翰林笔记技术文档

一、 Linux 系统启动过程

Linux 系统过程如下图所示：

步骤 1: 加载 BIOS 的硬件信息，并获取第一个启动设备的代号。
步骤 2: 读取第一个启动设备的 MBR 的引导加载程序（即 lilo、grub、spfdisk 等）的启动信息。
步骤 3: 加载操作系统的核心信息，核心开始解压，并尝试驱动所有的硬件设备。
步骤 4: 核心执行 init 程序并获得运行信息。
步骤 5: init 执行/etc/rc.d/rc.sysinit 文件。
步骤 6: 启动核心的外挂模块（/etc/modprobe.conf）。
步骤 7: init 执行运行的各个批处理文件（Script）
步骤 8: init 执行/etc/rc.d/rc.local 文件。
步骤 9: 执行/bin/login 程序，等待用户登录。
步骤 10: 登录之后开始以 shell 控制主机。

1. linux 核心的引导

操作系统的核心是放在文件系统中的，要想正确加载核心就必须提前识别文件系统。系统刚启动的时候，就需要通过引导加载程序（即 lilo、grub、spfdisk 等）来识别文件系统，引导核心。要想加载 linux 的核心就必须能识别 linux 的文件系统，要加载 windows 核心就必须能识别 windows 文件系统。而 grub 是同时支持 linux 和 windows 的。但 windows 的加载程序并不支持 linux 文件系统，所以在多重启动设置的时候，总是要先装 windows 而后装 Linux。

核心文件，一般来说，它会放在/boot 里，并且取名为/boot/vmlinuz。

在加载核心的过程中，必须知道，系统只会“载入根目录”，并且是以只读方式载入的。有时为了让某些功能可以用文件的方法来读取，有的系统在启动的时，会建立虚拟盘（ramdisk），这就需要使用 initrd 以及 linuxrc 了。在加载核心的时候，一起加载 initrd 的映像文件（boot/initrd-xxxx.img），并利用 linuxrc（在 initrd 的映像文件内）程序来加载模块。在核心驱动外部硬件的工作完成之后，initrd 所建立的虚拟盘就会被删除。

2. 第一个程序 init 的加载和 runlevel 设置

在核心加载完成之后，系统就准备好了，等待程序的执行。整个 linux 系统中，第一个执行的程序就是“/sbin/init”。init 程序做的工作相当多，除了利用设置文件“/etc/inittab”来获取运行等级之外，还会通过运行等级的设置值启动不同的服务项目。运行等级是指 linux 通过设置不同等级来规定系统用不同的服务来启动，让 linux 的使用环境不同。

"/etc/inittab"中有这么一句"si::sysinit:/etc/rc.d/rc.sysinit"，表明系统需要主动使用"rc.sysinit"这个 shell 脚本来设置系统环境。但这个文件的文件名在各个版本中是不一样的，需要自行查看确认。

3. init 处理系统的初始化流程（/etc/rc.d/rc.sysinit）
（1）获取网络环境与主机类型。首先会读取网络环境设置文件"/etc/sysconfig/network"，获取主机名称与默认网关等网络环境。
（2）测试与载入内存设备/proc 及 usb 设备/sys。除了/proc 外，系统会主动检测是否有 usb 设备，并主动加载 usb 驱动，尝试载入 usb 文件系统。
（3）决定是否启动 SELinux。
（4）接口设备的检测与即插即用（pnp）参数的测试。
（5）用户自定义模块的加载。用户可以再"/etc/sysconfig/modules/*.modules"加入自定义的模块，此时会加载到系统中。
（6）加载核心的相关设置。又一个文件"/etc/sysctl.conf"，按这个文件的设置值配置功能。
（7）设置系统时间（clock）。
（8）设置终端的控制台的字形。
（9）设置 raid 及 LVM 等硬盘功能。
（10）以方式查看检验磁盘文件系统。
（11）进行磁盘配额 quota 的转换。
（12）重新以读取模式载入系统磁盘。
（13）启动 quota 功能。
（14）启动系统随机数设备（产生随机数功能）。
（15）清楚启动过程中的临时文件。
（16）将启动信息加载到"/var/log/dmesg"文件中。
如果想知道启动过程中发生了什么事可以查看 dmesg 文件。

4. 启动系统服务"/etc/rc.d/rc*.d"与启动设置文件"/etc/sysconfig"

之前结束了 inittab 中的 rc.sysinit 之后，系统可以顺利工作了，只是还需要启动系统所需要的各种服务，这样主机才可以提供相关的网络和主机功能。因此根据之前设置的运行等级，会启动不同的服务项目。如果当时我们在 inittab 中选择了等级 3，系统则会在"/etc/rc.d/rc3.d"目录中运行相应的服务内容，选择等级 5，就在"/etc/rc.d/rc5.d"目录内。

该目录下的内容全部都是以 S 或 K 开头的链接文件，都链接到"/etc/rc.d/init.d"目录下的各种 shell 脚本。S 表示的是启动时需要 start 的服务内容，K 表示关机时需要关闭的服务内容。如果我们需要自己增加启动的内容，可以再 init.d 目录中增加相关的 shell 脚本，然后在 rc*.d 目录中建立链接文件指向该 shell 脚本。这些 shell 脚本的启动或结束顺序是由 S 或 K 字母后面的数字决定，例如 S01sysstat 表示第一个执行 sysstat 脚本，S99local 表示排在第 99 位执行 rc.local 脚本。

5. 用户自定义引导程序（/etc/rc.d/rc.local）

一般来说，自定义的程序不需要执行上面所说的繁琐的建立 shell 增加链接文件的步骤，只需要将命令放在 rc.local 里面就可以了，这个 shell 脚本就是保留给用户自定义启动内容的。

6. 启动终端和 X-Window 界面

完成了系统所有的启动任务后，linux 会启动终端或 X-Window 来等待用户登录。

tty1,tty2,tty3...这表示在运行等级 1，2，3，4 的时候，都会执行"/sbin/mingetty"，而且执行了 6 个，所以 linux 会有 6 个纯文本终端，mingetty 就是启动终端的命令。
除了这 6 个之外还会执行"/etc/X11/prefdm -nodaemon"这个主要启动 X-Window
7. 有关变换运行等级

当 linux 已经登录之后，有时候还希望更换运行等级，一种方法是改变"/etc/inittab"内的设置内容，将"id:3:initdefault:"中的数字改成相应等级，然后重启即可。

如果只是想暂时地改变运行等级，下次启动还是按原等级登录，可以直接使用 init [0-6]命令来改变运行等级。一般来说，运行等级的不同只是相关的启动服务内容的不同而已，因此使用命令改变等级会比较两个改变等级之间的服务内容，关闭一些新等级中不需要的服务项目，启动新等级需要的服务，而保量新等级和原等级中共有的服务内容。查询目前等级的命令也很简单，只需要输入 runlevel 即可

二、 Linux 驱动相关方法

嵌入式系统中，操作系统是通过各种驱动程序来驾驭硬件设备的。设备驱动程序是操作系统内核和硬件设备之间的接口，它为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，可以像操作普通文件一样对硬件设备进行操作。设备驱动程序是内核的一部分，完成以下功能：

◇ 驱动程序的注册和注销。
◇ 设备的打开和释放。
◇ 设备的读写操作。
◇ 设备的控制操作。
◇ 设备的中断和轮询处理。

Linux 主要将设备分为三类：字符设备、块设备和网络设备。字符设备是指发送和接收数据以字符的形式进行，没有缓冲区的设备；块设备是指发送和接收数据以整个数据缓冲区的形式进行的设备；网络设备是指网络设备访问的 BSD socket 接口。下面以字符设备为例，写出其驱动编写框架：

1、 编写驱动程序初始化函数

驱动程序的初始化在函数 xxx_init()中完成，包括对硬件初始化、中断函数、向内核注册驱动程序等。

首先理解硬件结构，搞清楚其功能，接口寄存器以及 CPU 怎么访问控制这些寄存器等。

其次向内核注册驱动程序。设备驱动程序可以直接编译进内核，在系统启动的时候初始化，也可以在需要的时候以模块的方式动态加载到内核中去。每个字符设备或是块设备都是通过 register_chrdev()函数注册，调用该函数后就可以向系统申请主设备号，操作成功，设备名就会出现在/proc/devices 里。

此外，在关闭设备时，需要先解除原先设备的注册，需要有清除函数，在 xxx_exit()中通过 unregister_chrdev()函数在实现，此后设备就会从/proc/devices 里消失。

当驱动程序被编译成模块时，使用 insmod 加载模块，模块的初始化函数 xxx_init()被调用，向内核注册驱动程序；使用 rmmod 卸载模块，模块的清除函数 xxx_exit()被调用。

