linux ， Android 基础知识总结

==========

==========

android 系统开发指南（常用环境的搭建和使用）
说明:
有的步骤会用到脚本简化操作，脚本通过 svn 服务器获取：
svn co svn://192.168.2.148/smartphone/td0901/release/images/scripts
用户名为各位的姓名拼音，密码与用户名相同

一　编译 android 源码，制作文件系统
二　 ubuntu 下烧录内核和文件系统

一　编译 android 源码，制作文件系统
1. 开发主线源码位置：
svn://192.168.2.148/smartphone/td0901/trunk/cupcake-jianping　//cupcake 源代码
svn://192.168.2.148/smartphone/td0901/trunk/linux-2.6.28-a1　//内核源代码
2. 打标的源代码位置
svn list svn://192.168.2.148/smartphone/td0901/tag
我们可以通过 svn list　svn://192.168.2.148/smartphone 查看 svn 版本库内核
更多信息请参卡以下文档：
http://192.168.2.148/svn/smartphone/
http://192.168.2.148/svn/smartphone/智能平台开发部资料管理手册 V1.0.doc
http://192.168.2.148/svn/smartphone/linux 下 svn 操作指南及规范.doc
用户名为各位的姓名拼音，密码与用户名相同

3. 编译源码
进入 cupcake 工作拷贝的顶层目录，执行：
. ./make_image15.sh
部分执行结果：
out/target/product/littleton/root/　内核需要使用的 initramfs
out/target/product/littleton/system　文件系统的系统分区
out/target/product/littleton/data/　文件系统数据分区

4. 编译内核

此处内核编译主要针对驱动组之外的同事
1> 设置工具链
内核的 linux-2.6.28-a1/Makefile 中设定了：
CROSS_COMPILE ?= arm-linux-
所以设置 PATH 环境变量，保证能找到正确的工具链
假设工具链位于： /usr/local/marvell-arm-linux-4.1.1/ 设置为：
export PATH:=/usr/local/marvell-arm-linux-4.1.1/bin/:$PATH

2> 更改编译选项（网络启动或者本机启动）
内核顶层目录执行：
make menuconfig
General setup --->
[*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
() Initramfs source file(s) (NEW)
如果需要支持网络启动反选 [] Initial RAM filesystem and RAM disk (initramfs/initrd) support
如果需要支持本地启动选中 [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
设置 () Initramfs source file(s) (NEW) 为 root
拷贝 cupcake 编译结果 out/target/product/littleton/root/ 到内核顶层目录

3> 编译
内核顶层目录执行 make zImage
编译好的内核：
arch/arm/boot/zImage

5. 搭建网络开发环境
1> 安装 nfs 服务器
sudo apt-get install nfs-kernel-server nfs-common
2> 修改 nfs 服务器配置文件/etc/exports ，确保有以下配置项
/nfsroot/rootfs *(rw,no_root_squash,sync)
我们在内核中已经固定，手机通过网络方式启动，默认从 /nfsroot/rootfs
读取文件系统，修改配置项后需要重启 nfs 服务器：
sudo /etc/init.d/nfs-kernel-server restart
3> 配置网络根文件系统
拷贝 out/target/product/littleton/root/ 内容到 /nfsroot/rootfs 目录
拷贝 out/target/product/littleton/system 内容到 /nfsroot/rootfs/system
修改 /nfsroot/rootfs/init.rc 去掉几个 mount 命令
为了使大家的过程，结果统一，可以使用脚本 mkfs.cupcake 完成
在执行 mkfs.cupcake.nfs 脚本前先到 cupcake-jianping 目录下执行: . ./make_env15.sh 设置环境变量,
获取通过手动输入 android 源码的位置，让脚本来设置环境变量。

二 ubuntu 下烧录内核和文件系统
1. 硬件：
手机一台
usb 转串口线一根
usb 转网卡线一根

2. 软件环境

1> tftp 服务器
执行脚本： setup_tftpd.sh 安装和配置 tftp 服务器，我们默认以 /tftpboot
为 tftp 服务器的根目录，需要下载的文件都放在该目录下。

2> 获取待烧录的镜像文件
svn list svn://192.168.2.148/smartphone/td0901/release/images 查看服务器上的
版本情况，通常我们下载最新的,例如，下载 9 月 18 号发布的版本：
svn co svn://192.168.2.148/smartphone/td0901/release/images/images20090918

3> 烧录镜像文件
用以下文件为例，示范通过 tftp 烧写内核和文件系统
内核 zImage0917
系统分区： system0918.img
数据分区 data0918.img
待烧写的以上文件必须存在于 tftp 服务器根目录/tftpboot 下。
具体步骤：
首先连接好硬件设备进入 blob 下载模式
1> blob 起来后按任意键

```
Processing obm parameters...
Can't detect micco. Set PMIC as normal I2C mode.

NAND flash(Manu=0x98 Device=0xba) detected!
Slot 0 Found
get relocation table
Found Main Bad block table at address 0x0f000000, version 0x01
Found Mirror Bad block table at address 0x0efc0000, version 0x01

Consider yourself BLOBed!

blob version 2.0.5-pre3 for Marvell Littleton
Copyright (C) 1999 2000 2001 2002 2003 Jan-Derk Bakker and Erik Mouw
blob comes with ABSOLUTELY NO WARRANTY; read the GNU GPL for details.
This is free software, and you are welcome to redistribute it
under certain conditions; read the GNU GPL for details.
length not align with page size, change to 0x0
Read flash from 0x60000, length 0x0
Done
Autoboot (2 seconds) in progress, press any key to stop ..
Autoboot aborted
Type "help" to get a list of commands
blob>
```

2> 通过 tftp 下载内核到 pc 内存 0x80800000 地址处

```
blob> tftp zImage0917
Begin init ether usbnet!!!
***** Plug-in USB cable & config usbdnet now ******
exit check_usb_connection:1
TFTPing zImage0917*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ OK.
received 6144 blocks (3145156 bytes)
tftp_cmd: file 'zImage0917' loaded via tftp to address 0x80800000.
```

3> 擦除原来的内核分区，0x100000 为分区起始地址，0x300000 为分区长度
blob> nanderase -z 0x100000 0x400000
the current NAND chip does not support Block Unlocking.
Erase 0x300000 length data from flash: 0x100000
Erase flash from 0x100000, length 0x300000
.......................Done

4> 烧写内存 0x80800000 开始 实际长度为 3145156 的内核数据到起始地址为 0x100000 的内核分区
blob> nandwrite -z 0x80800000 0x100000 3145156
the current NAND chip does not support Block Unlocking.
Write 0x2ffdc4 length data from RAM: 0x80800000 to flash: 0x100000
Write flash from 0x100000, length 0x2ffdc4
Erase flash from 0x100000, length 0x300000
.......................Done
.......................Done

5> 下载系统分区镜像文件到 pc 内存 0x80800000 地址处
blob> tftp system0918.img
TFTPing system0918.img*^^
^^
^^
^^
^^^ OK.
received 113138 blocks (57925824 bytes)
tftp_cmd: file 'system0918.img' loaded via tftp to address 0x80800000.

6> 擦除原来的 flash 系统分区
blob> nanderase -z 0x500000 0x4000000
the current NAND chip does not support Block Unlocking.
Erase 0x3e0f800 length data from flash: 0x400000
Erase flash from 0x400000, length 0x3e0f800
....
....
....
.........................Done

7> 烧写数据到 flash 系统分区
blob> nandwrite -y 0x80800000 0x500000 57925824
the current NAND chip does not support Block Unlocking.
Write 0x373e0c0 length data from RAM: 0x80800000 to flash: 0x400000
Write flash from 0x400000, length 0x3591800
Erase flash from 0x400000, length 0x3591800
....
....
....Done
....
....
....Done

8> 下载数据分区镜像文件到 pc 内存 0x80800000 地址处

```
blob> tftp data0918.img
TFTPing data0918.img*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ OK.
received 33992 blocks (17402880 bytes)
tftp_cmd: file 'data0918.img' loaded via tftp to address 0x80800000.
blob>
```

9> 擦除原来的 flash 数据分区

```
blob> nanderase -z 0x4500000 0xBB00000
the current NAND chip does not support Block Unlocking.
Erase 0xa81f000 length data from flash: 0x4400000
Erase flash from 0x4400000, length 0xa81f000
................................................................................
................................................................................
................................................................................
................................................................................
................................................Done
```

10> 烧写数据镜像到 flash 数据分区

```
blob> nandwrite -y 0x80800000 0x4500000 17402880
the current NAND chip does not support Block Unlocking.
Write 0x1098c00 length data from RAM: 0x80800000 to flash: 0x4400000
Write flash from 0x4400000, length 0x1018000
Erase flash from 0x4400000, length 0x1018000
.............................................................................Done
.............................................................................Done
blob>
```

flash 分区图：

```
*******************************************
*         *          *         *           *
*  blob   *  kernel  *  system *   data    *
*         *          *         *           *
*******************************************
```

```
nanderase -z 0x100000 0x400000
tftp zImage
nandwrite -z 0x80800000 0x100000 <zImage-actual-length>
```

烧写 system.img：

```
nanderase -z 0x500000 0x4000000
tftp system.img
nandwrite -y 0x80800000 0x500000 <system.img actual length>
```

烧写 userdata.img ：

```
nanderase -z 0x4500000 0xBB00000
tftp userdata.img
nandwrite -y 0x80800000 0x4500000 <userdata.img actual length>
```

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

涉及的内容：
svn 服务器的使用
android 的编译系统，原理，工具链，辅助工具，参数等，环境变量，怎样单独添加编译一个单独的模块等。
android 的编译结果：文件系统分析
文件系统的使用，启动流程
设置模块流程分析
==========

==========

1. Android 编译系统分析
编译脚本及系统变量
build/envsetup.sh 脚本分析
在编译源代码之前通常需要在 android 源代码顶层目录执行 . ./build/envsetup.sh 目的是为了使用
脚本 envsetup.sh 里面定义了一些函数：
function help()
function get_abs_build_var()
function get_build_var()
function check_product()
function check_variant()
function setpaths()
function printconfig()
function set_stuff_for_environment()
function set_sequence_number()
function settitle()
function choosetype()
function chooseproduct()
function choosevariant()
function tapas()
function choosecombo()
function print_lunch_menu()
function lunch()
function gettop
function m()
function findmakefile()
function mm()
function mmm()
function croot()
function pid()
function gdbclient()
function jgrep()
function cgrep()
function resgrep()
function getprebuilt
function tracedmdump()
function runhat()
function getbugreports()
function startviewserver()
function stopviewserver()

function isviewserverstarted()
function smoketest()
function runtest()
function runtest_py()
function godir ()

choosecombo 命令分析：
function choosecombo()
{
    choosesim $1
    echo
    echo
    choosetype $2

    echo
    echo
    chooseproduct $3

    echo
    echo
    choosevariant $4

    echo
    set_stuff_for_environment
    printconfig
}
会依次进行如下选择：
Build for the simulator or the device?
    1. Device
    2. Simulator
Which would you like? [1]

Build type choices are:
    1. release
    2. debug
Which would you like? [1]

Product choices are:
    1. emulator
    2. generic
    3. sim
    4. littleton
You can also type the name of a product if you know it.
Which would you like? [littleton]

Variant choices are:
    1. user
    2. userdebug
    3. eng
Which would you like? [eng] user
默认选择以后会出现：
TARGET_PRODUCT=littleton
TARGET_BUILD_VARIANT=user

TARGET_SIMULATOR=false
TARGET_BUILD_TYPE=release
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=
＝＝＝＝＝＝＝＝＝＝
function chooseproduct()函数分析：
choices=(`/bin/ls build/target/board/*/BoardConfig.mk vendor/*/*/BoardConfig.mk 2> /dev/null`)
读取 build/target/board/* 目录下的板配置文件：BoardConfig.mk
读取 vendor/*/*/目录下的板配置文件：BoardConfig.mk
choices 的值为：
build/target/board/emulator/BoardConfig.mk
build/target/board/generic/BoardConfig.mk
build/target/board/sim/BoardConfig.mk
vendor/marvell/littleton/BoardConfig.mk
经过：

```
for choice in ${choices[@]}
do
    # The product name is the name of the directory containing
    # the makefile we found, above.
    prodlist=(${prodlist[@]} `dirname ${choice} | xargs basename`)
done
```

的处理，prodlist 的值为：
emulator generic sim littleton
所以选择菜单为：
Product choices are:

1. emulator
2. generic
3. sim
4. littleton

如果选择 4，那么 TARGET_PRODUCT 被赋值为： littleton。

```
board_config_mk := \
  $(strip $(wildcard \
            $(SRC_TARGET_DIR)/board/$(TARGET_DEVICE)/BoardConfig.mk \
          vendor/*/$(TARGET_DEVICE)/BoardConfig.mk \
    ))
```

