 8962352 bytes to 48000000 (0xc8000000) finished
Flash : cache
switch to partitions #0, OK
mmc2(part 0) is current device
Flash : cache - DONE
```

烧写cache目标板回显

第九步：烧写 userdata 系统

在命令行中输入：`fastboot flash system Images/userdata.img` 命令烧写 userdata 镜像。

```
F:\OURS6818\Android\Fast_Boot\Fastboot_x32>fastboot flash userdata Images/userdata.img
sending 'userdata' <137614 KB>... OKAY
writing 'userdata'... OKAY
F:\OURS6818\Android\Fast_Boot\Fastboot_x32>
```

Fastboot烧写userdata

```
Starting download of 140917740 bytes
downloading 139509248 -- 99% complete.
downloading of 140917740 bytes to 48000000 (0xc8000000) finished
Flash : userdata
switch to partitions #0, OK
mmc2(part 0) is current device
Flash : userdata - DONE
```

烧写userdata目标板回显

完成以上步骤即完成 Fastboot 镜像烧写，之后便可重启系统，系统将执行新更新的镜像文件。可以通过在目标板串口终端中按下 CTRL+C 键退出 fastboot 模式，在 u-boot 命令行输入 boot 启动，也可以直接通过电源开关重新启动目标板。

```
Fastboot ended by user
DONE: Logo bmp 1024 by 600 (3bpp), len=1843254
DRAW: 0x47000000 -> 0x46000000
s5p6818# boot
## Booting kernel from Legacy Image at 48000000 ...
Image Name: Linux-3.4.39
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 4579976 Bytes = 4.4 MiB
Load Address: 40008000
Entry Point: 40008000
Verifying Checksum ... OK
Loading Kernel Image ... OK
```

按下Ctrl+C返回U-boot交互窗口

输入boot启动系统

注意：在使用 fastboot 烧写过程中，如果长时间没有操作，再次烧写可能没有响应，此时需要重启目标板，并重新进入 fastboot 模式。重启不会影响之前已完成的烧写，可以接着上次步骤继续。

②批量烧写

fastboot 批量烧写与单文件烧写没有本质区别，只是方便更新镜像将逐条命令放在一个脚本中。当目标板进入 fastboot 模式后，进入 windows 系统下 fastboot 目录，直接双击运行 write_all_6818.bat 脚本。

```
Hit Enter key or Space key to stop autoboot: 0
s5p6818# fastboot
Fastboot Partitions:
mmc.2: 2ndboot, img : 0x200, 0x7e00
mmc.2: bootloader, img : 0x8000, 0x77000
mmc.2: boot, fs : 0x100000, 0x4000000
mmc.2: system, fs : 0x4100000, 0x2f200000
mmc.2: cache, fs : 0x33300000, 0x1ac00000
mmc.2: misc, fs : 0x4e000000, 0x800000
mmc.2: recovery, fs : 0x4e900000, 0x1600000
mmc.2: userdata, fs : 0x50000000, 0x0
Support fstype : 2nd boot factory raw fat ext4 emmc nand ubi ubifs
Reserved part : partmap mem env cmd
DONE: Logo bmp 300 by 270 (3bpp), len=243054
DRAW: 0x47000000 -> 0x46000000
Load USB Driver: android
Core usb device tie configuration done
OTG cable Connected!
```

目标板进入Fastboot模式

```
F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash partmap Images/partmap.txt
sending 'partmap' (0 KB)... OKAY
writing 'partmap'... OKAY
F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash 2ndboot Images/2ndboot.bin
sending '2ndboot' (30 KB)... OKAY
writing '2ndboot'... OKAY
F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash bootloader Images/u-boot.bin
sending 'bootloader' (261 KB)... OKAY
writing 'bootloader'... OKAY
F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash boot Images/boot.img
sending 'boot' (22300 KB)... OKAY
writing 'boot'... OKAY
F:\OURS6818\Android\FAST_BOOT\fastboot_x32>fastboot flash system Images/system.img
sending 'system' (515429 KB)...
```

Fastboot批量烧写Android

升级完成后，批处理文件会自动关闭，升级过程可以到串口工具里可以看到，烧写完成后，系统自动重启。

说明：在批处理最后加入了 fastboot reboot 命令控制系统重启，用户可自行编辑批处理文件，删除此命令或者添加其他命令以完成更多功能。

7.6.6 实验总结

至此 fastboot 烧写已全部完成。本节 fastboot 烧写需要熟练掌握，目前绝大多数的手机平板都是采用此方法，此方法具有很强通用性。