2、 构造 file_operations 结构中要用到的各个成员函数

Linux 操作系统将所有的设备都看成文件，以操作文件的方式访问设备。应用程序不能直接操作硬件，使用统一的接口函数调用硬件驱动程序，这组接口被成为系统调用。每个系统调用中都有一个与之对应的函数（open、release、read、write、ioctl 等），在字符驱动程序中，这些函数集合在一个 file_operations 类型的数据结构中。以一个键盘驱动程序为例：

```
struct file_operations Key7279_fops =
{
.open = Key7279_Open,
.ioctl = Key7279_Ioctl,
.release = Key7279_Close,
.read = Key7279_Read,
};
```

a、 设备的打开和释放

打开设备是由 open()函数来完成，在大部分设备驱动中 open 完成如下工作：

◇ 递增计数器

◇ 检查特定设备的特殊情况

◇ 初始化设备

◇ 识别次设备号

释放设备由 release()函数来完成。当一个进程释放设备时，其它进程还能继续使用该设备，只是该进程暂时停止对该设备的的使用，而当一个进程关闭设备时，其它进程必须重新打开此设备才能使用。Release 完成如下工作：

◇ 递减计数

◇ 在最后一次释放设备操作时关闭设备

b、 设备的读写操作

读写设备的主要任务就是把内核空间的数据复制到用户空间，或者是从用户空间复制到内核空间，也就是将内核空间缓冲区里的数据复制到用户空间的缓冲区中或者相反。字符设备使用各自的 read()函数和 write()函数来进行数据读写。

c、 设备的控制操作

大部分设备除了读写能力，还可进行超出简单的数据传输之外的操作，所以设备驱动也必须具备进行各种硬件控制操作的能力. 这些操作常常通过 ioctl 方法来支持。与读写操作不同，ioctl()的用法与具体设备密切相关。以键盘 Key7279_Ioctl 为例：

```
static int Key7279_Ioctl(struct inode *inode,struct file *file,unsigned int cmd,
unsigned long arg)
{
switch(cmd)
  {
  case Key7279_GETKEY:
return key7279_getkey();
  default:
  printk("Unkown Keyboard Command ID.\n");
  }
```

```
  return 0;
}
```

cmd 的取值及含义都与具体的设备有关，除了 ioctl()，设备驱动程序还可能有其他控制函数，比如 llseek()等。

当应用程序使用 open、release 等函数打开某个设备时，设备驱动程序的 file_operations 结构中的相应成员就会被调用。

3、设备的中断和轮询处理

对于不支持中断的设备，读写时需要轮询设备状态，以及是否需要继续进行数据传输。例如，打印机。如果设备支持中断，则可按照中断方式进行。

模块在使用中断前要先请求一个中断通道(或者 IRQ 中断请求)，并在使用后释放它。通过 request_irq()函数来注册中断，free_irq()函数来释放。

三、 安卓内核编译流程

1、重要的 Make 文件

build/buildspec.mk
build/envsetup.sh
build/core/main.mk
build/core/envsetup.mk
build/config.mk

总的来说，Android 以模块(module/package)的形式来组织各个系统的部件，每个模块(module/package)的目录下都会有一个 Android.mk。所谓 module 就是指系统的 Native Code，而相对于 Java 写的 Android application 称为 package。

2、Makefile 主要流程

以下主要流程都在 build/core/main.mk 里安排。

初始化相关的参数设置(buildspec.mk、envsetup.mk、config.mk)
检测编译环境和目标环境
决定目标 product
读取 product 的配置信息及目标平台信息
清除输出目录
检查版本号
读取 Board 的配置
读取所有 Module 的配置
根据配置产生必要的规则(build/core/Makefile)
生成 image

2.1 初始化参数设置

在 main.mk 里，简单设置几个主要编译路径的变量后，来到 config.mk：

————————————————————config.mk————————————————

其中设置了源文件的一系列路径，包括头文件、库文件、服务、API 已经编译工具的路径。（前 36 行）

从 40 行开始，定义一些编译模块的生成规则：

这里面除了第一个 CLEAR_VARS 外，其他都对应的一种模块的生成规则，每一个 module 都会来 include 其中某个来生成目标模块。

例如：

Camera 模块的 makefile 里（Android.mk）就包含了其中的一种生成规则

BUILD_PACKAGE：
也就是 Camera 会按照 package.mk 里的生成规则去生成目标模块。
回到 config.mk，接着会尝试读取 buildspec.mk 的设置：

如同注释所说，会尝试查找 buildspec.mk，如果文件不存在会自动使用环境变量的设置，如果仍然未定义，会按 arm 默认的设置去 build。
这里的 buildspec.mk 可以自己创建，也可以将原先 build/下的 buildspec.mk.default 直接命名为 buildspec.mk 并移到根目录。

实际上，buildspec.mk 配置都被屏蔽了，我们可以根据需要直接打开和修改一些变量。在这里我们可以加入自己的目标产品信息：

```
ifndef TARGET_PRODUCT
TARGET_PRODUCT:=jinke

endif
```

以及输出目录设置：

```
OUT_DIR:=$(TOPDIR)jinke
```

读取 Product 的设定
回到 config.mk，接着进行全局变量设置，进入 envsetup.mk：
————————————————————envsetup.mk————————————————
里面的大部分函数都在 build/envsetup.sh 中定义。

首先，设置版本信息，(11 行)在 build/core/version_defaults.mk 中具体定义平台版本、SDK 版本、Product 版本，我们可以将 BUILD_NUMBER 作为我们产品 jinke 的 version 信息，当然，也可以自定义一个版本变量。

回到 envsetup.mk，接着设置默认目标产品(generic)，这里由于我们在 buildspec.mk 里设置过 TARGET_PRODUCT，事实上这个变量值为 jinke。

然后读取 product 的设置(41 行)，具体实现在 build/core/product_config.mk 中，进而进入 product.mk，从 build/target/product/AndroidProducts.mk 中读出 PRODUCT_MAKEFILES，这些 makefile 各自独立定义 product，而我们的产品 jinke 也应添加一个 makefile 文件 jinke.mk。在 jinke.mk 中我们可以加入所需编译的 PRODUCT_PACKAGES。
下面为 HOST 配置信息及输出目录，最后打印相关信息：
读取 BoardConfig 接着回到 config.mk，(114 行)这里会搜索所有的 BoardConfig.mk，主要有以下两个地方：

```
$(SRC_TARGET_DIR)/board/$(TARGET_DEVICE)/BoardConfig.mk
vendor/*/$(TARGET_DEVICE)/BoardConfig.mk
```

这里的 TARGET_DEVICE 就是 jinke，就是说为了定义我们自己的产品 jinke，我们要在 build/target/board 下添加一个自己的目录 jinke 用来加载自己的 board 配置。在 BoardConfig.mk 中会决定是否编译 bootloader、kernel 等信息。
读取所有 Module 结束全局变量配置后，回到 main.mk，马上对编译工具及版本进行检查，错误便中断编译。142 行，包含文件 definitions.mk，这里面定义了许多变量和函数供 main.mk 使用。main.mk 第 446 行，这里会去读取所有的 Android.mk 文件：

```
include $(ONE_SHOT_MAKEFILE)
```

这个 ONE_SHOT_MAKEFILE 是在前面提到的 mm(envsetup.mk)函数中赋值的：

```
ONE_SHOT_MAKEFILE=$M make -C $T files $@
```

而 M=$(findmakefile),最终实现在：
回到 main.mk，最终将遍历查找到的所有子目录下的 Android.mk 的路径保存到 subdir_makefiles 变量里(main.mk 里的 470 行)：

在 package/apps 下每个模块根目录都能看到 Android.mk，里面会去定义当前本地模块的 Tag：LOCAL_MODULE_TAGS，Android 会通过这个 Tag 来决定哪些本地模块会编译进系统，通过 PRODUCT 和 LOCAL_MODULE_TAGS 来决定哪些应用包会编译进系统。(也可以通过 buildspec.mk 来制定要编译进系统的模块)这个过程在 mian.mk 的 445 行开始，最后需要编译的模块路径打包到 ALL_DEFAULT_INSTALLED_MODULES(602 行)：
产生相应的 Rules，生成 image。
所有需要配置的准备工作都已完成，下面该决定如何生成 image 输出文件了，这一过程实际上在 build/core/Makefile 中处理的。
这里定义各种 img 的生成方式，包括 ramdisk.img、userdata.img、system.img、update.zip、recover.img 等。

当 Make include 所有的文件，完成对所有 make 文件的解析以后就会寻找生成对应目标的规则，依次生成它的依赖，直到所有满足的模块被编译好，然后使用相应的工具打包成相应的 img。

2.2 具体 make 操作：

2.2.1 完整编译

在根目录下输入 make 命令即可开始完全编译。这个命令实际编译生成的默认目标是 droid。也就是说，大家敲入 make 实际上执行的 make droid。而接下来大家看看 main.mk 文件里最后面的部分，会有很多伪目标，如 sdk、clean、clobber 等，这些在默认的 make droid 的命令下是不会执行的。我们可以在 make 后加上这些标签来单独实现一些操作。如：输入 make sdk 将会生成该版本对应的 SDK，输入 make clean 会清除上次编译的输出。