怎样添加一个模块

```
LOCAL_PATH:= $(call my-dir)
#编译静态库
include $(CLEAR_VARS)
LOCAL_MODULE = libhellos
LOCAL_CFLAGS = $(L_CFLAGS)
LOCAL_SRC_FILES = hellos.c
LOCAL_C_INCLUDES = $(INCLUDES)
LOCAL_SHARED_LIBRARIES := libcutils
```

```
LOCAL_COPY_HEADERS_TO := libhellos
LOCAL_COPY_HEADERS := hellos.h
include $(BUILD_STATIC_LIBRARY)

#编译动态库
include $(CLEAR_VARS)
LOCAL_MODULE = libhellod
LOCAL_CFLAGS = $(L_CFLAGS)
LOCAL_SRC_FILES = hellod.c
LOCAL_C_INCLUDES = $(INCLUDES)
LOCAL_SHARED_LIBRARIES := libcutils
LOCAL_COPY_HEADERS_TO := libhellod
LOCAL_COPY_HEADERS := hellod.h
include $(BUILD_SHARED_LIBRARY)

BUILD_TEST=true
ifeq ($(BUILD_TEST),true)
#使用静态库
include $(CLEAR_VARS)
LOCAL_MODULE := hellos
LOCAL_STATIC_LIBRARIES := libhellos
LOCAL_SHARED_LIBRARIES :=
LOCAL_LDLIBS += -ldl
LOCAL_CFLAGS := $(L_CFLAGS)
LOCAL_SRC_FILES := mains.c
LOCAL_C_INCLUDES := $(INCLUDES)
include $(BUILD_EXECUTABLE)

#使用动态库
include $(CLEAR_VARS)
LOCAL_MODULE := hellod
LOCAL_MODULE_TAGS := debug
LOCAL_SHARED_LIBRARIES := libc libcutils libhellod
LOCAL_LDLIBS += -ldl
LOCAL_CFLAGS := $(L_CFLAGS)
LOCAL_SRC_FILES := maind.c
LOCAL_C_INCLUDES := $(INCLUDES)
include $(BUILD_EXECUTABLE)
endif # ifeq ($(WPA_BUILD_SUPPLICANT),true)

########################
#local_target_dir := $(TARGET_OUT)/etc/wifi
#include $(CLEAR_VARS)
#LOCAL_MODULE := wpa_supplicant.conf
#LOCAL_MODULE_TAGS := user
#LOCAL_MODULE_CLASS := ETC
#LOCAL_MODULE_PATH := $(local_target_dir)
#LOCAL_SRC_FILES := $(LOCAL_MODULE)
#include $(BUILD_PREBUILT)
########################
系统变量解析
LOCAL_MODULE        一 编译的目标对象
```

LOCAL_SRC_FILES　－ 编译的源文件
LOCAL_C_INCLUDES － 需要包含的头文件目录
LOCAL_SHARED_LIBRARIES － 链接时需要的外部库
LOCAL_PRELINK_MODULE　－ 是否需要 prelink 处理
BUILD_SHARED_LIBRARY　－ 指明要编译成动态库

LOCAL_PATH - 编译时的目录
$(call 目录，目录….) 目录引入操作符
如该目录下有个文件夹名称 src，则可以这样写 $(call src)，那么就会得到 src 目录的完整路径

include $(CLEAR_VARS) -清除之前的一些系统变量
CLEAR_VARS:= $(BUILD_SYSTEM)/clear_vars.mk
在 build/core/config.mk 定义 CLEAR_VARS:= $(BUILD_SYSTEM)/clear_vars.mk
通过 include 包含自定义的.mk 文件（即是自定义编译规则）或是引用系统其他的.mk 文件（系统定义的编译规则）。

LOCAL_SRC_FILES － 编译的源文件
可以是.c, .cpp, .java, .S（汇编文件）或是.aidl 等格式
不同的文件用空格隔开。如果编译目录子目录，采用相对路径，如子目录/文件名。也可以通过$(call 目录)，指明编译某目录
下所有.c/.cpp/.java/.S/ .aidl 文件.追加文件 LOCAL_SRC_FILES += 文件

LOCAL_C_INCLUDES － 需要包含的头文件目录
可以是系统定义路径，也可以是相对路径. 如该编译目录下有个 include 目录，写法是 include/*.h

LOCAL_SHARED_LIBRARIES　－ 链接时需要的外部共享库
LOCAL_STATIC_LIBRARIES　－ 链接时需要的外部外部静态
LOCAL_JAVA_LIBRARIES　　加入 jar 包

LOCAL_MODULE － 编译的目标对象
module 是指系统的 native code，通常针对 c,c++代码
./system/core/sh/Android.mk:32:LOCAL_MODULE:= sh
./system/core/libcutils/Android.mk:71:LOCAL_MODULE := libcutils
./system/core/cpio/Android.mk:9:LOCAL_MODULE := mkbootfs
./system/core/mkbootimg/Android.mk:8:LOCAL_MODULE := mkbootimg
./system/core/toolbox/Android.mk:61:LOCAL_MODULE:= toolbox
./system/core/logcat/Android.mk:10:LOCAL_MODULE:= logcat
./system/core/adb/Android.mk:65:LOCAL_MODULE := adb
./system/core/adb/Android.mk:125:LOCAL_MODULE := adbd
./system/core/init/Android.mk:20:LOCAL_MODULE:= init
./system/core/vold/Android.mk:24:LOCAL_MODULE:= vold
./system/core/mountd/Android.mk:13:LOCAL_MODULE:= mountd

LOCAL_PACKAGE_NAME
Java 应用程序的名字用该变量定义
./packages/apps/Music/Android.mk:9:LOCAL_PACKAGE_NAME := Music
./packages/apps/Browser/Android.mk:14:LOCAL_PACKAGE_NAME := Browser

./packages/apps/Settings/Android.mk:8:LOCAL_PACKAGE_NAME := Settings
./packages/apps/Stk/Android.mk:10:LOCAL_PACKAGE_NAME := Stk
./packages/apps/Contacts/Android.mk:10:LOCAL_PACKAGE_NAME := Contacts
./packages/apps/Mms/Android.mk:8:LOCAL_PACKAGE_NAME := Mms
./packages/apps/Camera/Android.mk:8:LOCAL_PACKAGE_NAME := Camera
./packages/apps/Phone/Android.mk:11:LOCAL_PACKAGE_NAME := Phone
./packages/apps/VoiceDialer/Android.mk:8:LOCAL_PACKAGE_NAME := VoiceDialer

BUILD_SHARED_LIBRARY ― 指明要编译成动态库。
编译的目标，用 include 操作符
UILD_STATIC_LIBRARY 来指明要编译成静态库。
如果是 java 文件的话，会用到系统的编译脚本 host_java_library.mk，用 BUILD_PACKAGE 来指明。三个编译
――――――――――――――――――
include $(BUILD_STATIC_LIBRARY)
BUILD_STATIC_LIBRARY:= $(BUILD_SYSTEM)/static_library.mk
――――――――――――――――――
include $(BUILD_SHARED_LIBRARY)
./build/core/config.mk:50:BUILD_SHARED_LIBRARY:= $(BUILD_SYSTEM)/shared_library.mk
――――――――――――――――――
include $(BUILD_HOST_SHARED_LIBRARY)
BUILD_HOST_SHARED_LIBRARY:= $(BUILD_SYSTEM)/host_shared_library.mk
――――――――――――――――――
include $(BUILD_EXECUTABLE)
build/core/config.mk:51:BUILD_EXECUTABLE:= $(BUILD_SYSTEM)/executable.mk
――――――――――――――――――
include $(BUILD_HOST_EXECUTABLE)
./build/core/config.mk:53:BUILD_HOST_EXECUTABLE:=
$(BUILD_SYSTEM)/host_executable.mk
――――――――――――――――――
BUILD_HOST_JAVA_LIBRARY:= $(BUILD_SYSTEM)/host_java_library.mk
――――――――――――――――――
BUILD_JAVA_LIBRARY
./build/core/config.mk:58:BUILD_JAVA_LIBRARY:= $(BUILD_SYSTEM)/java_library.mk
――――――――――――――――――
BUILD_STATIC_JAVA_LIBRARY 编译静态 JAVA 库
./build/core/config.mk:59:BUILD_STATIC_JAVA_LIBRARY:=
$(BUILD_SYSTEM)/static_java_library.mk
――――――――――――――――――
BUILD_HOST_JAVA_LIBRARY 编译本机用的 JAVA 库
./build/core/config.mk:60:BUILD_HOST_JAVA_LIBRARY:=
$(BUILD_SYSTEM)/host_java_library.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

＝＝＝＝＝＝＝＝＝＝＝＝
LOCAL_PRELINK_MODULE
Prelink 利用事先链接代替运行时链接的方法来加速共享库的加载，它不仅可以加快起动速度，还可以减少部分内存开销，
是各种 Linux 架构上用于减少程序加载时间、缩短系统启动时间和加快应用程序启动的很受欢迎的一个工具。程序运行时的
动态链接尤其是重定位(relocation)的开销对于大型系统来说是很大的。
动态链接和加载的过程开销很大，并且在大多数的系统上, 函数库并不会常常被更动, 每次程序被执行时所进行的链接
动作都是完全相同的，对于嵌入式系统来说尤其如此。因此，这一过程可以改在运行时之前就可以预先处理好，即花一些时间
利用 Prelink 工具对动态共享库和可执行文件进行处理，修改这些二进制文件并加入相应的重定位等信息，节约了本来在程序
启动时的比较耗时的查询函数地址等工作，这样可以减少程序启动的时间，同时也减少了内存的耗用。
Prelink 的这种做法当然也有代价：每次更新动态共享库时，相关的可执行文件都需要重新执行一遍 Prelink 才能保
证有效，因为新的共享库中的符号信息、地址等很可能与原来的已经不同了，这就是为什么 android framework 代码一改动，
这时候就会导致相关的应用程序重新被编译。
这种代价对于嵌入式系统的开发者来说可能稍微带来一些复杂度，不过好在对用户来说几乎是可以忽略的。
－－－－－－－－－－－－－－－－－－－－
变量设置为 false 那么将不做 prelink 操作
LOCAL_PRELINK_MODULE := false
默认是需要 prlink 的，同时需要在 build/core/prelink-linux-arm.map 中加入
libhellod.so 0x96000000
这个 map 文件好像是制定动态库的地址的，在前面注释上面有一些地址范围的信息，注意库与库之间的间隔数，
如果指定不好的话编译的时候会提示说地址空间冲突的问题。另外，注意排序，这里要把数大的放到前面去，
按照大小降序排序。
解析 LOCAL_PRELINK_MODULE 变量
build/core/dynamic_binary.mk:94:ifeq ($(LOCAL_PRELINK_MODULE),true)
ifeq ($(LOCAL_PRELINK_MODULE),true)
$(prelink_output): $(prelink_input) $(TARGET_PRELINKER_MAP) $(APRIORI)

```
    $(transform-to-prelinked)
transform-to-prelinked 定义：
./build/core/definitions.mk:1002:define transform-to-prelinked
define transform-to-prelinked
@mkdir -p $(dir $@)
@echo "target Prelink: $(PRIVATE_MODULE) ($@)"
$(hide) $(APRIORI) \
          --prelinkmap $(TARGET_PRELINKER_MAP) \
          --locals-only \
         --quiet \
           $< \
         --output $@
endef
./build/core/config.mk:183:APRIORI :=
$(HOST_OUT_EXECUTABLES)/apriori$(HOST_EXECUTABLE_SUFFIX)
prelink 工具不是常用的 prelink 而是 apriori，其源代码位于” <your_android>/build/tools/apriori”
参考文档：
动态库优化——Prelink（预连接）技术
http://www.eefocus.com/article/09-04/71629s.html


==========
LOCAL_ARM_MODE := arm
目前 Android 大部分都是基于 Arm 处理器的，Arm 指令用两种模式 Thumb(每条指令两个字节)
和 arm 指令（每条指令四个字节）

LOCAL_CFLAGS += -O3 -fstrict-aliasing -fprefetch-loop-arrays
通过设定编译器操作，优化级别，-O0 表示没有优化,-O1 为缺省值，-O3 优化级别最高
LOCAL_CFLAGS += -W -Wall
LOCAL_CFLAGS += -fPIC -DPIC
LOCAL_CFLAGS += -O2 -g -DADB_HOST=1    -Wall -Wno-unused-parameter
LOCAL_CFLAGS += -D_XOPEN_SOURCE -D_GNU_SOURCE -DSH_HISTORY
LOCAL_CFLAGS += -DUSEOVERLAY2
根据条件选择相应的编译参数
ifeq ($(TARGET_ARCH),arm)
LOCAL_CFLAGS += -DANDROID_GADGET=1
LOCAL_CFLAGS :=     $(PV_CFLAGS)
endif
ifeq ($(TARGET_BUILD_TYPE),release)
  LOCAL_CFLAGS += -O2
endif

LOCAL_LDLIBS := -lpthread
LOCAL_LDLIBS += -ldl

ifdef USE_MARVELL_MVED
LOCAL_WHOLE_STATIC_LIBRARIES += lib_il_mpeg4aspdecmved_wmmx2lnx
lib_il_h264decmved_wmmx2lnx
LOCAL_SHARED_LIBRARIES += libMrvlMVED
else
LOCAL_WHOLE_STATIC_LIBRARIES += lib_il_h264dec_wmmx2lnx
lib_il_mpeg4aspdec_wmmx2lnx
```