2.2.2 模块编译

有时候我们只修改了某一个模块，希望能单独编译这个模块而不是重新完整编译一次，这时候我们要用到 build/envsetup.sh 中提供的几个 bash 的帮助函数。
在源代码根目录下执行：
. build/envsetup.sh(.后面有空格)
这时可以用 help 命令查看帮助信息：
其中对模块编译有帮助的是 tapas、m、mm、mmm 这几个命令。
1、tapas——以交互方式设置 build 环境变量。
输入：tapas
第一步，选择目标设备：
例如 我们选择 1
第二步，选择代码格式：
第三步，选择产品平台：
注意：这里，Google 源代码里默认是 generic，而我们针对自己的产品应修改成响应的代码
2、m、mm、mmm 使用独立模块的 make 命令。
几个命令的功能使用 help 命令查看。
举个例子，我们修改了 Camera 模块的代码，现在需要重新单独编译这一块，这时可以使用 mmm 命令，后面跟指定模块的路径(注意是模块的根目录)。

具体如下：
mmm packages/apps/Camera/
为了可以直接测试改动，编译好后需要重新生成 system.img
可以执行：make snod
单独编译 image 文件
一般我们完整编译后，会生成三个重要的 image 文件：ramdisk.img、system.img 和 userdata.img。当然我们可以分开单独去编译这三个目标：
make ramdisk —— ramdisk.img
make userdataimage —— userdata.img
make systemimage —— system.img
2.3 一些疑问和解答：
1) 什么是 recovery.img？
顾名思义，recovery.img 是为了恢复系统的，相对于普通的 boot.img，recovery.img 多了一些图片文件(恢复时界面的背景)、/sbin/recovery/目录(跟恢复有关的二进制文件)，一些初始化文件也不相同(init.rc、init.goldfish.rc、default.prop)
这就是为什么启动恢复模式时会进入类似文本界面而不是图形界面。
将 recovery.img 文件复制到 SD 卡中，进入 shell 下输入：
mount -a
flash_image recovery /sdcard/recovery.img
若提示“no space on device”，可用 fastboot 模式刷
fastboot erase recovery
fastboot flash recovery recovery.img
在关机状态下按音量键+复位键进入 recovery 模式，根据选项选择需要的操作。
2) make sdk 和 make droid 编译有什么不同？
make sdk：
其实，执行 make sdk，编译后会在目录 out/host/linux-x86 里生成 sdk 目录，这个 sdk 和官方下载的 sdk 包是一样的，可以直接使用。
make droid：
实际上 droid 就是默认的生成目标，和直接敲 make 是一样的，都会完整编译出目标 image 文件，但不会编译出 sdk。
3) 我们将默认产品改为 jinke，那我怎么编译原先的 generic?
其实，无论编译哪个目标产品版本，只要产品相关设置存在，都可以直接在编译时加上目标产品名来编译。会在 out/target/product/下生成对应的产品输出目录，如：要编译 generic 版本：
make TARGET_PRODUCT := generic
3、编译前准备
sudo apt-get install build-essential
sudo apt-get install make
sudo apt-get install gcc
sudo apt-get install g++
sudo apt-get install libc6-dev
sudo apt-get install flex
sudo apt-get install bison
sudo apt-get install patch

sudo apt-get install texinfo
sudo apt-get install libncurses-dev
sudo apt-get install git-core gnupg //(gnupg 系统可能已自带)
sudo apt-get install flex bison gperf libsdl-dev libesd0-dev libwxgtk2.6-dev
buildessential
zip curl
sudo apt-get install ncurses-dev
sudo apt-get install zlib1g-dev

4、错误及解决方法

4.1、问题一：/bin/bash: bison: 未找到命令

解决方法：$ sudo apt-get install bison

4.2、问题二：

You are attempting to build with the incorrect version of java.
Your version is: /bin/bash: java: 未找到命令.
The correct version is: 1.5.

解决方法：

注意：必须使用1.5的JDK 版本。不能使用1.6等其他版本。

（1） 下载

在sun 官网上下载jdk-1_5_0_22-linux-i586.bin。注意不是jdk-1_5_0_22-linux-i586rpm.bin

（2） 部署

进入jdk-1_5_0_22-linux-i586.bin 所在目录

cd ~

赋予该bin 文件可执行属性

chmod +x jdk-1_5_0_22-linux-i586.bin

一直回车跳过协议到最后一行：

Do you agree to the above license terms? [yes or no]

键入yes 回车，开始解压。默认解压到当前目录下的jdk1.5.0_22目录里。

（3） 配置环境变量

执行$gedit ~/.bashrc 打开.bashrc 文件。

修改./bashrc 文件，在文件末尾添加

#set java environment
JAVA_HOME=“你的路径”/jdk1.5.0_10
export JRE_HOME=“你的路径”/jdk1.5.0_10/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH

保存并关闭文件。

在终端执行source ~/.bashrc 刷新该配置

使用java –version 检查JDK 是否配置正确。

4.3、问题三：

host c++: libhost <= build/libs/host/pseudolocalize.cpp/bin/bash: g++: 未找到命令
make: ***[out/host/linux-x86/obj/STATIC_LIBRARIES/libhost_intermediates/pseudolocalize.o]
错误127

解决方法：$ sudo apt-get install build-essential
4.4、问题四：
external/clearsilver/cgi/cgi.c:22:18: 致命错误: zlib.h: 没有那个文件或目录编译终端。
make: ***[out/host/linux-x86/obj/SHARED_LIBRARIES/libneo_cgi_intermediates/cgi.o] 错误1
解决方法：$ sudo apt-get install zlib1g-dev
4.5、问题五：
make: ***[out/host/linux-x86/obj/EXECUTABLES/aapt_intermediates/appt] 错误1
解决方法：
打开Android.mk 文件
$gedit frameworks/base/tools/aapt/Android.mk
编辑下面一行：
ifeq ($(HOST_OS),linux)
#LOCAL_LDLIBS += -lrt 把这行注释掉，改为下面一行。
LOCAL_LDLIBS += -lrt -lpthread
endif
4.6、问题六：
host c: adb<= system/core/adb/fdevent.c
host Executable:adb (out/host/linux-86/obj/EXECUTABLES/adb_intermediates/adb)/user/bin/ld:
cannot find －lncurses
collect2: ld 返回1
make: ***[out/host/linux-86/obj/EXECUTABLES/adb_intermediates/adb] 错误1
解决方法：$sudo apt-get install libncurses5-dev
4.7、问题七：
make:
***[out/target/product/sam9g45/obj/STATIC_LIBRARIES/libwebcore_intermediates/WebCore/cs
s/CSSPropertyNames.h] 错误25
解决方法：$sudo apt-get install gpert
4.8、问题八：
host C++: libutils <= frameworks/base/libs/utils/RefBase.cpp
frameworks/base/libs/utils/RefBase.cpp: In member function ‘void android::RefBase::weakref_type::trackMe(bool, bool)’:
frameworks/base/libs/utils/RefBase.cpp:483:67: error: passing ‘const android::RefBase::weakref_impl’ as ‘this’ argument of ‘void android::RefBase::weakref_impl::trackMe(bool, bool)’ discards qualifiers [-fpermissive]
make: *** [out/host/linux-x86/obj/STATIC_LIBRARIES/libutils_intermediates/RefBase.o] 错误1
解决方法：

打开Android.mk
$ gedit frameworks/base/libs/utils/Android.mk
将下面这一行
LOCAL_CFLAGS += -DLIBUTILS_NATIVE=1 $(TOOL_CFLAGS)
改为
LOCAL_CFLAGS += -DLIBUTILS_NATIVE=1 $(TOOL_CFLAGS) -fpermissive
4.9、问题九：
make:
***[out/host/linux-
x86/obj/EXECUTABLES/emulator_intermediates/android/skin/window.o]
Error 1
解决方法：$ sudo apt-get install libx11-dev
4.10、问题十：
make: *** [out/host/linux-x86/obj/EXECUTABLES/localize_intermediates/localize]
错误1
解决方法：
打开Android.mk：
$gedit ./framework/base/tools/localize/Android.mk 文件(注意与问题五中的文件是不同文件)
编辑下面一行：
ifeq ($(HOST_OS),linux)
#LOCAL_LDLIBS += -lrt 把这行注释掉，改为下面一行。
LOCAL_LDLIBS += -lrt －lpthread
4.11、问题十一：
make:
***[out/host/linux-
x86/obj/EXECUTABLES/accRuntimeTest_intermediates/accRuntimeTest] 错误1
解决方法：
打开Android.mk:
$gedit system/core/libacc/tests/Android.mk(注意与问题五和问题十不是同一个文件)
将以下缺少的语句添加进去：
LOCAL_SHARED_LIBRARIES := \
libacc
LOCAL_LDLIBS += -ldl
LOCAL_MODULE_TAGS := tests
在include $(BUILD_HOST_EXECUTABLE)下面添加以下缺少的语句：
LOCAL_SHARED_LIBRARIES := \
libacc
LOCAL_LDLIBS += -ldl
LOCAL_MODULE_TAGS := tests
1. source build/envsetup.sh //初始化与环境envsetup.sh 脚本
初始化完成，显示如下