endif
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
其他一些变量和脚本：
HOST_JNILIB_SUFFIX
LOCAL_MODULE_SUFFIX
LOCAL_MODULE_SUFFIX := $(HOST_JNILIB_SUFFIX)
HOST_GLOBAL_LDFLAGS
TARGET_GLOBAL_LDFLAGS
PRIVATE_LDFLAGS
LOCAL_LDLIBS
LOCAL_C_INCLUDES
LOCAL_STATIC_LIBRARIES
LOCAL_STATIC_LIBRARIES += codecJPDec_WMMX2LNX miscGen_WMMX2LNX
LOCAL_SHARED_LIBRARIES
LOCAL_SHARED_LIBRARIES += libMrvlIPP
LOCAL_SHARED_LIBRARIES += $(common_SHARED_LIBRARIES)
LOCAL_SHARED_LIBRARIES += libMrvlIPP
LOCAL_SHARED_LIBRARIES += libdl
ifeq ($(TARGET_PRODUCT),littleton)
LOCAL_C_INCLUDES += vendor/marvell/littleton/m2d \
LOCAL_SHARED_LIBRARIES += libOmxCore
endif

vendor/marvell/littleton/littleton.mk:27:PRODUCT_NAME := littleton
vendor/marvell/littleton/littleton.mk:28:PRODUCT_DEVICE := littleton
vendor/marvell/littleton/AndroidProducts.mk:13:	$(LOCAL_DIR)/littleton.mk

vendor/sample/products/sample_addon.mk:40:PRODUCT_NAME := sample_addon
vendor/htc/dream-open/htc_dream.mk:6:PRODUCT_NAME := htc_dream
./vendor/htc/dream-open/htc_dream.mk:7:PRODUCT_DEVICE := dream-open
./vendor/htc/dream-open/AndroidProducts.mk:3:	$(LOCAL_DIR)/htc_dream.mk

build/target/product/generic.mk:26:PRODUCT_NAME := generic
build/target/product/generic_with_google.mk:20:PRODUCT_NAME := generic_with_google
build/target/product/min_dev.mk:6:PRODUCT_NAME := min_dev
build/target/product/core.mk:2:PRODUCT_NAME :=
build/target/product/sim.mk:7:PRODUCT_NAME := sim
build/target/product/sdk.mk:37:PRODUCT_NAME := sdk
build/tools/buildinfo.sh:20:echo "ro.product.name=$PRODUCT_NAME"

lunch sample_addon-eng
lunch htc_dream-eng
lunch generic-eng
lunch 1
lunch sim-eng
TARGET_BUILD_TYPE=release
lunch 2
TARGET_BUILD_TYPE=debug
lunch generic-user

.PHONY: systemtarball-nodeps
systemtarball-nodeps: $(FS_GET_STATS) \
	$(filter-out systemtarball-nodeps stnod,$(MAKECMDGOALS))

```
    $(build-systemtarball-target)

.PHONY: stnod
stnod: systemtarball-nodeps
systemimage-nodeps snod

./core/main.mk:BUILD_SYSTEM := $(TOPDIR)build/core

./core/main.mk:include $(BUILD_SYSTEM)/config.mk
./core/main.mk:include $(BUILD_SYSTEM)/cleanbuild.mk
./core/main.mk:include $(BUILD_SYSTEM)/version_defaults.mk
./core/main.mk:include $(BUILD_SYSTEM)/definitions.mk
./core/main.mk:include $(BUILD_SYSTEM)/Makefile
./core/static_java_library.mk:include $(BUILD_SYSTEM)/java_library.mk
./core/host_java_library.mk:include $(BUILD_SYSTEM)/base_rules.mk
./core/executable.mk:include $(BUILD_SYSTEM)/dynamic_binary.mk
./core/java_library.mk:include $(BUILD_SYSTEM)/java.mk
./core/binary.mk:include $(BUILD_SYSTEM)/base_rules.mk
./core/raw_executable.mk:include $(BUILD_SYSTEM)/binary.mk
./core/prebuilt.mk:include $(BUILD_SYSTEM)/base_rules.mk
./core/host_executable.mk:include $(BUILD_SYSTEM)/binary.mk

./core/combo/select.mk:$(combo_target)PRELINKER_MAP := $(BUILD_SYSTEM)/prelink-
$(combo_os_arch).map
./core/shared_library.mk:include $(BUILD_SYSTEM)/dynamic_binary.mk
./core/config.mk:include $(BUILD_SYSTEM)/pathmap.mk
./core/config.mk:BUILD_COMBOS:= $(BUILD_SYSTEM)/combo
./core/config.mk:CLEAR_VARS:= $(BUILD_SYSTEM)/clear_vars.mk
./core/config.mk:BUILD_HOST_STATIC_LIBRARY:= $(BUILD_SYSTEM)/host_static_library.mk
./core/config.mk:BUILD_HOST_SHARED_LIBRARY:=
$(BUILD_SYSTEM)/host_shared_library.mk
./core/config.mk:BUILD_STATIC_LIBRARY:= $(BUILD_SYSTEM)/static_library.mk
./core/config.mk:BUILD_RAW_STATIC_LIBRARY := $(BUILD_SYSTEM)/raw_static_library.mk
./core/config.mk:BUILD_SHARED_LIBRARY:= $(BUILD_SYSTEM)/shared_library.mk
./core/config.mk:BUILD_EXECUTABLE:= $(BUILD_SYSTEM)/executable.mk
./core/config.mk:BUILD_RAW_EXECUTABLE:= $(BUILD_SYSTEM)/raw_executable.mk
./core/config.mk:BUILD_HOST_EXECUTABLE:= $(BUILD_SYSTEM)/host_executable.mk
./core/config.mk:BUILD_PACKAGE:= $(BUILD_SYSTEM)/package.mk
./core/config.mk:BUILD_HOST_PREBUILT:= $(BUILD_SYSTEM)/host_prebuilt.mk
./core/config.mk:BUILD_PREBUILT:= $(BUILD_SYSTEM)/prebuilt.mk
./core/config.mk:BUILD_MULTI_PREBUILT:= $(BUILD_SYSTEM)/multi_prebuilt.mk
./core/config.mk:BUILD_JAVA_LIBRARY:= $(BUILD_SYSTEM)/java_library.mk
./core/config.mk:BUILD_STATIC_JAVA_LIBRARY:= $(BUILD_SYSTEM)/static_java_library.mk
./core/config.mk:BUILD_HOST_JAVA_LIBRARY:= $(BUILD_SYSTEM)/host_java_library.mk
./core/config.mk:BUILD_DROIDDOC:= $(BUILD_SYSTEM)/droiddoc.mk
./core/config.mk:BUILD_COPY_HEADERS := $(BUILD_SYSTEM)/copy_headers.mk
./core/config.mk:BUILD_KEY_CHAR_MAP := $(BUILD_SYSTEM)/key_char_map.mk
./core/config.mk:HOST_JDK_TOOLS_JAR:= $(shell $(BUILD_SYSTEM)/find-jdk-tools-jar.sh)
./core/version_defaults.mk:INTERNAL_BUILD_ID_MAKEFILE := $(wildcard
$(BUILD_SYSTEM)/build_id.mk)

./core/config.mk:include $(BUILD_SYSTEM)/envsetup.mk
./core/config.mk:include $(BUILD_SYSTEM)/combo/select.mk
```

./core/config.mk:include $(BUILD_SYSTEM)/combo/select.mk
./core/config.mk:include $(BUILD_SYSTEM)/combo/javac.mk
./core/product_config.mk:include $(BUILD_SYSTEM)/node_fns.mk
./core/product_config.mk:include $(BUILD_SYSTEM)/product.mk
./core/product_config.mk:include $(BUILD_SYSTEM)/device.mk
./core/dynamic_binary.mk:include $(BUILD_SYSTEM)/binary.mk
./core/host_static_library.mk:include $(BUILD_SYSTEM)/binary.mk
./core/java.mk:include $(BUILD_SYSTEM)/base_rules.mk
./core/host_shared_library.mk:include $(BUILD_SYSTEM)/binary.mk
./core/key_char_map.mk:include $(BUILD_SYSTEM)/base_rules.mk
./core/package.mk:include $(BUILD_SYSTEM)/java.mk
./core/static_library.mk:include $(BUILD_SYSTEM)/binary.mk
./core/definitions.mk:include $(BUILD_SYSTEM)/distdir.mk
./core/envsetup.mk:include $(BUILD_SYSTEM)/product_config.mk
./tools/apicheck/Android.mk:include $(BUILD_SYSTEM)/base_rules.mk
./tools/dexpreopt/Android.mk:include $(BUILD_SYSTEM)/host_prebuilt.mk

COMMON_GLOBAL_CFLAGS:= -DANDROID -fmessage-length=0 -W -Wall -Wno-unused
COMMON_DEBUG_CFLAGS:=
COMMON_RELEASE_CFLAGS:= -DNDEBUG -UDEBUG
COMMON_PACKAGE_SUFFIX := .zip
COMMON_JAVA_PACKAGE_SUFFIX := .jar
COMMON_ANDROID_PACKAGE_SUFFIX := .apk

ACP := $(HOST_OUT_EXECUTABLES)/acp$(HOST_EXECUTABLE_SUFFIX)
AIDL := $(HOST_OUT_EXECUTABLES)/aidl$(HOST_EXECUTABLE_SUFFIX)
MKBOOTFS := $(HOST_OUT_EXECUTABLES)/mkbootfs$(HOST_EXECUTABLE_SUFFIX)
MKBOOTIMG := $(HOST_OUT_EXECUTABLES)/mkbootimg$(HOST_EXECUTABLE_SUFFIX)

MKYAFFS2 :=
$(HOST_OUT_EXECUTABLES)/mkyaffs2image$(HOST_EXECUTABLE_SUFFIX)
APICHECK := $(HOST_OUT_EXECUTABLES)/apicheck$(HOST_EXECUTABLE_SUFFIX)
FS_GET_STATS :=
$(HOST_OUT_EXECUTABLES)/fs_get_stats$(HOST_EXECUTABLE_SUFFIX)
MKEXT2IMG := $(HOST_OUT_EXECUTABLES)/genext2fs$(HOST_EXECUTABLE_SUFFIX)
MKEXT2BOOTIMG := external/genext2fs/mkbootimg_ext2.sh
MKTARBALL := build/tools/mktarball.sh
DX := $(HOST_OUT_EXECUTABLES)/dx
LOCALIZE := $(HOST_OUT_EXECUTABLES)/localize$(HOST_EXECUTABLE_SUFFIX)
HOST_GLOBAL_LDFLAGS
TARGET_GLOBAL_LDFLAGS
PRIVATE_LDFLAGS

build/core/combo/linux-arm.mk:16:$(combo_target)NO_UNDEFINED_LDFLAGS := -Wl,--no-
undefined
save_CFLAGS="$CFLAGS -g -mabi=aapcs-linux"
LDFLAGS='$LDFLAGS -lX11 -lxml2 -lXdmcp -lXau -lexpat -lXrender -lXft -lfontconfig -
lfreetype -lz'
--without-libtiff " # --with-gdktarget=directfb"
LDFLAGS=" -Wl,-rpath-link=$LD_LIBRARY_PATH -L$PREFIX/lib ${env_LDFLAGS}
${save_LDFLAGS}"
./vendor/marvell/external/alsa/alsa-lib/src/Mdroid.mk:43:LOCAL_CFLAGS += -mabi=aapcs-linux
./vendor/marvell/external/alsa/alsa-tools/Mdroid.mk:8:LOCAL_CFLAGS += -mabi=aapcs-linux

./vendor/marvell/littleton/libaudio/Mdroid.mk:22:LOCAL_CPPFLAGS += -mabi=aapcs-linux
./external/wpa_supplicant/Android.mk:35:L_CFLAGS += -mabi=aapcs-linux
./system/wlan/ti/sta_dk_4_0_4_32/CUDK/tiwlan_loader/Android.mk:88:LOCAL_CFLAGS = -Wall -Wstrict-prototypes
$(CLI_DEBUGFLAGS) -D__LINUX__ $(DK_DEFINES) -mabi=aapcs-linux
./kernel/arch/arm/Makefile
ifeq ($(CONFIG_AEABI),y)
CFLAGS_ABI :=-mabi=aapcs-linux -mno-thumb-interwork
else
CFLAGS_ABI :=$(call cc-option,-mapcs-32,-mabi=apcs-gnu) $(call cc-option,-mno-thumb-interwork,)
endif
# Need -Uarm for gcc < 3.x
KBUILD_CFLAGS +=$(CFLAGS_ABI) $(arch-y) $(tune-y) $(call cc-option,-mshort-load-bytes,$(call cc-option,
-malignment-traps,)) -msoft-float -Uarm
KBUILD_AFLAGS +=$(CFLAGS_ABI) $(arch-y) $(tune-y) -msoft-float
CFLAGS="${temp_CFLAGS} -I/include -I/usr/include -I/usr/X11R7/include"
LDFLAGS="${temp_LDFLAGS} -L/lib -L/usr/lib -L/usr/X11R7/lib"