including device/samsung/maguro/vendorsetup.sh
including device/samsung/tuna/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including sdk/bash_completion/adb.bash
2. lunch full-eng //选择的目标
==
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.1
TARGET_PRODUCT=full
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=ITL41D
==
//建立与一切使。GNU 的make -JN 参数可以并行处理任务，它是共同使用的任务数，
//N 的1 倍和2 倍之间是被用于建立计算机硬件线程数量。例如在E5520 双机（2 个CPU，
//每个CPU 4 核，每核心2 线程），最快的构建与命令之间的J16 和 -J32。
3. make -j4
编译完成
Target system fs image:
out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
Install system fs image: out/target/product/generic/system.img
编译 sdk
1.source build/envsetup.sh 初始化构建环境
2.lunch sdk-eng 选择目标
3.make sdk
编译成功，在 out/host/linux-x86/sdk/下生成文件
Package SDK: out/host/linux-x86/sdk/android-sdk_eng.root_linux-x86.zip
如果木有权限查看该目录，那么目录下面为显示为空，但通过 root 终端可以看到得。
out/host/linux-x86/sdk/ 添加如下权限，就可以看到编译完成得sdk 了。
chmod 777 * -R
常见错误：
11、 /bin/bash: flex: 未找到命令
解决方法：sudo apt-get install bison flex
12、

system/core/gpttool/gpttool.c:24:18: fatal error: zlib.h: 没有那个文件或目录
compilation terminated.
make: *** [out/host/linux-x86/obj/EXECUTABLES/gpttool_intermediates/gpttool.o] 错误 1
make: *** 正在等待未完成的任务....
解决方法： sudo apt-get install zlib-devel
13、host Executable: bb2sym (out/host/linux-x86/obj/EXECUTABLES/bb2sym_intermediates/bb2sym)
/usr/bin/ld: cannot find -lncurses
collect2: ld returned 1 exit status
make: *** [out/host/linux-x86/obj/EXECUTABLES/adb_intermediates/adb] 错误 1
解决方法：sudo apt-get install libncurses5-dev
14、
development/tools/emulator/opengl/host/libs/Translator/GLcommon/../include/EGL/eglplatform.h:85:
22: fatal error: X11/Xlib.h: 没有那个文件或目录
compilation terminated.
make: *** [out/host/linux-x86/obj/EXECUTABLES/triangleCM_intermediates/triangleCM.o] 错误 1
解决方法：sudo apt-get install libx11-dev
6.fatal error: GL/glx.h: 没有那个文件或目录
解决方法：sudo apt-get install libgl1-mesa-dev
15、gperf: not found
解决方法：sudo apt-get install gperf

5、编译完成之后的代码结构
Android 编译完成后，将在根目录中生成一个out 文件夹，所有生成的内容均放置在这个文件夹中。
out 文件夹如下所示：

```
out/
|-- CaseCheck.txt
|-- casecheck.txt
|-- host
| |-- common
| `-- linux-x86
`-- target
|-- common
`-- product
```

主要的两个目录为host 和target，前者表示在主机（x86）生成的工具，后者表示目标机（模认
为ARMv5）运行的内容。
host 目录的结构如下所示：

```
out/host/
|-- common
```

| `-- obj （JAVA 库）
`-- linux-x86
|-- bin （二进制程序）
|-- framework （JAVA 库,＊.jar 文件）
|-- lib （共享库*.so）
`-- obj （中间生成的目标文件）
host 目录是一些在主机上用的工具，有一些是二进制程序,有一些是JAVA 的程序。
target 目录的结构如下所示：
out/target/
|-- common
| |-- R （资源文件）
| |-- docs
| `-- obj （目标文件）
`-- product
`-- generic
其中common 目录表示通用的内容，product 中则是针对产品的内容。
在common 目录的obj 中，包含两个重要的目录：
APPS 中包含了JAVA 应用程序生成的目标，每个应用程序对应其中一个子目录，将结合每个应
用程序的原始文件生成Android 应用程序的APK 包。
JAVA_LIBRARIES 中包含了JAVA 的库，每个库对应其中一个子目录。
在默认的情况下，Android 编译将生成generic 目录，如果选定产品还可以生成其他的目录。
generic 包含了以下内容：
out/target/product/generic/
|-- android-info.txt
|-- clean_steps.mk
|-- data
|-- obj
|-- ramdisk.img
|-- root
|-- symbols
|-- system
|-- system.img
|-- userdata-qemu.img
`-- userdata.img
在generic/obj/APPS 目录中包含了各种JAVA 应用，与common/APPS 相对应，但是已经打成
了APK 包。
system 目录是主要的文件系统，data 目录是存放数据的文件系统。
obj/SHARED_LIBRARIES 中存放所有动态库。
obj/STATIC_LIBRARIES 中存放所有静态库。
几个以img 为结尾的文件是几个目标映像文件，其中ramdisk 是作为内存盘的

根文件系统映像，
system.img 是主要文件系统的映像，这是一个比较大的文件，data.img 是数据内容映像。这几个
image 文件是运行时真正需要的文件。

6、 build 系统简介
6.1.1.build 系统文件结构
```
./build
|-- CleanSpec.mk
|-- buildspec.mk.default
|-- core
| |-- Makefile
| |-- apicheck_msg_current.txt
| |-- apicheck_msg_last.txt
| |-- armelf.x
| |-- armelf.xsc
| |-- armelflib.x
| |-- base_rules.mk
| |-- binary.mk
| |-- build-system.html
| |-- build_id.mk
| |-- checktree
| |-- cleanbuild.mk
| |-- cleanspec.mk
| |-- clear_vars.mk
| |-- combo
| | |-- HOST_darwin-x86.mk
| | |-- HOST_linux-x86.mk
| | |-- HOST_windows-x86.mk
| | |-- TARGET_linux-arm.mk
| | |-- TARGET_linux-sh.mk
| | |-- TARGET_linux-x86.mk
| | |-- arch
| | | `-- arm
| | | |-- armv4t.mk
| | | |-- armv5te-vfp.mk
| | | |-- armv5te.mk
| | | |-- armv7-a-neon.mk
| | | `-- armv7-a.mk
| | |-- javac.mk
| | `-- select.mk
| |-- config.mk
| |-- copy_headers.mk
| |-- definitions.mk
```

```
| |-- device.mk
| |-- distdir.mk
| |-- droiddoc.mk
| |-- dynamic_binary.mk
| |-- envsetup.mk
| |-- executable.mk
| |-- filter_symbols.sh
| |-- find-jdk-tools-jar.sh
| |-- host_executable.mk
| |-- host_java_library.mk
| |-- host_prebuilt.mk
| |-- host_shared_library.mk
| |-- host_static_library.mk
| |-- java.mk
| |-- java_library.mk
| |-- key_char_map.mk
| |-- main.mk
| |-- multi_prebuilt.mk
| |-- node_fns.mk
| |-- notice_files.mk
| |-- package.mk
| |-- pathmap.mk
| |-- prebuilt.mk
| |-- prelink-linux-arm-2G.map
| |-- prelink-linux-arm.map
| |-- process_wrapper.sh
| |-- process_wrapper_gdb.cmds
| |-- process_wrapper_gdb.sh
| |-- product.mk
| |-- product_config.mk
| |-- proguard.flags
| |-- proguard_tests.flags
| |-- raw_executable.mk
| |-- raw_static_library.mk
| |-- root.mk
| |-- shared_library.mk
| |-- static_java_library.mk
| |-- static_library.mk
| |-- tasks
| | |-- apicheck.mk
| | |-- cts.mk
| | |-- product-graph.mk
| | `-- sdk-addon.mk
| `-- version_defaults.mk
```

```
|-- envsetup.sh
|-- libs
| `-- host
| |-- Android.mk
| |-- CopyFile.c
| |-- include
| | `-- host
| | |-- CopyFile.h
| | |-- Directories.h
| | `-- pseudolocalize.h
| |-- list.java
| `-- pseudolocalize.cpp
|-- target
| |-- board
| | |-- Android.mk
| | |-- emulator
| | | |-- AndroidBoard.mk
| | | |-- BoardConfig.mk
| | | |-- README.txt
| | | |-- tuttle2.kcm
| | | `-- tuttle2.kl
| | |-- generic
| | | |-- AndroidBoard.mk
| | | |-- BoardConfig.mk
| | | |-- README.txt
| | | |-- system.prop
| | | |-- tuttle2.kcm
| | | `-- tuttle2.kl
| | `-- sim
| | |-- AndroidBoard.mk
| | `-- BoardConfig.mk
| `-- product
| |-- AndroidProducts.mk
| |-- core.mk
| |-- full.mk
| |-- generic.mk
| |-- languages_full.mk
| |-- languages_small.mk
| |-- sdk.mk
| |-- security
| | |-- README
| | |-- media.pk8
| | |-- media.x509.pem
| | |-- platform.pk8
```