Android Build System
http://www.cublog.cn/u/8059/showart_1420446.html
==========

==========
2. 文件系统分析
2.1 文件系统概述
2.2 ext2 ,ext3 文件系统
2.3 jffs，jffs2 文件系统
2.4 yafss，yaffs2 文件系统
2.5 虚拟文件系统（sysfs，proc，tsmpfs 等）
2.6 一些必要重要的系统文件 （ /etc/fstab ，inittab，init.rc 等）
2.7 制作文件系统

2.1 文件系统概述
文件系统（File system）指代贮存在计算机上的文件和目录。文件系统可以有不同的格式，叫做文件系统类型（file system types）。
这些格式决定信息是如何被贮存为文件和目录。Linux 支持多种文件系统，包括 sysfs，proc，tmpfs,ext2，ext3，cramfs，ramfs,nfs,
vfat,jffs,jffs2,yaffs,yaffs2 等，为了对各类文件系统进行统一管理，Linux 引入了虚拟文件系统 VFS(Virtual File System)，
为各类文件系统提供一个统一的操作界面和应用编程接口。Linux 启动时，第一个必须挂载的是根文件系统；若系统不能从指定设备上挂载根文件

系统，则系统会出错而退出启动。之后可以自动或手动挂载其他的文件系统。因此，一个系统中可以同时存在不同的文件系统。

不同的文件系统类型有不同的特点，因而根据存储设备的硬件特性、系统需求等有不同的应用场合。在嵌入式 Linux 应用中，主要的存储设备
为 RAM(DRAM, SDRAM)和 ROM(常采用 FLASH 存储器)，常用的基于存储设备的文件系统类型包括：jffs2, yaffs, cramfs, romfs,
ramdisk, ramfs/tmpfs 等。

2.2 ext2 ,ext3 文件系统
在异常断电或系统崩溃（又称不洁系统关机，unclean system shutdown）发生时，每个在系统上挂载了的 ext2 文件系统必须要使用 e2fsck
程序来检查其一致性。这是一个很费时的过程，特别是在检查包含大量文件的庞大文件卷时，它会大大耽搁引导时间。在这期间，文件卷上的所有数据
都不能被访问。由 ext3 文件系统提供的登记报表方式意味着不洁系统关机后没必要再进行此类文件系统检查。使用 ext3 系统时，一致性检查只在
某些罕见的硬件失效（如硬盘驱动器失效）情况下才发生。
参考文档：
Linux ext2/ext3 文件系统详解
http://www.blueidea.com/computer/system/2008/5536.asp
Linux EXT2 文件系统结构分析(详情见附件)
http://chenguang.blog.51cto.com/350944/69655
Ext2 文件系统的硬盘布局
http://www.ibm.com/developerworks/cn/linux/filesystem/ext2/

2.3 jffs，jffs2 文件系统
jffs2 文件系统制作工具 mkfs.jffs2
1．从网上下载：
mkjffs2-arm.rar http://blogimg.chinaunix.net/blog/upfile2/080701192924.rar
mkjffs2-pc.rar http://blogimg.chinaunix.net/blog/upfile2/080701193005.rar
2．通过源代码获得 mkfs.jffs2
源码下载地址: ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-1.0.1.tar.gz
编译的过程中缺少 sys/acl.h 文件，ubunt－8.10 通过下面命令安装
sudo apt-get install libacl1-dev
制作 jffs2 文件系统
mkfs.jffs2 -r rootfs/ -o rootfs-jffs2.img -e 0x4000 --pad=0x500000 -s 0x200 -n
各参数的意义：
(1)-r ：指定要做成 image 的源文件夹.
(2)-o：指定输出 image 文档的文件名.
(3)-e：每一块要抹除的 block size，默认值是 64KB.要注意，不同的 flash, 其 block size 会不一样
(4)--pad （-p）：用 16 进制來表示所要输出文档的大小，也就是 root.jffs2 的 size。很重要的是, 为了不浪费 flash 空间, 這个值
最好符合 flash driver 所规划的分区大小。
(5)如果挂载后会出现类似：CLEANMARKER node found at 0x0042c000 has totlen 0xc != normal 0x0 的警告，则加上 -n 就会消失。
在 pc 上 mount jffs2 镜像文件
首先确保系统支持 jffs2 文件系统，通过命令 cat /proc/filesystems 查看
sudo modprobe mtdram

sudo modprobe mtdblock　插入此模块以后将会生成节点：/dev/mtdblock0
sudo modprobe jffs2
在一些系统 fedora core 5 ,linux 2.6.15.1 上的 mtdram 只有 4.2MB
所以最好自己指定。
例如:
modprobe mtdram total_size=49152 erase_size=128
sudo dd if=rootfs-jffs2.img of=/dev/mtdblock0
sudo mount -t jffs2 /dev/mtdblock0 /mnt
更多参考文档：
mkfs.jffs2 参数详解
http://blog.sina.com.cn/s/blog_4a4163880100cogf.html~type=v5_one&label=rela_prevarticle
在 linux pc 上挂载 jffs2 文件系
http://blog.sina.com.cn/s/blog_4a4163880100cozw.html

2.4 yafss，yaffs2 文件系统
2.4.1 yaffs2 文件系统制作工具 mkyaffs2image
2.4.2 在 pc 上挂载 yaffs2 文件系统
2.4.3 通过工具释放 yaffs2 文件系统

YAFFS ， Yet Another Flash File System ，是一种类似于 JFFS/JFFS2 的专门为 Flash 设计的嵌入式文件系统。与 JFFS 相比，
它减少了一些功能，因此速度更快、占用内存更少。 YAFFS 和 JFFS 都提供了写均衡，垃圾收集等底层操作。它们的不同之处在于：
1 ）、 JFFS 是一种日志文件系统，通过日志机制保证文件系统的稳定性。 YAFFS 仅仅借鉴了日志系统的思想，不提供日志机能，所以稳定
性不如 JAFFS ，但是资源占用少。
2 ）、 JFFS 中使用多级链表管理需要回收的脏块，并且使用系统生成伪随机变量决定要回收的块，通过这种方法能提供较好的写均衡，在
YAFFS 中是从头到尾对块搜索，所以在垃圾收集上 JFFS 的速度慢，但是能延长 NAND 的寿命。
3 ）、 JFFS 支持文件压缩，适合存储容量较小的系统； YAFFS 不支持压缩，更适合存储容量大的系统。
YAFFS 还带有 NAND 芯片驱动，并为嵌入式系统提供了直接访问文件系统的 API ，用户可以不使用 Linux 中的 MTD 和 VFS ，直接对文件
进行操作。 NAND Flash 大多采用 MTD+YAFFS 的模式。 MTD （ Memory Technology Devices ，内存技术设备）是对 Flash 操作的
接口，提供了一系列的标准函数，将硬件驱动设计和系统程序设计分开。
YAFFS2 是 YAFFS 的升级版，能更好的支持 NAND FLASH 。

2.4.1 yaffs2 文件系统制作工具 mkyaffs2image
1. android yaffs2 源代码 external/yaffs2/
2. 从网上下载 yaffs2 源码
下载：http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs/
下载：http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs2/
//点击左下角的 Download tarball 下整个 tar 包
cvs 下载:
export CVSROOT=:pserver:anonymous@cvs.aleph1.co.uk:/home/aleph1/cvs cvs logon
cvs co yaffs2

tar -xvf yaffs2.tar.bz;cd yaffs2;make
为 ubuntu 8.10 添加 yaffs 文件系统支持
sudo mkdir -p /lib/modules/2.6.27-4-generic/kernel/fs/yaffs2
sudo cp yaffs2.ko /lib/modules/2.6.27-4-generic/kernel/fs/yaffs2/
sudo insmod /lib/modules/2.6.27-4-generic/kernel/fs/yaffs2/yaffs2.ko
制作 yaffs2 文件系统
mkyaffs2image /nfsroot/rootfs/system system.img
/nfsroot/rootfs/system 为文件系统所在的目录 system.img 为生成的镜像文件

2.4.2 在 pc 上挂载 yaffs2 文件系统
sudo mkdir -p /mnt/mtd/yaffs2
sudo modprobe mtdblock
sudo modprobe mtdram total_size=100000 erase_size=256
sudo insmod /lib/modules/2.6.27-4-generic/kernel/fs/yaffs2/yaffs2.ko
sudo dd if=rootfs.yaffs2 of=/dev/mtdblock0
sudo mount -t yaffs2 /dev/mtdblock0 /mnt/mtd/yaffs2
#modprobe mtdram total_size=49152 erase_size=128
#cat rootfs.yaffs2 >/dev/mtdblock0

2.4.3 通过工具释放 yaffs2 文件系统
yaffs2 image 逆向工具
http://blog.csdn.net/absurd/archive/2008/11/05/3223825.aspx
获取源代码：
http://www.limodev.cn/bbs/download/file.php?id=1

2.5 虚拟文件系统（sysfs，proc，tsmpfs 等）
2.5.1 虚拟文件系统概述
2.5.2 proc 文件系统
2.5.3 sysfs 文件系统
2.5.4 tmpfs 文件系统
2.5.5 usbdevfs 文件系统
2.5.6 devpts 文件系统

2.5.1 虚拟文件系统概述
虚拟内核文件系统（Virtual Kernel File Systems），是指那些是由内核产生但并不存在于硬盘上（存在于内存中）的文件系统，
他们被用来与内核进行通信前面介绍的 ext2，ext3，jffs2，yaffs2 等目录和文件，都是真真正正、实实在在的存储在具体的外部存
储设备上的，它们可能是在本机的硬盘、闪存、光盘中，可能保存在不只一个磁盘分区中，也可能保存在网络中其它主机的存储设备中的。
虚拟文件系统，虽然它们出现在根文件系统中，但它里面的内容却无法在任何外部存储设备中找到，因为它们都在内存中。
＝＝＝＝＝＝＝＝＝＝
android 网络挂载：
rootfs / rootfs rw 0 0
/dev/root / nfs rw,vers=2,rsize=1024,wsize=1024,...
tmpfs /dev tmpfs rw,mode=755 0 0
devpts /dev/pts devpts rw,mode=600 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0

tmpfs /sqlite_stmt_journals tmpfs rw,size=4096k 0 0
/dev/block/mmcblk0p1 /sdcard vfat rw,...

==========

android 本机挂载（使用 flash 中的文件系统）
rootfs / rootfs ro 0 0

tmpfs /dev tmpfs rw,mode=755 0 0
devpts /dev/pts devpts rw,mode=600 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
tmpfs /sqlite_stmt_journals tmpfs rw,size=4096k 0 0

/dev/block/mtdblock2 /system yaffs2 ro 0 0
/dev/block/mtdblock3 /data yaffs2 rw,nosuid,nodev 0 0
/dev/block/mmcblk0p1 /sdcard vfat rw
==========

ubuntu 系统：
/dev/sda8 on / type ext3 (rw,relatime,errors=remount-ro)
tmpfs on /lib/init/rw type tmpfs (rw,nosuid,mode=0755)
/proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
varrun on /var/run type tmpfs (rw,nosuid,mode=0755)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=620)
/dev/sda7 on /boot type ext3 (rw,relatime)
/dev/sda11 on /home type ext3 (rw,relatime)
/dev/sdb5 on /opt type ext3 (rw,relatime)
/dev/sda9 on /usr/local type ext3 (rw,relatime)
/dev/sda1 on /windows/c type vfat (rw,utf8,umask=007,gid=1000)
/dev/sda5 on /windows/d type vfat (rw,utf8,umask=007,gid=1000)
/dev/sda6 on /windows/e type vfat (rw,utf8,umask=007,gid=1000)
==========

2.5.2 proc 文件系统

proc 是一个重要虚拟文件系统，通过它里面的一些文件，可以获取系统状态信息并修改某些系统的配置信息。proc 文件系统本身不占用
磁盘空间，它仅存在于内存之中，为操作系统本身和应用程序之间的通信提供了一个安全的接口。当我们在内核中添加了新功能或设备驱
动时，经常需要得到一些系统状态的信息，一般这样的功能可能需要经过一些象 ioctl()这样的系统调用来完成。系统调用接口对于一些
功能性的信息可能是适合的，因为应用程序必须将这些信息读出后再做一定的处理。但对于一些实时性的系统信息，例如内存的使用状况，
或者是驱动设备的统计资料等，我们更需要一个比较简单易用的接口来取得它们。proc 文件系统就是这样的一个接口，我们可以简单的用
cat、strings 程序来查看这些信息。例如，执行下面的命令：

```
cat /proc/filesystems      //操作系统支持的文件系统类型
cat /proc/meminfo          //内存的实时信息，内存大小等
cat /proc/partitions       //存储器分区信息
```

cat /proc/cpuinfo　　//查看 cpu 信息
同样的，free、df、top、ps 等程序的功能实现，强烈依赖于 proc 文件系统，为了使用那些程序，一定要使内核支持 proc 文件系统，
并将其挂接到根文件系统的/proc 目录下。
其他使用 /proc 文件系统的例子：
processor　: 0
vendor_id　: AuthenticAMD
processor : 1
vendor_id　: AuthenticAMD
model name　: AMD Athlon(tm) 64 X2 Dual Core CPU 5000+
1．vmware 虚拟机无法正常启动
在 Linux 下，单个进程的最大内存使用量受/proc/sys/kernel/shmmax 中设置的数字限制(单位为字节)，
例如 ubuntu 8.10 的 shmmax 默认值为 33554432 字节(33554432bytes/1024/1024=32MB)。
2．scratchbox 开发工具不能登录
/scratchbox/login
Inconsistency detected by ld.so: rtld.c: 1192: dl_main: Assertion `(void *) ph->p_vaddr == _rtld_local._dl_sysinfo_dso' failed!
NOTE: on Ubuntu installation, you have to disable VDSO to make Scratchbox work fine,
or you'll get errors like this:
在 ubuntu 系统中，我们必须关闭 VDSO 标记，以便 scratchbox 能正常工作
echo 0 | sudo tee /proc/sys/vm/vdso_enabled
echo 4096 | sudo tee /proc/sys/vm/mmap_min_addr
vm.vdso_enabled = 0
vm.mmap_min_addr = 4096
修改 /proc 文件系统值的方法
1．直接修改
echo "2147483648" | sudo tee /proc/sys/kernel/shmmax
echo 0 | sudo tee /proc/sys/vm/vdso_enabled
echo 4096 | sudo tee /proc/sys/vm/mmap_min_addr
2.将以下命令放入 /etc/rc.local 启动文件中：
echo "2147483648" > /proc/sys/kernel/shmmax
echo 0　　> /proc/sys/vm/vdso_enabled
echo 4096　　> /proc/sys/vm/mmap_min_addr
3.使用 sysctl 命令来更改 SHMMAX 的值：
sysctl -w kernel.shmmax=2147483648
4.内核参数插入到 /etc/sysctl.conf 启动文件中，使这种更改永久有效
echo "kernel.shmmax=2147483648" >> /etc/sysctl.conf
sudo sysctl –p
./system/core/logcat/logcat.cpp:403:　　fd = open("/proc/cmdline", O_RDONLY);
./system/core/init/init.c:553:　　char cmdline[1024];
./system/core/init/init.c:557:　　fd = open("/proc/cmdline", O_RDONLY);
./system/core/init/init.c:580:　　chmod("/proc/cmdline", 0440);
./system/core/init/bootchart.c:139:　　proc_read("/proc/cmdline", cmdline, sizeof(cmdline));
./system/core/init/bootchart.c:319:　　proc_read("/proc/cmdline", cmdline, sizeof(cmdline));
./system/core/init/bootchart.c:320:　　s = strstr(cmdline, KERNEL_OPTION);
./system/core/rootdir/init.rc:162:　　chown root radio /proc/cmdline

2.5.3 sysfs 文件系统
与 proc 文件系统类似，sysfs 文件系统也是一个不占有任何磁盘空间的虚拟文件系统。它通常被挂接在/sys 目录下。sysfs 文件系统是 Linux2.6 内核引入的，它把连接在系

统上的设备和总线组织成为一个分级的文件，使得它们可以在用户空间存取。其实
sysfs 是从 proc 和 devfs 中划分出来的。

一、devfs

linux 下有专门的文件系统用来对设备进行管理，devfs 和 sysfs 就是其中两种。
在 2.6 内核以前一直使用的是 devfs，devfs 挂载于/dev 目录下，提供了一种类似于文件的方法来管理位于/dev 目录下的所有设备，我们知道
/dev 目录下的每一个文件都对应的是一个设备，至于当前该设备存在与否先且不论，而且这些特殊文件是位于根文件系统上的，在制作文件
系统的时候我们就已经建立了这些设备文件，因此通过操作这些特殊文件，可以实现与内核进行交互。但是 devfs 文件系统有一些缺点，例如：
不确定的设备映射，有时一个设备映射的设备文件可能不同，例如我的 U 盘可能对应 sda 有可能对应 sdb；没有足够的主/辅设备号，当设备过多
的时候，显然这会成为一个问题；/dev 目录下文件太多而且不能表示当前系统上的实际设备；命名不够灵活，不能任意指定等等。

二、sysfs

正因为上述这些问题的存在，在 linux2.6 内核以后，引入了一个新的文件系统 sysfs，它挂载于/sys 目录下，跟 devfs 一样它也是一个
虚拟文件系统，也是用来对系统的设备进行管理的，它把实际连接到系统上的设备和总线组织成一个分级的文件，用户空间的程序同样可以利用
这些信息以实现和内核的交互，该文件系统是当前系统上实际设备树的一个直观反应，它是通过 kobject 子系统来建立这个信息的，当一个
kobject 被创建的时候，对应的文件和目录也就被创建了，位于/sys 下的相关目录下，既然每个设备在 sysfs 中都有唯一对应的目录，那么也
就可以被用户空间读写了。用户空间的工具 udev 就是利用了 sysfs 提供的信息来实现所有 devfs 的功能的，但不同的是 udev 运行在用户空间中，
而 devfs 却运行在内核空间，而且 udev 不存在 devfs 那些先天的缺陷。很显然，sysfs 将是未来发展的方向。

2.5.4 tmpfs 文件系统

tmpfs 是 Linux 特有的文件系统，唯一的标准挂接点是/dev/shm。当然，用户可以将其挂接在其他地方。tmpfs 有些像虚拟磁盘（ramdisk），
但不是一回事。说其像虚拟磁盘，是因为它可以使用你的 RAM，但它也可以使用你的交换分区。传统的虚拟磁盘是一个块设备，而且需要一个 mkfs
之类的命令格式化它才能使用。tmpfs 是一个独立的文件系统，不是块设备，只要挂接，立即就可以使用。tmpfs 的大下是不确定的，它最初只有
很小的空间，但随着文件的复制和创建，它的大小就会不断变化，换句话说，它会根据你的实际需要而改变大小；tmpfs 的速度非常惊人，毕竟它
是驻留在 RAM 中的，即使用了交换分区，性能仍然非常卓越；由于 tmpfs 是驻留在 RAM 的，因此它的内容是不持久的，断电后，tmpfs 的内容就消
失了，这也是被称作 tmpfs 的根本原因。
tmpfs 是 ramfs 的衍生物，用来限制缓存大小、向 swap 空间写入数据。它是用来保存 VM 所有文件的文件系统。
tmpfs 中缓存的内容全部是临时的。一旦卸载，所有的内容都会遗失。它把所有的缓存置于内核，它的规模随着
文件的规模同步变化。但是它规模有大小限制，可以修改。它可以把当前不再需要的页写入到 swap 空间。
tmpfs 和 ramfs 本身就是一个文件系统, 用的时候只需要直接挂载就可以. tmpfs 可以使用 ram,

也可以使用 swap
共享内存的时候会使用 tmpfs
系统默认共享内存是内存的一半大小！ /dev/shm 是挂载点！
通过 df -h 可以看出，默认状况下它为内存大小的一半：
文件系统 容量 已用 可用 已用% 挂载点
tmpfs 1013M 12K 1013M 1% /dev/shm
mount | grep tmpfs 显示当前系统中的 tmpfs：
tmpfs on /lib/init/rw type tmpfs (rw,nosuid,mode=0755)
varrun on /var/run type tmpfs (rw,nosuid,mode=0755)
varlock on /var/lock type tmpfs (rw,noexec,nosuid,nodev,mode=1777)
udev on /dev type tmpfs (rw,mode=0755)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
lrm on /lib/modules/2.6.27-4-generic/volatile type tmpfs (rw,mode=755)

2.5.6 usbdevfs 文件系统
顾名思义，usbdevfs 就是 USB 设备文件系统，它是一个动态生成的文件系统，有些类似于 proc 文件系统。它的标准挂接点是/proc/bus/usb，当然，也可以挂接到其他地方。它主要用于：用户级驱动、即插即用、提供 USB 设备信息、应用程序轮询 USB 设备的变化等。

2.5.7 devpts 文件系统
devpts 文件系统为伪终端提供了一个标准接口，它的标准挂接点是/dev/pts。只要 pty 的主复合设备/dev/ptmx 被打开，就会在/dev/pts 下动态的创建一个新的 pty 设备文件。挂接时，UID、GID 及其工作模式会指定给 devpts 文件系统的所有 pty 文件。这可以保证伪终端的安全性。
讨论 devpts 文件系统的详细内容，已经超过本文范围，还请读者参考其他专著。

2.6 一些必要重要的系统文件 （ /etc/fstab ， inittab，init.rc 等）
2.6.1 /etc/inittab
2.6.2 /etc/init.d/rcS
2.6.3 /etc/fstab 文件

==========
2.6.1 /etc/inittab
initab 被 init 使用
2.6.1.1 老平台 inittab 文件内容
2.6.1.1 gpephone 官方的 inittab 文件（与 redhat，federo 差不多）
2.6.1.1 ubuntu 中没有 inittab 文件
==========
2.6.1.1 老平台 inittab 文件内容

::sysinit:/etc/init.d/rcS
::respawn:-/bin/sh
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::shutdown:/sbin/swapoff -a

2.6.1.2 gpephone 官方的 inittab 文件（与 redhat，federo 差不多

```
# /etc/inittab: init(8) configuration.
# $Id: inittab,v 1.91 2002/01/25 13:35:21 miquels Exp $

# The default runlevel.
id:5:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin

# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.

l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6
# Normally not reached, but fallthrough in case of emergency.
z6:6:respawn:/sbin/sulogin
1:2345:respawn:/sbin/getty 38400 tty1
```

2.6.1.3 ubuntu 中没有 inittab 文件

在 unbutu 系统中我们没看到此文件，是因为 ubuntu 用的是 upstart ，lfs 中使用的是 sysvinit ，嵌入式系统中

一般使用的是 busybox 中的 init ,android 系统使用的是 system/core/init

init:

```
main()
   init_main()
      read_inittab();
```

gdm 运行后

```
/etc/rc5.d/S30gdm -> ../init.d/gdm
/etc/init.d/gdm:19:DAEMON=/usr/sbin/gdm
/etc/init.d/gdm:24:      SSD_ARG="--startas $DAEMON"
```

/etc/init.d/gdm:27: SSD_ARG="--exec $DAEMON"
启动 gdm：
log_begin_msg "Starting GNOME Display Manager..."
start-stop-daemon --start --quiet --oknodo --pidfile $PIDFILE --name gdm $SSD_ARG --
$CONFIG_FILE >/dev/null

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
2.6.2 /etc/init.d/rcS
－－－－－－－－－－－－－－－－－－－
#!/bin/sh
挂在 /etc/fstab 中的文件系统
/bin/mount -a
. /etc/default/rcS
#环境变量
. /etc/profile
#屏幕叫准备
. /etc/X11/run-calibrate
#启动 X
. /etc/X11/Xserver
. /etc/scripts/testd-bus.sh
#启动 dbus 消息总线
#启动 gpephone
－－－－－－－－－－－－－－－－－－－

ubuntu 系统
－－－－－－－－－－－－－－－－
exec /etc/init.d/rc S
－－－－－－－－－－－－－－－－
会依此执行 /etc/rcS.d/ 下以
S01mountkernfs.sh
S02hostname.sh
S10udev
S11mountdevsubfs.sh
S20checkroot.sh
S22mtab.sh
S30checkfs.sh
S35mountall.sh
S40networking
S43portmap
S55bootmisc.sh

./rc3.d/S30gdm
./rc2.d/S30gdm
./rc4.d/S30gdm
./rc5.d/S30gdm

/etc/rcS.d/S35mountall.sh -> ../init.d/mountall.sh
mount -a -t nonfs,nfs4,smbfs,cifs,ncp,ncpfs,coda,ocfs2,gfs,gfs2 -O no_netdev
mount 命令的一些解析：
mount -a [-t|-O] ...　　: mount all stuff from /etc/fstab
mount -t type dev dir　　: ordinary mount command