| | |-- platform.x509.pem
| | |-- shared.pk8
| | |-- shared.x509.pem
| | |-- testkey.pk8
| | `-- testkey.x509.pem
| `-- sim.mk
6.1.2.make 文件分类
配置类
主要用来配置product、board，以及根据你的Host 和Target 选择相应的工具以及设定相应的通用
编译选项：
config 文件
说明
build/core/config.mk
Config 文件的概括性配置
build/core/envsetup.mk
generate 目录构成等配置
build/target/product
产品相关的配置
build/target/board
硬件相关的配置
build/core/combo
编译选项配置
这里解释下这里的board 和product。board 主要是设计到硬件芯片的配置，比如是否提供硬件的某些功能，比如说GPU 等等，或者芯片支持浮点运算等等。product 是指针对当前的芯片配置定义你将
要生产产品的个性配置，主要是指APK 方面的配置，哪些APK 会包含在哪product 中，哪些APK在当前product 中是不提供的。config.mk 是一个总括性的东西，它里面定义了各种module 编译所需要使用的HOST 工具以及如何来编译各种模块，比如说BUILT_PREBUILT 就定义了如何来编译预编译模块。
envsetup.mk 主要会读取由envsetup.sh 写入环境变量中的一些变量来配置编译过程中的输出目录，combo 里面主要定义了各种Host 和Target 结合的编译器和编译选项。
模块组织类这类文件主要定义了如何来处理Module 的Android.mk，以及采用何种方式来生成目标模块，这些模块生成规则都定义在config.mk 里面。可以阅读：
CLEAR_VARS:= $(BUILD_SYSTEM)/clear_vars.mk
BUILD_HOST_STATIC_LIBRARY:=$(BUILD_SYSTEM)/host_static_library.mk
BUILD_HOST_SHARED_LIBRARY:=$(BUILD_SYSTEM)/host_shared_library.mk
BUILD_STATIC_LIBRARY:=$(BUILD_SYSTEM)/static_library.mk
BUILD_RAW_STATIC_LIBRARY :=$(BUILD_SYSTEM)/raw_static_library.mk
BUILD_SHARED_LIBRARY:=$(BUILD_SYSTEM)/shared_library.mk
BUILD_EXECUTABLE:= $(BUILD_SYSTEM)/executable.mk
BUILD_RAW_EXECUTABLE:=$(BUILD_SYSTEM)/raw_executable.mk

BUILD_HOST_EXECUTABLE:=$(BUILD_SYSTEM)/host_executable.mk
BUILD_PACKAGE:= $(BUILD_SYSTEM)/package.mk
BUILD_HOST_PREBUILT:=$(BUILD_SYSTEM)/host_prebuilt.mk
BUILD_PREBUILT:= $(BUILD_SYSTEM)/prebuilt.mk
BUILD_MULTI_PREBUILT:=$(BUILD_SYSTEM)/multi_prebuilt.mk
BUILD_JAVA_LIBRARY:= $(BUILD_SYSTEM)/java_library.mk
BUILD_STATIC_JAVA_LIBRARY:=$(BUILD_SYSTEM)/static_java_library.mk
BUILD_HOST_JAVA_LIBRARY:=$(BUILD_SYSTEM)/host_java_library.mk
BUILD_DROIDDOC:= $(BUILD_SYSTEM)/droiddoc.mk
BUILD_COPY_HEADERS := $(BUILD_SYSTEM)/copy_headers.mk
BUILD_KEY_CHAR_MAP :=$(BUILD_SYSTEM)/key_char_map.mk
除了CLEAR_VARS 是清楚本地变量之外，其他所有的都对应了一种模块的生成规则，每一个本地模块最后都会include 其中的一种来生成目标模块。大部分上面的.mk 都会包含base_rules.mk，这是对模块进行处理的基础文件。
单个模块编译类：
本地模块的Makefile 文件就是我们在Android 里面几乎上随处可见的Android.mk。Android 进行编
译的时候会通过下面的函数来遍历所有子目录中的Android.mk，一旦找到就不会再往层子目录继续寻
找(所有你的模块定义的顶层Android.mk 必须包含自己定义的子目录中的Android.mk)。
subdir_makefiles += \
$(shellbuild/tools/findleaves.sh --prune="./out" $(subdirs) Android.mk)
不同类型的本地模块具有不同的语法，但基本上是相通的，只有个别变量的不同。
Android 通过LOCAL_MODULE_TAGS 来决定哪些本地模块会不会编译进系统，通过PRODUCT 和
LOCAL_MODULE_TAGS 来决定哪些应用包会编译进系统，如果用户不指定LOCAL_MODULE_TAGS，默认它的值是user。此外用户可以通过buildspec.mk 来指定你需要编译进系统的模块。用户也可以通过mm 来编译指定模块，或者通过make clean-module_name 来删除
指定模块。
系统生成类
这主要指的是build/core/Makefile 这个文件，它定义了生成各种img 的方式，包括ramdisk.img
userdata.img system.img update.zip recover.img 等。
对应着我们常用的几个make goals。
在实际的过程中，我们也可以自己编辑out 目录下的生成文件，然后手工打包相应生成相应的img，最常用的是加入一些需要集成进的prebuilt file。所有的Makefile 都通过build/core/main.mk 这个文件组织在一起，它定义了一个默认goals：droid，当我们在TOP 目录下敲Make 实际上就等同于我们执行make droid。当Make include 所有的文件，完成对所有make 文件的解析以后就会寻找生成droid 的规则，依次生成它的依赖，直到所有满足的模块被编译好，然后使用相应的工具打包成相应的img。

6.2. makefile 文件
控制整个android 系统编译的make 文件。其内容如下：
DO NOT EDIT THIS FILE
include build/core/main.mk
DO NOT EDIT THIS FILE
可以看出，实际上控制编译的文件是：build/core/main.mk
6.3. Make 命令
make droid：等同于make 命令。droid 是默认的目标名称。
make all： make all 将make 所有make droid 会编译的项目。同时，将编译LOCAL_MODULE_TAGS 定义的不包括android tag 的模块。这将确保所有的在代码树里面同时有
Android.mk 文件的模块。
clean-$(LOCAL_MODULE)和clean-$(LOCAL_PACKAGE_NAME)：
删除某个模块的目标文件。例如：clean-libutils 将删除所有的libutils.so 以及和它相关的中间文件；
clean-Home 将删除Home 应用。
make clean：删除本次配置所编译输出的结果文件。类似于：rm －rf ./out/<configuration>
make clobber：删除所有配置所编译输出的结果文件。类似于：rm －rf ./out/
make dataclean：make dataclean deletes contents of the data directory inside the current combo directory. This is especially useful on the simulator and emulator, where
the persistent data remains present between builds.
make showcommands：在编译的时候显示脚本的命令，而不是显示编译的简报。用于调试脚本。
make LOCAL_MODULE：编译一个单独得模块（需要有Android.mk 文件存在）。
make targets：将输出所有拟可以编译的模块名称列表。
注：还有一些命令，从make 文件里面应该可以找到。本文不做探讨。
6.4. build/core/config.mk
config.mk 文件的主要内容如下：
头文件的定义；（各种include 文件夹的设定）
在定义头文件的部分，还include 了pathmap.mk，如下：
include $(BUILD_SYSTEM)/pathmap.mk
该文件设置include 目录和frameworks/base 下子目录等的信息。
编译系统内部mk 文件的定义； <Build system internal files>
设定通用的名称；<Set common values>
Include 必要的子配置文件；<Include sub-configuration files>
buildspec.mk
envsetup.mk
BoardConfig.mk
/combo/select.mk
/combo/javac.mk
检查BUILD_ENV_SEQUENCE_NUMBER 版本号；
In order to make easier for people when the build system changes, when it is

necessary
to make changes to buildspec.mk or to rerun the environment setup scripts, they contain
a version number in the variable BUILD_ENV_SEQUENCE_NUMBER. If this variable does
not match what the build system expects, it fails printing an error message explaining what happened. If you make a change that requires an update, you need to update two
places so this message will be printed.

- In config/envsetup.make, increment the CORRECT_BUILD_ENV_SEQUENCE_NUMBER definition.
- In buildspec.mk.default, update the BUILD_ENV_SEQUENCE_DUMBER definition to match the one in config/envsetup.make

The scripts automatically get the value from the build system, so they will trigger the warning as well.