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

2.6.3 /etc/fstab 文件
Util-linux 软件包包含许多工具。其中比较重要的是加载、卸载、格式化、分区和管理硬盘驱动器，打开 tty 端口和得到内核消息
arch　　报告机器的体系结构
blockdev　　在命令行中调用块设备的 ioctl
cal　　显示一个简单的日历。
cfdisk　　处理指定设备的分区表
column　　把输出格式化为几列
ctrlaltdel　　设置 CTRL+ALT+DEL 组合键的功能为硬重启或软重启
dmesg　　显示内核的启动信息
fdisk　磁盘分区管理程序
fsck.cramfs　　对 Cramfs 文件系统的一致性进行检查
getopt　　在给出的命令行进行选项和参数解析
hexdump　　用用户指定的方式(包括 ASCII, 十进制, 十六进制, 八进制)显示一个文件或者标准输入的数据
hwclock　　查询和设置硬件时钟(也被称为 RTC 或 BIOS 时钟)。
ipcrm　　删除给定的进程间通信(IPC)资源
mkfs　　在一个设备(通常是一个硬盘分区)设备上建立文件系统
mkfs.cramfs　　创建 cramfs 文件系统
mkswap　初始化指定设备或文件，以用做交换分区
more　分屏显示文件，但没有 less 好用
mount　　把一个文件系统从一个设备挂载到一个目录
ramsize　显示或者改变 RAM disk 的大小
raw 将一个原始的 Linux 字符设备绑定到一个块设备
rdev　查询和设置内核的根设备和其他信息
readprofile　　显示内核侧写文件/proc/profile 的信息
rename　对文件进行重命名
renice　修改正在运行进程的优先级
sfdisk　磁盘分区表管理工具
umount　　卸载一个被挂载的文件系统
mount 挂载与/etc/fstab
mount 源目录 目的目录
mount -a 自动挂载/etc/fstab 中的文件系统
根目录 / 是必须挂载的，而且一定要先于其它 mount point 被挂载进来。　　其它 mount point 必须为已建立的目录，可任意指定，
但一定要遵守必须的系统目录架构原则　　所有 mount point 在同一时间之内，只能挂载一次。　　所有 partition 在同一时间之内，
只能挂载一次。　　如若进行卸载，您必须先将工作目录移到 mount point(及其子目录) 之外。
/etc/fstab

第一列：label
第二列：挂载点
第三列：分区的文件系统
第四列：文件系统挂载选项,看附件啦
第五列：是否被 dump 作用。0 代表不要做 dump 备份，1 代表要每天进行 dump 的动作。 2 也代表其它不定日期的 dump 备份动作，通常这个数值不是 0 就是 1 啦！
第六列：是否以 fsck 检查分区（开机时候检查分区）0 为不检查，1 为开机的时候检查，2 为在稍后的时间检查

/dev/sda8 on / type ext3 (rw,relatime,errors=remount-ro)
/proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=620)
/dev/sda7 on /boot type ext3 (rw,relatime)
/dev/sda11 on /home type ext3 (rw,relatime)
/dev/sdb5 on /opt type ext3 (rw,relatime)
/dev/sda9 on /usr/local type ext3 (rw,relatime)
/dev/sda1 on /windows/c type vfat (rw,utf8,umask=007,gid=1000)
/dev/sda5 on /windows/d type vfat (rw,utf8,umask=007,gid=1000)
/dev/sda6 on /windows/e type vfat (rw,utf8,umask=007,gid=1000)

可以在/etc/fstab 中进行指定
proc /proc proc defaults 0 0
none /tmp ramfs defaults 0 0
sysfs /sys sysfs defaults 0 0
none /dev/pts devpts defaults 0 0
./util-linux-2.12r/mount/mount.c
main()
result = do_mount_all (types, options, test_opts);

mount --help 可以知道 mount -a 是 mount 所有/etc/fstab
mount -a [-t|-O] ... : mount all stuff from /etc/fstab

==========

2.7 制作文件系统
2.7.1 原始方式
2.7.2 通过 scratchbox 等工具
2.7.3 通过 android 源码集成开发环境

2.7.1 原始方式
创建基本文件系统标准目录（根据不同的 linux 系统，ubuntu 跟 android 目录结构就完全不同）
lfs 中的标准目录：

创建修改必要的配置文件
/scratchbox/source2/source/busybox/busybox-1.1.2/examples/bootfloppy/etc/
vim ${CLFS_ROOTFS_DIR}/etc/profile
vim ${CLFS_ROOTFS_DIR}/etc/inittab
vim ${CLFS_ROOTFS_DIR}/etc/fstab

vim　${CLFS_ROOTFS_DIR}/etc/init.d/rcS

创建帐号以及密码文件
sudo vim ${CLFS_ROOTFS_DIR}/passwd
拷贝必须的动态库文件
cd　${CLFS_ROOTFS_DIR}/lib
cp -d　$COMPILER_LIB/ld* ./
cp　　$COMPILER_LIB/libc-2.3.5.so ./
cp -d　$COMPILER_LIB/libc.so.6 ./
cp　　$COMPILER_LIB/libm-* ./
cp -d　$COMPILER_LIB/libm.s* ./
cp　　$COMPILER_LIB/libcrypt-* ./
cp -d　$COMPILER_LIB/libcrypt.s* ./
拷贝可选的动态库文件
如果需要域名解析：
1）增加/etc/resolv.conf
[root@lqm /etc]#cat resolv.conf
nameserver 192.168.x.x　//加入域名解析器
2）增加相应动态库的支持
增加如下：
libnss_files
libnss_dns
libresolv.so
find find . -name "libnss*"　$COMPILER_LIB/
./libnss_files.so.2
./libnss_files.so
./libnss_dns-2.3.2.so
./libnss_dns.so
./libnss_files-2.3.2.so
./libnss_dns.so.2
find . -name "libresolv*"　/scratchbox/compilers/arm-linux-gcc-3.4.4-glibc-2.3.5/arm-unknown-linux-gnu/lib/
./libresolv.so
./libresolv.so.2
./libresolv-2.3.2.so

2.7.2 通过 scratchbox 等工具

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
2.7.3 通过 android 源码集成开发环境
环境搭建问题：
　　1.为什么拷贝　cupcake 编译结果　out/target/product/littleton/root/　到内核顶层目录?
　2.cupcake-jianping/make_image15.sh 中的 choosecombo 是什么作用？
　　3.make_image15.sh 与 make_env15.sh 只差一句 make -j2？
　4.补充 shell 脚本知识。

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
2.7.4 配置 android 网络文件系统

下面是曾经用过的几种开发板的命令行参数：
S3C2410 启动参数：
noinitrd root=/dev/nfs nfsroot=192.168.2.56:/nfsroot/rootfs
ip=192.168.2.188:192.168.2.56:192.168.2.56:255.255.255.0::eth0:on console=ttySAC0
S3C2440 启动参数：
setenv bootargs console=ttySAC0 root=/dev/nfs nfsroot=192.168.2.56:/nfsroot/rootfs
ip=192.168.2.175:192.168.2.56:192.168.2.201:255.255.255.0::eth0:on mem=64M init=/init
marvell 310 启动参数：
boot root=/dev/nfs nfsroot=192.168.2.56:/nfsroot/rootfs,rsize=1024,wsize=1024
ip=192.168.2.176:192.168.2.201:192.168.2.201:255.255.255.0::eth0:-On
console=ttyS2,115200 mem=64M init=/init
当前 android 内核的.config 文件中的命令行参数：
CONFIG_CMDLINE="root=/dev/nfs nfsroot=192.168.1.100:/nfsroot/rootfs,rsize=1024,wsize=1024
ip=192.168.1.101:192.168.1.100:192.168.1.100:255.255.255.0::usb0:on
console=ttyS1,115200 mem=128M init=/init android uart_dma=1"

`root=' 参数
此参数告诉内核启动时以那个设备作为根文件系统使用。我的 pc 根文件系统：
/dev/sda8 9614116 6522156 2603588 72% /
ubuntu 的/boot/grub/menu.lst 参数：
kernel /vmlinuz-2.6.27-4-generic root=UUID=2ffa7dc6-2dc5-4b66-8661-1226c086951a
ro locale=zh_CN quiet splash
initrd /initrd.img-2.6.27-4-generic
其中 root 可以设置为：root=/dev/sda8
/dev/nfs, 这并非真的是个设备， 而是一个告诉核心经由网络取得根文件系统
lfs 的/boot/grub/menu.lst 参数：
title LFS 6.4
root (hd1,1)
kernel /boot/lfskernel-2.6.27.4 root=/dev/sdb1

`nfsroot=' 参数
这个参数告诉内核到哪台 pc 的哪个目录读取根文件系统。此参数的格式如下：
nfsroot=[<server-ip>:]<root-dir>[,<nfs-options>]
<server-ip> --pc 机的 ip 地址，如果此字段没给值，那么将使用由 nfsaddrs 变量（见下面）所决定的值。
<root-dir> -- pc 服务端上要作为根挂入的目录域名(/nfsroot/rootfs)
<nfs-options> -- 标准的网络文件系统选项。所有选项都以逗号分开。如果没有给定此选项字段则使用下列的缺省值：

port = as given by server portmap daemon
rsize = 1024
wsize = 1024
timeo = 7
retrans = 3
acregmin = 3
acregmax = 60
acdirmin = 30
acdirmax = 60
flags = hard, nointr, noposix, cto, ac

`init=' 参数
内核启动时缺省执行 `init' 程序，内核将会到/sbin/, /bin/ 等目录下查找默认的 init，如果没有

找到那么就报告出错。
而最后它会去试 /bin/sh （可能在 /etc/rc ）。如果说，例如，如果你的 init 程序坏掉了，只要使用 init=/bin/sh
这个启动参数就能让你在启动时直接跳到解译环境(shell)，使你能够换掉坏掉的程序。

`ip=' 参数
nfsaddrs=<my-ip>:<serv-ip>:<gw-ip>:<netmask>:<name>:<dev>:<auto>
ip=192.168.1.101:192.168.1.100:192.168.1.100:255.255.255.0::usb0:on
ip=192.168.2.175:192.168.2.56:192.168.2.201:255.255.255.0::eth0:on
<my-ip> -- 板子的 ip 使用何种协议端视配置核心时打开的选项以及 <auto> 参数而定。如果设定此参数，就不会使用反向地址解析协议或启动协议。
<serv-ip> -- 网络文件系统服务端之互联网地址。
<gw-ip> -- 网关(gateway)，
<netmask> -- 本地网络界面的网络掩码。如果为空白，则网络掩码由客户端的互联网地址导出，除非由启动协议接收到值。
<name> -- 客户端的域名。如果空白，则使用客户端互联网地址之 ASCII-标记法，或由启动协议接收的值。
<dev> -- 要使用的网络设备域名。
<auto> -- 用以作为自动配置的方法。

参考文档：
ramfs, rootfs, initrd and initramfs
http://blog.chinaunix.net/u2/89923/showart_1890405.html
嵌入式系统文件系统比较
http://blog.sina.com.cn/s/blog_53ad41a50100eptc.html
LINUX 系统性能调谐
http://www.host01.com/article/server/00070002/0621409052193755_2.htm
怎样限制或者修改/dev/shm 的大小
http://www.linuxfly.cn/html/65/t-665.html

==========

==========

3. 制作交叉工具链
3.1 什么是工具链
3.2 获取交叉工具链的几种途径
3.3 android 工具链与 gnu 工具链的比较

每一个软件，在编译的过程中，都要经过一系列的处理，才能从源代码变成可执行的目标代码。这一系列处理包括：预编译，高级语言编译，
汇编，连接及重定位。这一套流程里面用到的每个工具和相关的库组成的集合，就称为工具链（tool chain）。以 GNU 的开发工具 GCC 为例，
它就包括了预编译器 cpp，c 编译器 gcc，汇编器 as，和连接器 ld 等。在 GNU 自己对工具链定义中，还加进了一套额外的用于处理二进制包的
工具包 binutils，整个工具链应该是 GCC+binutils+Glibc， binutils 其实与 Glibc 关系不是很大，它可以被独立安装的，所以 GNU 工具

链也可以狭义地被理解为 GCC+Glibc。
要构建出一个交叉工具链，需要解决三个问题。一是这个工具链必须是可以运行在原工作站平台上的。二是我们需要更换一个与目标平台对应的
汇编器，使得工具链能产生对应的目标代码，三是要更换一套与目标平台对应的二进制库，使得工具链在连接时能找到正确的二进制库。
3.2 获取交叉工具链的几种途径
3.2.1 利用源代码制作交叉工具链
网上直接下载工具链或者从方案商处获取(如：marvell)
下载地址：
http://www.angstrom-distribution.org/unstable/
3.2.2 用脚本制作工具链
3.2.2.1 croostool-0.43
http://www.kegel.com/crosstool/crosstool-0.43.tar.gz
制作工具链的源码包搭配情况: http://www.kegel.com/crosstool/crosstool-0.43/buildlogs/
3.2.2.2 buildroot
http://buildroot.uclibc.org/downloads/snapshots/buildroot-snapshot.tar.bz2
若想详细地了解 buildroot 可参考该文档 http://buildroot.uclibc.org/buildroot.html
3.2.3 利用 OE 制作工具链
http://www.scratchbox.org/wiki/OpenEmbedded

3.3 android 工具链与 gnu 工具链的比较
Android 所用的 Toolchain（即交叉编译工具链）可从下面的网址下载：
http://android.kernel.org/pub/android-toolchain-20081019.tar.bz2。如果下载了完整的 Android 项目的源代码，则可以在
“<your_android>/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin”目录下找到交叉编译工具，比如 Android 所用的
arm-eabi-gcc-4.2.1。Android 并没有采用 glibc 作为 C 库，而是采用了 Google 自己开发的 Bionic Libc，它的官方 Toolchain 也是基于
Bionic Libc 而并非 glibc 的。这使得使用或移植其他 Toolchain 来用于 Android 要比较麻烦：在 Google 公布用于 Android 的官方 Toolchain 之前，
多数的 Android 爱好者使用的 Toolchain 是在
http://www.codesourcery.com/gnu_toolchains/arm/download.html 下载的一个通用的
Toolchain，它用来编译和移植 Android 的 Linux 内核是可行的，因为内核并不需要 C 库，但是开发 Android 的应用程序时，直接采用或者移植其他
的 Toolchain 都比较麻烦，其他 Toolchain 编译的应用程序只能采用静态编译的方式才能运行于 Android 模拟器中，这显然是实际开发中所不能接
受的方式。目前尚没有看到说明成功移植其他交叉编译器来编译 Android 应用程序的资料。
与 glibc 相比，Bionic Libc 有如下一些特点：
- 采用 BSD License，而不是 glibc 的 GPL License；
- 大小只有大约 200k，比 glibc 差不多小一半，且比 glibc 更快；
- 实现了一个更小、更快的 pthread；
- 提供了一些 Android 所需要的重要函数，如”getprop”, “LOGI”等；
- 不完全支持 POSIX 标准，比如 C++ exceptions，wide chars 等；
- 不提供 libthread_db 和 libm 的实现

另外，Android 中所用的其他一些二进制工具也比较特殊：
- 加载动态库时使用的是/system/bin/linker 而不是常用的/lib/ld.so;
- prelink 工具不是常用的 prelink 而是 apriori，其源代码位于”<your_android>/build/tools/apriori”

- strip 工具也没有采用常用的 strip，即“<your_android>/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin”
目录下的 arm-eabi-strip，而是位于<your_android>/out/host/linux-x86/bin/的 soslim 工具。

参考文档：
CLFS2.0 原理分析
http://www.linuxsir.org/bbs/showthread.php?t=267672
Cross-Compiled Linux From Scratch
http://cross-lfs.org/view/clfs-sysroot/arm/
全手工制作 arm-linux 交叉编译工具链《一》
http://blog.chinaunix.net/u2/62168/showart_1898748.html
自己制作 arm-linux 交叉编译环境(一)－scratch 篇
http://blog.csdn.net/chenzhixin/archive/2007/01/12/1481442.aspx
如何建立交叉编译工具链
http://www.decell.org/article.asp?id=53
Android Toolchain 与 Bionic Libc
http://www.top-e.org/jiaoshi/html/?151.html
ndroid 编译环境(2) - 手工编译 C 模块

==========

==========
4. 软件编译常识
4.1 链接器和加载器
4.2 android 的标准链接器和加载器
4.3 Makefile 基本语法
何为链接器和加载器？
链接器为 ld,加载为 ld-linux.so.2,两个的区别很大，一个编译时用,一个运行时用,ld 负责在编译的搜索路径里找到要求的库,并查看
是否有提供了需要的 符号(如函数等),如果有,记录相关信息到程序中,由 ld-linux.so.2 在执行时查找到该库,并根据相关信息进行需
要符号的重定位等工作.注意 这两者的搜索库的方式是不同的。
动态连接器通常是指的动态加载器(不要与 Binutils 里的标准连接器 ld 混淆了)。动态连接器由 Glibc 提供，用来
找到并加载一个程序运行时所需的共享库，在做好程序运行的准备之后，运行这个程序。动态连接器的名称通常是
ld-linux.so.2，标准连接器 ld 由 Binutils 这个包提供。

标准连接器
查看 gcc 使用的标准连接器
mhf@mhf-desktop:/usr/local/marvell-arm-linux-4.1.1/bin$ arm-linux-gcc -print-prog-name=ld
编译时库的搜索路径,以下几种方式让连接器去找需要的库
1. 编译的时候明确指定，如： gcc test.c ./say.so -o test 中的 ./say.so
2. 编译 Binutils 的时候通过 LIB_PATH 变量指定，
如：make -C ld LIB_PATH=/tools/lib
-C ld LIB_PATH=/tools/lib

这个选项重新编译 ld 子目录中的所有文件。在命令行中指定 Makefile 的 LIB_PATH 变量值，使它明确指向/tools/lib 工具目录，
以覆盖默认值。这个变量的值指定了连接器的默认库搜索路径。
来源：Linux From Scratch - 版本 6.4 第 5 章 构建临时系统 5.4. Binutils-2.18 - 第一遍
http://www.bitctp.org/lfsbook-6.4/chapter05/binutils-pass1.html
3. 在源码包 configure 的时候通过 --with-lib-path 指定，或者 --lib- path
例如：
binutils-2.18/configure --prefix=/tools --disable-nls --with-lib-path=/tools/lib
配置选项的含义：
--with-lib-path=/tools/lib
告诉配置脚本在为编译 Binutils 的过程中使用正确的库搜索路径，也就是将 /tools/lib 传递给连接器。这防止连接器搜索宿主系统中的库文件目录。
来源： Linux From Scratch - 版本 6.4 第 5 章 构建临时系统 lfs 5.13. Binutils-2.18 - 第二遍
http://www.bitctp.org/lfsbook-6.4/chapter05/binutils-pass2.html
4. 到 ld –verbose | grep SEARCH 列出的默认目录下去找
5. -L/usr/gpephone/lib 指定的目录找
经常以 LDFLAGS=" -L/usr/gpephone/lib -L/lib -L/usr/lib -L/usr/X11R7/lib" 的方式传入

参数 -rpath 与 -rpath-link
如果使用了'-rpath'选项, 那运行时搜索路径就只从'-rpath'选项中得到
'nodefaultlib'标志一个对象,使在搜索本对象所依赖的库时,忽略所有缺省库搜索路径.
LDFLAGS="-Wl,-rpath-link=/usr/gpephone/lib/:/usr/gphone/lib:/usr/local/lib -L/usr/gpephone/lib -L/usr/gphone/lib"
-rpath 与 -rpath-link 的特性：
1. 在编译的时候我们都可以使用这两个路径，
2. '-rpath'跟'-rpath_link'的不同之处在于,由'-rpath'指定的路径会被包含到可执行程序中，并在运行时使用，
而'-rpath-link'选项仅仅在链接时起作用。

-dumpspecs Display all of the built in spec strings
-dumpversion Display the version of the compiler
-dumpmachine Display the compiler's target processor
-print-search-dirs Display the directories in the compiler's search path
-print-prog-name=<prog> Display the full path to compiler component <prog>
-specs=<file> Override built-in specs with the contents of <file>
-Wa,<options> Pass comma-separated <options> on to the assembler
-Wp,<options> Pass comma-separated <options> on to the preprocessor
-Wl,<options> Pass comma-separated <options> on to the linker
从工具链内建的规范中查看动态加载器
gcc -dumpspecs | grep dynamic-linker //本机
查看编译起所指定的动态加载器
1. s3c2440 (arm9tdmi) 平台的工具链
/scratchbox/compilers/arm-9tdmi-softfloat-linux-gcc-3.4.4-glibc-2.3.5/bin/arm-softfloat-linux-gnu-gcc -dumpspecs | grep dynamic-linker
/scratchbox/compilers/arm-softfloat-linux-gcc-3.4.4-glibc-2.3.5/bin/arm-softfloat-linux-gnu-gcc -dumpspecs | grep dynamic-linker
2. marvell 的工具链
/scratchbox/compilers/marvell-arm-linux-4.1.1/bin/arm-linux-gcc -dumpspecs | grep dynamic-linker
3. scrathbox 中工具链 host-gcc

/scratchbox/compilers/host-gcc/bin/host-gcc -dumpspecs | grep dynamic-linker

如果我们在编译的时候给编译起 gcc 指定 -specs＝/scratchbox/compilers/host-gcc/host-gcc.spec ， 那么-specs 指定
的规范将会覆盖工具链内建的规范。
cat /scratchbox/compilers/host-gcc/host-gcc.specs | grep ld 有如下内容：
-dynamic-linker /scratchbox/host_shared/lib/ld.so
/scratchbox/compilers/host-gcc/bin/gcc -specs=/scratchbox/compilers/host-gcc/host-gcc.specs
mhf@mhf-desktop:/usr/local/marvell-arm-linux-4.1.1/arm-iwmmxt-linux-gnueabi/bin$ ./gcc -dumpspecs|grep dynamic-linker
gcc -dumpspecs | sed 's@/lib/ld-linux.so.2@/tools&@g' | sudo tee `dirname $(gcc -print-libgcc-file-name)`/specs
cat `dirname $(gcc -print-libgcc-file-name)`/specs | grep tools

查看本机应用程序使用的动态加载器
readelf -l /usr/bin/make | grep interpreter
[Requesting program interpreter: /lib/ld-linux.so.2]
查看 scratchbox 中应用程序使用的动态加载器
readelf -l /scratchbox/tools/bin/make | grep interpreter
[Requesting program interpreter: /scratchbox/host_shared/lib/ld.so]

cd ~/svn/mohuifu.svn/trunk/mysource/compiler_test
/scratchbox/compilers/host-gcc/bin/gcc -specs=/scratchbox/compilers/host-gcc/host-gcc.specs -o ld.so.test1 ld.so.test.c
/scratchbox/compilers/host-gcc/bin/gcc -o ld.so.test2 ld.so.test.c
readelf -l ./ld.so.test1 | grep interpreter
readelf -l ./ld.so.test2 | grep interpreter
其他示例：
readelf -l /scratchbox/tools/bin/make | grep interpreter
readelf -l /usr/bin/make | grep interpreter
分别显示：
[Requesting program interpreter: /scratchbox/host_shared/lib/ld.so]
[Requesting program interpreter: /lib/ld-linux.so.2]
下面的方式也可以查看应用程序所使用的加载器
strings /scratchbox/tools/bin/make |grep lib
strings /usr/bin/make |grep lib
分别为：
/scratchbox/host_shared/lib/ld.so
/lib/ld-linux.so.2

查看应程序加载器库的搜索路径
显示 scratchbox 中加载器的库搜索路径
strings /scratchbox/host_shared/lib/ld.so |grep lib
display library search paths
/scratchbox/host_shared/lib/
/scratchbox/tools/lib/
显示本机中加载器的库搜索路径
strings /lib/ld-linux.so.2 |grep lib
display library search paths
/lib/
/usr/lib/
/lib/i486-linux-gnu/

/usr/lib/i486-linux-gnu/
ldd 验证应用程序所使用动态库
ldd /scratchbox/tools/bin/make
libc.so.6 => /scratchbox/host_shared/lib/libc.so.6 (0xb7ef9000)
/scratchbox/host_shared/lib/ld.so => /scratchbox/host_shared/lib/ld.so (0xb802f000)
ldd /usr/bin/make
librt.so.1 => /lib/tls/i686/cmov/librt.so.1 (0xb7fb9000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e5b000)
libpthread.so.0 => /lib/tls/i686/cmov/libpthread.so.0 (0xb7e42000)
/lib/ld-linux.so.2 (0xb7fd5000)

参考文档：
交叉编译中 libtool 相关的问题
http://hi.baidu.com/lieyu063/blog/item/9c99a2dd23e41f365882dd39.html
静态库和共享库库的定位搜索路径
http://blog.csdn.net/lwhsyit/archive/2008/08/26/2830783.aspx
Linux 动态连接原理
http://blog.chinaunix.net/u2/67984/showart_1359874.html
程序编译链接运行时对库关系的探讨（原创）
http://www.360doc.com/content/061107/09/13188_251964.html
http://lamp.linux.gov.cn/Linux/LFS-6.2/chapter05/toolchaintechnotes.html
[Linux 命令] ld 中文使用手册完全版(译)
http://blog.csdn.net/rstevens/archive/2008/01/28/2070568.aspx
scratchbox 是 mameo (nokia) 提供的一个集成开发环境，可以去官方网站：
http://www.scratchbox.org/
http://www.scratchbox.org/download/