设置常用工具的常量；< Generic tools.>

设置目标选项；< Set up final options.>

遍历并设置SDK 版本；

6.5. buildspec.mk

默认情况下，buildspec.mk 文件是不存在的，表示使用的多少默认选项。Android 只提供了buildspec.mk 文件的模板文件build/buildspec.mk.default。如果需要使用buildspec.mk 文件，需要将该文件拷贝到<srcDir>根目录下面，并命名为buildspec.mk。同时，需要将模板文件里面的一些必要的配置项启用或者修改为你所需要的目标选项。buildspec.mk 文件主要配置下面的选项：

TARGET_PRODUCT：设置编译之后的目标（产品）类型；

可以设置的值在：build/target/product/中定义。比如，product 目录下有下面几个mk 文件：

AndroidProducts.mk
core.mk
full.mk
generic.mk
languages_full.mk
languages_small.mk
sdk.mk
sim.mk

那么，在这里可以设置的值就为上面几个mk 文件的前缀名称（generic 等）。

TARGET_BUILD_VARIANT：设置image 的类型；

包括三个选项：user、userdebug、eng。

usr： 出厂时候面向用户的image；

userdebug： 打开了一些debug 选项的image；

eng： 为了开发而包含了很多工具的image

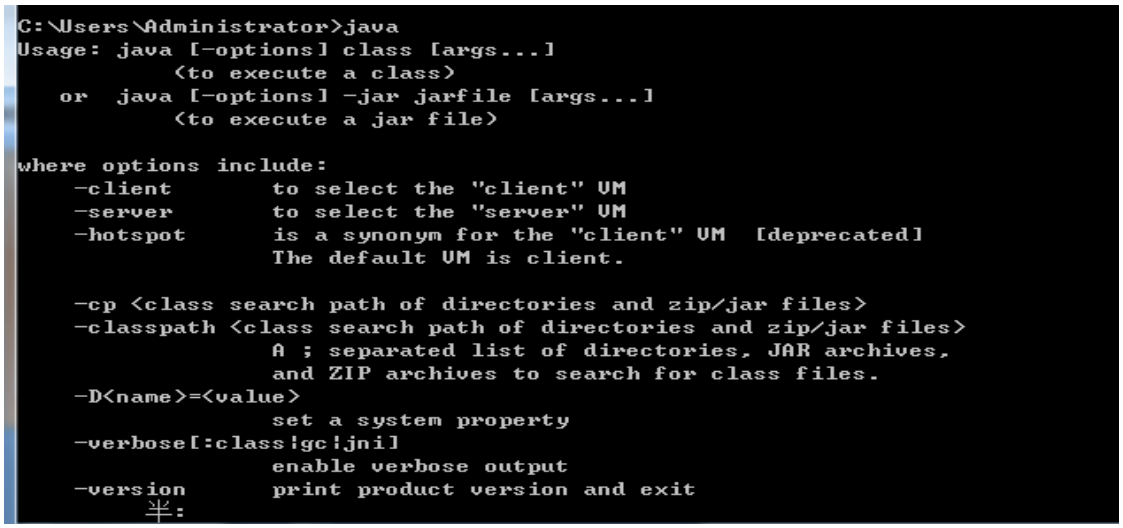
Ø CUSTOM_MODULES：设置额外的总是会被安装到系统的模块；

这里设置的模块名称采用的是简单目标名，比如：Browser 或者MyApp 等。这些名字在

LOCAL_MODULE 或者在LOCAL_PACKAGE_NAME 里面定义的。
LOCAL_MODULE is the name of what's supposed to be generated from your Android.mk.
For exmample, for libkjs, the LOCAL_MODULE is "libkjs" (the build system adds the appropriate suffix -- .so .dylib .dll). For app modules, use LOCAL_PACKAGE_NAME instead of LOCAL_MODULE. We're planning on switching to ant for the apps, so this might
become moot.
TARGET_SIMULATOR：设置是否要编译成simulator <true or false>；
TARGET_BUILD_TYPE：设置是debug 还是release 版本<release or debug>；
Set this to debug or release if you care. Otherwise, it defaults to release for arm and debug for the simulator.
HOST_BUILD_TYPE：设置Host 目标是debug 版还是release 版；
<release or debug, default is debug>
DEBUG_MODULE_ModuleName：配置单个模块的版本是debug 还是release；
<ture or false>
TARGET_TOOLS_PREFIX：工具名前缀，默认为NULL
HOST_CUSTOM_DEBUG_CFLAGS/ TARGET_CUSTOM_DEBUG_CFLAGS：增加额外的编译
选项LOCAL_CFLAGS。
LOCAL_CFLAGS：If you have additional flags to pass into the C or C++ compiler, add them here. For example: LOCAL_CFLAGS += -DLIBUTILS_NATIVE=1
CUSTOM_LOCALES：增加额外的LOCALES 到最总的image；
Any locales that appear in CUSTOM_LOCALES but not in the locale list for the selected
product will be added to the end of PRODUCT_LOCALES.
OUT_DIR：编译之后文件保存路径。默认为<build-root>/out 目录；
ADDITIONAL_BUILD_PROPERTIES：指定（增加）额外的属性文件；
NO_FALLBACK_FONT：设置是否只支持英文（这将减少image 的大小）。<true, false>
WEBCORE_INSTRUMENTATION：webcore 支持；
ENABLE_SVG：SVG 支持；
BUILD_ENV_SEQUENCE_NUMBER：编译系列号。

四、 翰林笔记应用层

1、开发环境配置
1.1、环境搭建
1.1.1、JDK 安装
如果还没有安装 JDK 的话，可以到 ORACLE 官网：
http://www.oracle.com/technetwork/indexes/downloads/index.html 这里下载，然后安装，并设置计算机环境变量。步骤如下：
安装完成之后，可以在检查 JDK 是否安装成功。打开 cmd 窗口，输入 java －version 查看 JDK 的版本信息。



1.1.2、Eclipse 安装

如果你还么有 Eclipse 的话，可以去这里下载，下载适合本地计算机操作系统的 Eclipse IDE 版本：

Eclipse 下载　可以到官网下载：http://www.eclipse.org/downloads/

1. 1.3、Android SDK 安装

在 Android Developers 下载 android-sdk，下载地址：

http://developer.android.com/sdk/index.html#download 下载完成后解压。

运行 SDK Setup.exe，点击 Available Packages。如果没有出现可安装的包，请点击 Settings，选中 Misc 中的"Force https://..."这项，再点击 Available Packages 。

选择希望安装的 SDK 及其文档或者其它包，点击 Installation Selected、Accept All、Install Accepted，开始下载安装所选包

在用户变量中新建 PATH 值为：Android SDK 中的 tools 绝对路径。

(E:\android\AndroidSDK\tools 我的本地环境)

设置 Android SDK 的环境变量

“确定”后，重新启动计算机。重启计算机以后，进入 cmd 命令窗口，检查 SDK 是不是安装成功。

运行 android －h 如果有类似以下的输出，表明安装成功：

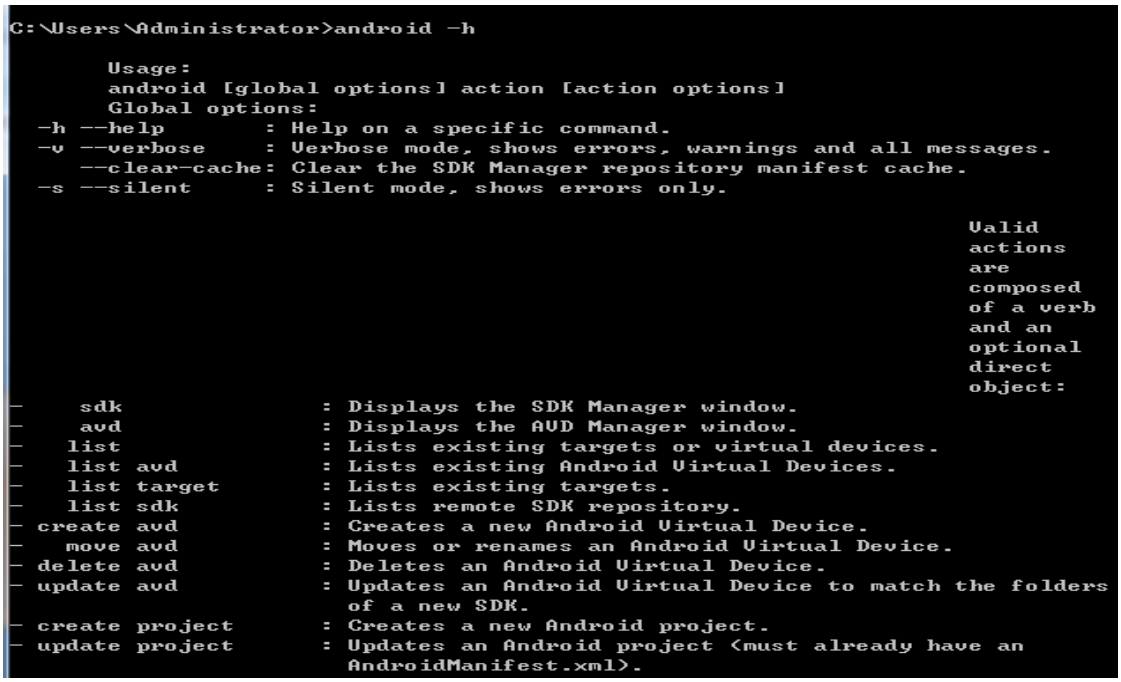
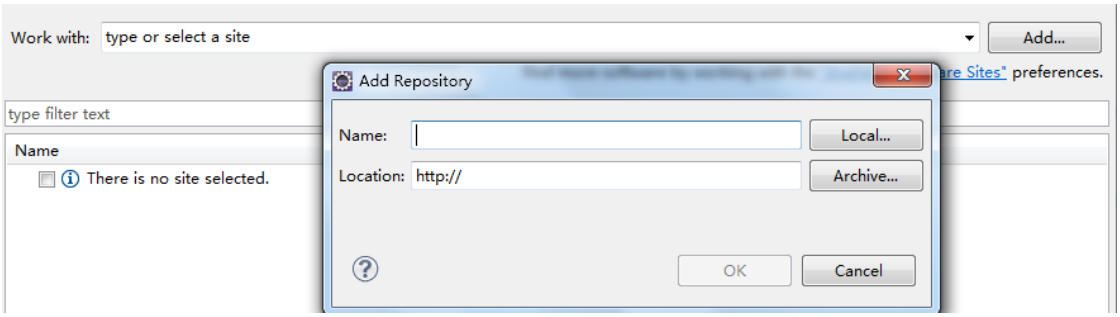


图 3、验证 Android SDK 是否安装成功

1.1.4、ADT 安装

打开 Eclipse IDE，进入菜单中的 "Help" -> "Install New Software"

点击 Add...按钮，弹出对话框要求输入 Name 和 Location：Name 自己随便取，Location 输入 http://dl-ssl.google.com/android/eclipse。



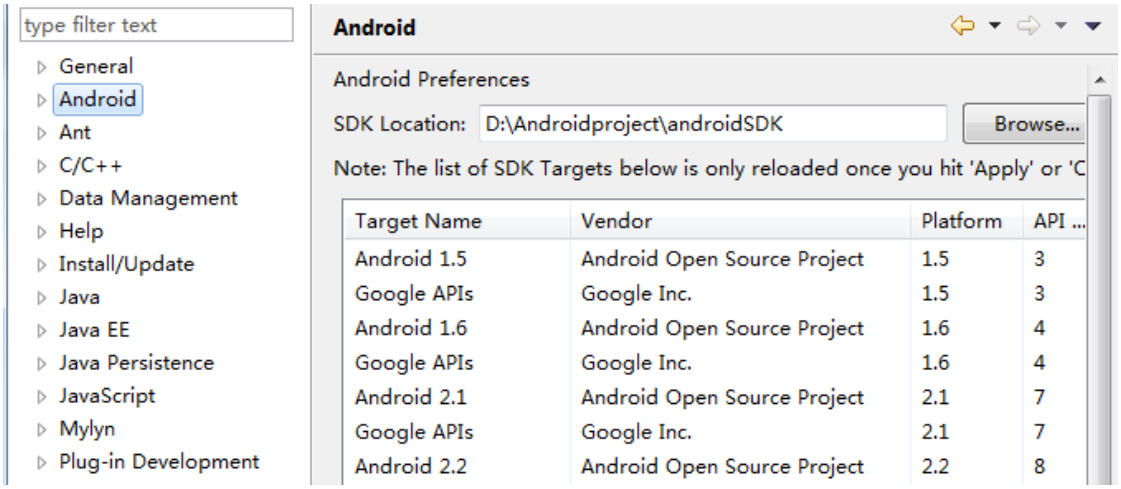
确定返回后，在 work with 后的下拉列表中选择我们刚才添加的 ADT，我们会看到下面出有 Developer Tools，展开它会有 Android DDMS 和 Android Development Tool，勾选他们。 如下图所示：

然后就是按提示一步一步 next。

完成之后：选择 Window > Preferences...

在左边的面板选择 Android，然后在右侧点击 Browse...并选中 SDK 路径，本机为：

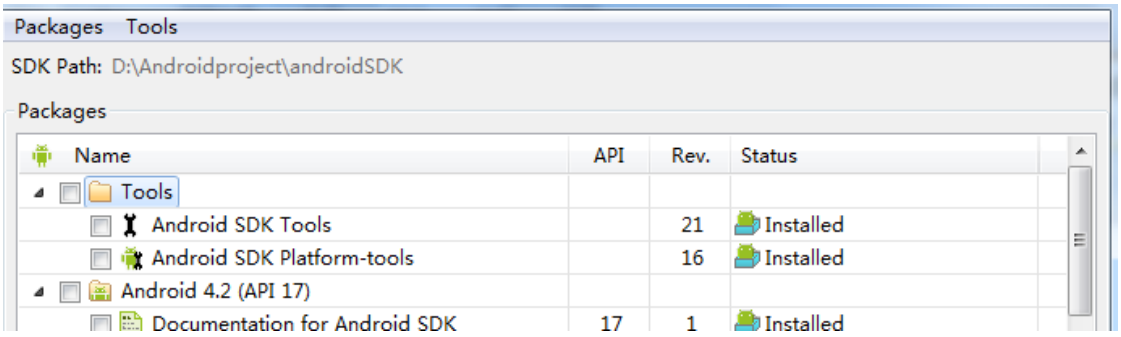
D:\Androidproject\androidSDK 点击 Apply、OK。配置完成。



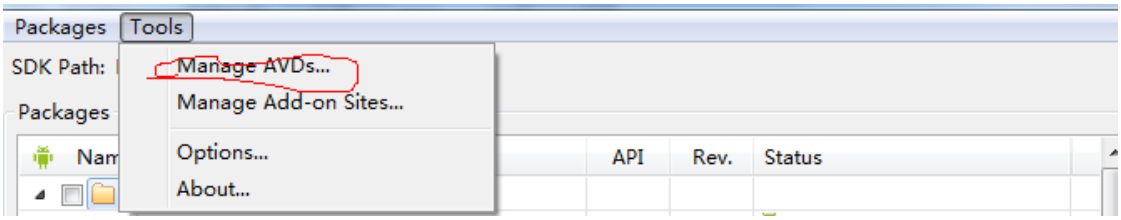
1.1.5、创建 AVD

为使 Android 应用程序可以在模拟器上运行，必须创建 AVD。

在 Eclipse 中。选择 Windows > Android SDK and AVD Manager (新版本的 eclipse 中没有这个选项，可以点击运行 AndroidSDK 有弹框如下：



点击左上角的 Tool：



如上图，点击后：

2、点击左侧面板的 Virtual Devices，再右侧点击 New

3、填入 Name，选择 Target 的 API，SD Card 大小任意，Skin 随便选，Hardware 目前保持默认值

4、点击 Create AVD 即可完成创建 AVD

注意：如果你点击左侧面板的 Virtual Devices，再右侧点击 New ，而 target 下拉列表没有可选项时，这时候你：

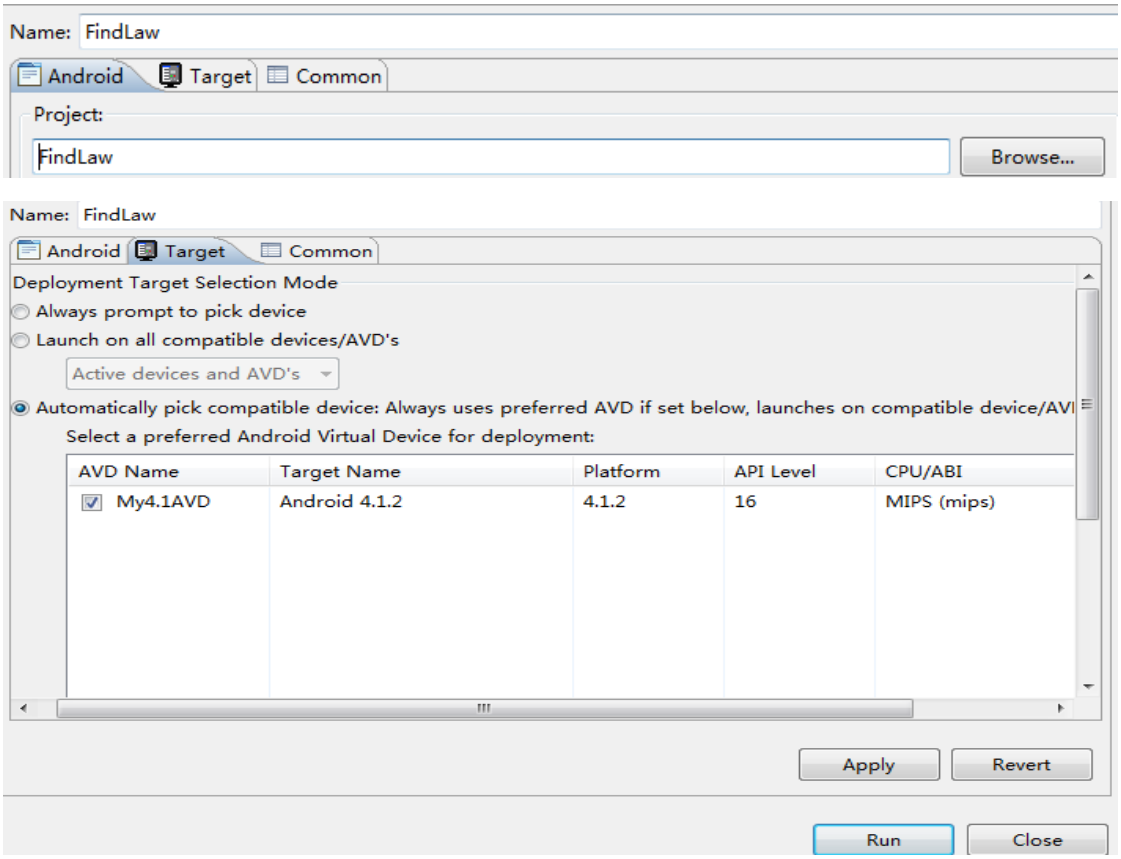
点击左侧面板的 Available Packages，在右侧勾选 https://dl-ssl.google.com/android/repository/repository.xml，

然后点击 Install Selected 按钮，接下来就是按提示做就行了

要做这两步，原因是在 1.3、Android SDK 安装中没有安装一些必要的可用包（Available Packages）。

2、FindLaw

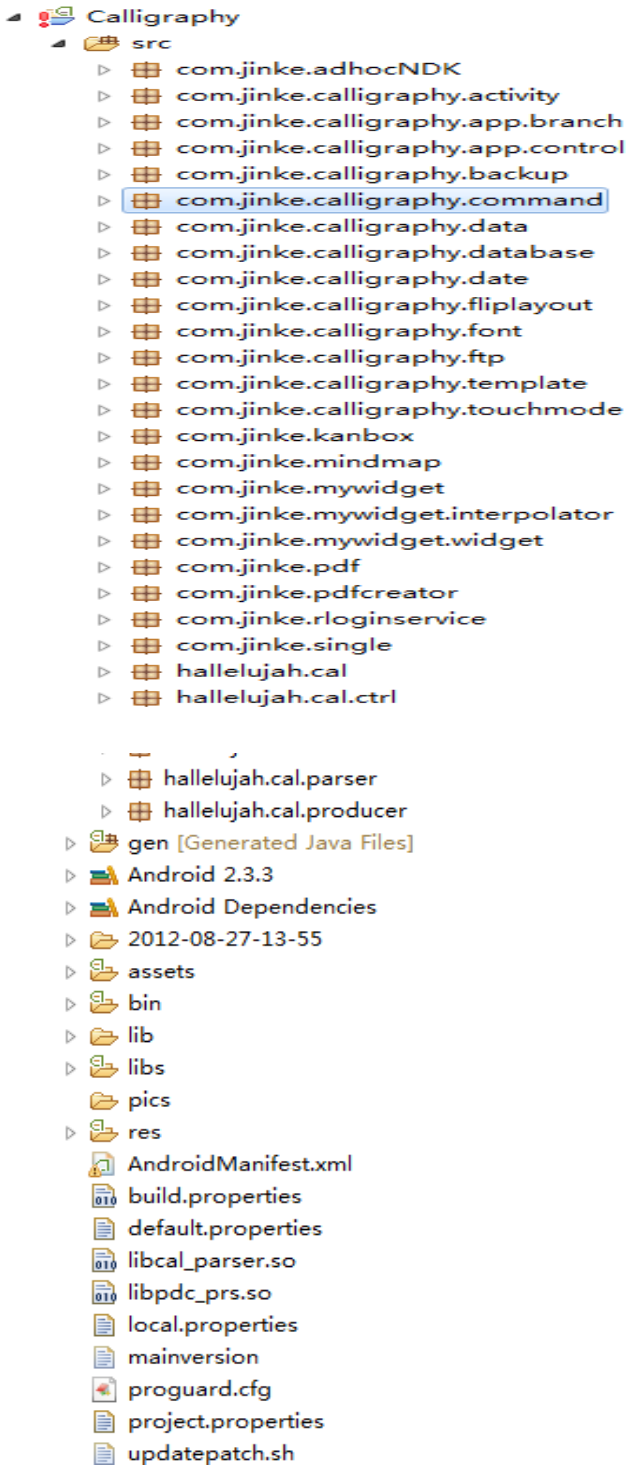
通过 File -> New -> Project 菜单，建立新项目"Android Project"
然后填写必要的参数，如下图所示：（注意这里我勾选的是 Google APIs，你可以选你喜欢的，但你要创建相应的 AVD）
相关参数的说明：
Project Name: 包含这个项目的文件夹的名称。
Package Name: 包名，遵循 JAVA 规范，用包名来区分不同的类是很重要的，我用的是 com.wayou.findlaw。
Activity Name: 这是项目的主类名，这个类将会是 Android 的 Activity 类的子类。一个 Activity 类是一个简单的启动程序和控制程序的类。它可以根据需要创建界面，但不是必须的。
Application Name: 一个易读的标题在你的应用程序上。
在"选择栏"的 "Use default location" 选项，允许你选择一个已存在的项目。
点击 Finish 后，点击 Eclipse 的 Run 菜单选择 Run Configurations…
选择“Android Application”，点击在左上角（按钮像一张纸上有个“+”号）或者双击“Android Application”， 有个新的选项“New_configuration”（可以改为我们喜欢的名字）。
在右侧 Android 面板中点击 Browse…，选择 FindLaw
在 Target 面板的 Automatic 中勾选相应的 AVD，如果没有可用的 AVD 的话，你需要点击右下角的 Manager…，然后新建相应的 AVD。如下图所示：



然后点 Run 按钮即可，运行成功的话会有 Android 的模拟器界面。

2、项目介绍

2.1. 目录：

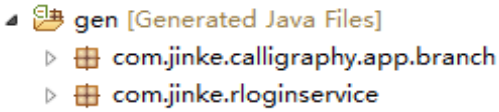


说明：

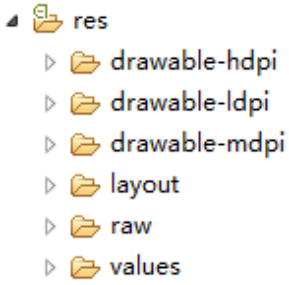
1).这里面 src 目录为.java 类文件目录，每个类必须继承 Activity 类；

2).gen 目录为系统自动生成的目录，不需要开发人员维护；

这里面包括两个文件：



3).res 目录为.xml 文件目录，对应我们 MVC 开发模式中的 view 层。



其中：

a).drawable 文件夹：

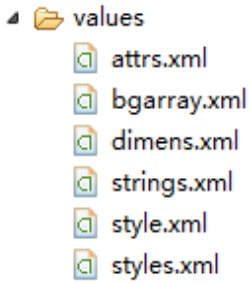
存放的项目中所用到的图片，图片格式为.png 格式。有以上几种规格，以适应各个方面应用。

b).layout 文件夹：

存放 view 视图布局对应的.xml 文件。

c).values 文件夹：

存放 view 视图界面中所用的字符(如文字).

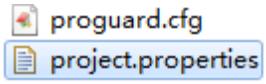


4).AndroidManifest.xml 类似我们 JavaEE 开发中用到的 Struts.config.xml 文件，(回顾：应用 struts 对应的 Action 需要在 Struts.config.xml 文件中进行配置).在这里我们需要对每一个视图文件进行配置

a).自己创建的 activity 类要在这里标注，写绝对路径，也可以写相对路径，建议写绝对路径。

b).每个项目只需要一个 activity 需要这段配置。其他的 activity 配置可以省略不写。

..properties



介绍了 android 项目的版本信息。

模块说明

原笔迹编辑系统架构主要包括了书写编辑界面、控制模块、存储模块。

编辑显示模块主要提供本系统的显示功能以及用户输入的接收功能，该模块处于系统结构的最顶层，直接与用户交互。

逻辑控制模块是核心模块，接收并处理上层的用户事件，识别后作出相应转发。

缓冲模块贯穿书写编辑界面与控制模块之间，能够保持系统稳定性和流畅性。
存储模块除了本地存储还包括与互联网结合的网络存储，分为两部分：与网盘的同步存储以及与服务器端的同步存储。