4.2 android 的标准链接器和加载器
android 的标准链接器 ./prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/arm-eabi-ld
android 中标准连接器搜索库的路径
./prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/arm-eabi-ld -verbose | grep SEARCH
SEARCH_DIR("/android/mathias/armdev/toolchain-eabi-4.2.1/arm-eabi/lib");
Android 编译环境所用的交叉编译工具链是./prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/arm-eabi-gcc，
-I 和-L 参数指定了所用的 C 库头文件和动态库文件路径分别是 bionic/libc /include 和 out/target/product/generic/obj/lib，
其他还包括很多编译选项以及-D 所定义的预编译宏。这里值得留意的是参数“-Wl,-dynamic-linker,/system/bin/linker”，它指定了
Android 专用的动态链接器/system/bin/linker，而不是通常所用的 ld.so。
上面的“make clean-$(LOCAL_MODULE)”是 Android 编译环境提供的 make clean 的方式。

android 中应用程序使用的加载器
strings out/target/product/littleton/obj/EXECUTABLES/rild_intermediates/rild | grep link
/system/bin/linker
./prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/arm-eabi-gcc -dumpspecs|grep dynamic-linker
%{mbig-endian:-EB} %{mlittle-endian:-EL} %{static:-Bstatic} %{shared:-shared} %{symbolic:-Bsymbolic}
%{!static:%{shared: -Bsymbolic} %{!shared:%{rdynamic:-export-dynamic} %{!dynamic-linker:-dynamic-linker /system/bin/linker}}} -X

android 中加载器搜索库的路径
strings /nfsroot/rootfs/system/bin/linker | grep lib
/system/lib
/lib

生成的可执行程序可用 file 和 readelf 命令来查看一下：
file out/target/product/littleton/obj/EXECUTABLES/rild_intermediates/rild
out/target/product/littleton/obj/EXECUTABLES/rild_intermediates/rild: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), stripped
readelf -d out/target/product/littleton/obj/EXECUTABLES/rild_intermediates/rild |grep NEEDED
 0x00000001 (NEEDED) Shared library: [liblog.so]
 0x00000001 (NEEDED) Shared library: [libcutils.so]
 0x00000001 (NEEDED) Shared library: [libril.so]
 0x00000001 (NEEDED) Shared library: [libc.so]
 0x00000001 (NEEDED) Shared library: [libstdc++.so]
 0x00000001 (NEEDED) Shared library: [libm.so]
 0x00000001 (NEEDED) Shared library: [libdl.so]
这是 ARM 格式的动态链接可执行文件，运行时需要 libc.so 和 libm.so。“not stripped”表示它还没被 STRIP。嵌入式系统中为节省空间通常
将编译完成的可执行文件或动态库进行 STRIP，即去掉其中多余的符号表信息。在前面“make helloworld showcommands”命令的最后我们也
可以看到，Android 编译环境中使用了 out/host/linux-x86/bin/soslim 工具进行 STRIP。

4.3 Makefile 基本语法
Makefile 详解（超级好）
linux/Unix 环境下的 make 和 makefile 详解
http://www.unlinux.com/doc/program/20051026/2365.html
跟我一起写 Makefile
http://dev.csdn.net/develop/article/20/20025.shtm
==========

==========
5. 设置模块流程分析
rild 流程分析
5.1 设置 pin 状态，pin 认证
5.1.1 设置 pin 状态
5.1.2 修改 sim 卡 pin
5.1.3 pin 认证流程

5.2 网络设置
5.3 屏幕背光设置
5.4 获取，显示电池状态
==========
EditPinPreference.java (packages\apps\settings\src\com\android\settings)
private OnPinEnteredListener mPinListener;
protected void onDialogClosed(boolean positiveResult)
  mPinListener.onPinEntered(this, positiveResult);

执行　SimLockSettings.java (packages\apps\settings\src\com\android\settings)中函数：
public void onPinEntered(EditPinPreference preference, boolean positiveResult)
修改 pin 状态：　tryChangeSimLockState();
修改 pin：　　tryChangePin();

5.1.1 设置 pin 状态
```
private void tryChangeSimLockState()
   Message callback = Message.obtain(mHandler, ENABLE_SIM_PIN_COMPLETE);
   mPhone.getSimCard().setSimLockEnabled(mToState, mPin, callback);
```
进入 sim lock 菜单会显示初始化 pin 状态，是通过下面语句得到：
mPinToggle.setChecked(mPhone.getSimCard().getSimLockEnabled());

mPhone.getSimCard().setSimLockEnabled(mToState, mPin, callback)调用的是文件：
GsmSimCard.java (frameworks\base\telephony\java\com\android\internal\telephony\gsm)中的函数：
```
public void setSimLockEnabled (boolean enabled,String password, Message onComplete) {
   int serviceClassX;
   serviceClassX = CommandsInterface.SERVICE_CLASS_VOICE +
                   CommandsInterface.SERVICE_CLASS_DATA +
                   CommandsInterface.SERVICE_CLASS_FAX;
   mDesiredPinLocked = enabled;
   phone.mCM.setFacilityLock(CommandsInterface.CB_FACILITY_BA_SIM,
                   enabled, password, serviceClassX,
                   obtainMessage(EVENT_CHANGE_FACILITY_LOCK_DONE, onComplete));
```
phone.mCM.setFacilityLock 调用的是文件：
RIL.java (frameworks\base\telephony\java\com\android\internal\telephony\gsm)中的函数：
```
    public void
    setFacilityLock (String facility, boolean lockState, String password,
                                int serviceClass, Message response)
    {
        String lockString;
         RILRequest rr
                = RILRequest.obtain(RIL_REQUEST_SET_FACILITY_LOCK, response);
        if (RILJ_LOGD) riljLog(rr.serialString() + "> " + requestToString(rr.mRequest));
        // count strings
        rr.mp.writeInt(4);
        rr.mp.writeString(facility);
        lockString = (lockState)?"1":"0";
        rr.mp.writeString(lockString);
        rr.mp.writeString(password);
        rr.mp.writeString(Integer.toString(serviceClass));
        send(rr);
    }
```
设置应用程序向 rild 发送 RIL_REQUEST_SET_FACILITY_LOCK　请求的 socket 消息，
android 的初始源代码中 RIL_REQUEST_SET_FACILITY_LOCK 请求，在参考实现
Reference-ril.c
(hardware\ril\reference-ril) 中没有实现。
我们需要做得工作是：

==========

5.1.2 修改 sim 卡 pin
```
private void tryChangePin()
```

mPhone.getSimCard().changeSimLockPassword(mOldPin,mNewPin, callback);
mPhone.getSimCard 调用的是文件：
GsmSimCard.java (frameworks\base\telephony\java\com\android\internal\telephony\gsm)中的函数：
public void changeSimLockPassword(String oldPassword, String newPassword,
Message onComplete)
phone.mCM.changeSimPin(oldPassword, newPassword,
obtainMessage(EVENT_CHANGE_SIM_PASSWORD_DONE, onComplete));
phone.mCM.changeSimPin 调用的是文件：
RIL.java (frameworks\base\telephony\java\com\android\internal\telephony\gsm)中的函数：

```
public void
changeSimPin(String oldPin, String newPin, Message result)
{
    RILRequest rr = RILRequest.obtain(RIL_REQUEST_CHANGE_SIM_PIN, result);

    if (RILJ_LOGD) riljLog(rr.serialString() + "> " + requestToString(rr.mRequest));

    rr.mp.writeInt(2);
    rr.mp.writeString(oldPin);
    rr.mp.writeString(newPin);

    send(rr);
}
```

rild 端处理流程：

5.1.3 pin 认证流程

＝＝＝＝＝＝＝＝

5.2 网络设置

＝＝＝＝＝＝＝

5.3 屏幕背光设置
packages/apps/Settings/src/com/android/settings/BrightnessPreference.java
背光设置滚动条和关闭按钮都会调用 setBrightness(mOldBrightness);
public void onProgressChanged(SeekBar seekBar, int progress,boolean fromTouch)
protected void onDialogClosed(boolean positiveResult)

```
private void setBrightness(int brightness) {
    try {
        IHardwareService hardware = IHardwareService.Stub.asInterface(
                ServiceManager.getService("hardware"));
        if (hardware != null) {
            hardware.setBacklights(brightness);
        }
    } catch (RemoteException doe) {
    }
}
```

调用硬件服务器 HardwareService 的 setBacklights 函数
HardwareService.java (frameworks\base\services\java\com\android\server):
public void setBacklights(int brightness)

```
{
    . . .
    // Don't let applications turn the screen all the way off
    brightness = Math.max(brightness, Power.BRIGHTNESS_DIM);
```

```
            setLightBrightness_UNCHECKED(LIGHT_ID_BACKLIGHT, brightness);
            setLightBrightness_UNCHECKED(LIGHT_ID_KEYBOARD, brightness);
            setLightBrightness_UNCHECKED(LIGHT_ID_BUTTONS, brightness);
    . . .
}
void setLightOff_UNCHECKED(int light)
{
   //本地调用  setLight_native
            setLight_native(mNativePointer, light, 0, LIGHT_FLASH_NONE, 0, 0);
}
        void setLightBrightness_UNCHECKED(int light, int brightness) {
            int b = brightness & 0x000000ff;
            b = 0xff000000 | (b << 16) | (b << 8) | b;
            setLight_native(mNativePointer, light, b, LIGHT_FLASH_NONE, 0, 0);
        }
```

因为有：

```
com_android_server_HardwareService.cpp (frameworks\base\services\jni):
static JNINativeMethod method_table[] = {
        { "init_native", "()I", (void*)init_native },
        { "finalize_native", "(I)V", (void*)init_native },
        { "setLight_native", "(IIIIII)V", (void*)setLight_native },
        { "vibratorOn", "(J)V", (void*)vibratorOn },
        { "vibratorOff", "()V", (void*)vibratorOff }
};
```

所以最终调用的是文件：

com_android_server_HardwareService.cpp (frameworks\base\services\jni)中的函数：

```
static void setLight_native(JNIEnv *env, jobject clazz, int ptr,
            int light, int colorARGB, int flashMode, int onMS, int offMS)
{
        Devices* devices = (Devices*)ptr;
        light_state_t state;

        if (light < 0 || light >= LIGHT_COUNT || devices->lights[light] == NULL) {
            return ;
        }
        memset(&state, 0, sizeof(light_state_t));
        state.color = colorARGB;
        state.flashMode = flashMode;
        state.flashOnMS = onMS;
        state.flashOffMS = offMS;
        devices->lights[light]->set_light(devices->lights[light], &state);
}

Lights.h (hardware\libhardware\include\hardware):#define LIGHTS_HARDWARE_MODULE_ID
"lights"
com_android_server_HardwareService.cpp (frameworks\base\services\jni)
err = hw_get_module(LIGHTS_HARDWARE_MODULE_ID, (hw_module_t const**)&module);
static const char *variant_keys[] = {
        "ro.hardware",    /* This goes first so that it can pick up a different
                                file on the emulator. */
        "ro.product.board",
        "ro.board.platform",
        "ro.arch"
```

```
};
int hw_get_module(const char *id, const struct hw_module_t **module)
   status = load(id, prop, &hmi);
   status = load(id, HAL_DEFAULT_VARIANT, &hmi);
static int load(const char *id, const char *variant,const struct hw_module_t **pHmi)
    snprintf(path, sizeof(path), "%s/%s.%s.so", HAL_LIBRARY_PATH, id, variant);
#define HAL_DEFAULT_VARIANT         "default"
#define HAL_LIBRARY_PATH "/system/lib/hw"
```

所以 path 等于：

```
/system/lib/hw/light.marvell.so
/system/lib/hw/light.default.so
```

我们编译的 light 模块放在　/system/lib/hw/light.default.so 所以初始化成功。

```
property_get(variant_keys[i], prop, NULL) 只有 ro.hardware 存在 [ro.hardware]: [marvell]
static int lights_device_open(const struct hw_module_t* module, const char* name,struct
hw_device_t** device)
      dev->set_light = set_light_backlight;
static struct hw_module_methods_t lights_module_methods = {
      open: lights_device_open
};
hardware/libhardware/modules/lights/Android.mk
LOCAL_MODULE:= lights.default
err = module->methods->open(module, name, &device);
```

执行的是 ： lights_device_open

```
const char * const brightness_file = "/sys/class/backlight/micco-bl/brightness";
static int   set_light_backlight(struct light_device_t* dev,
                  struct light_state_t const* state)
 {
    . .  .
    color = state->color;
      tmp = ((77*((color>>16)&0x00ff)) + (150*((color>>8)&0x00ff)) + (29*(color&0x00ff))) >> 8;
      brightness = tmp/16;
      LOGD("---->calling %s(),line=%d state-
>color=%d,brightness=%d\n",__FUNCTION__,__LINE__,state->color,brightness);
      len = sprintf(buf,"%d",brightness);
      len = write(fd, buf, len);
      . . .
}
```

上面的函数完成了与内核的交互

综上所述，程序调用流程如下，上层应用通过 /sys/class/leds/lcd-backlight/brightnes 于内核打交道

设置模块 -> 硬件服务器 -> 本地调用 ->功能库 -> 读写　/sys/class/leds/lcd-backlight/brightness 函数与内核交互

```
Init.rc (vendor\marvell\littleton):      chown system system /sys/class/leds/keyboard-
backlight/brightness
Init.rc (vendor\marvell\littleton):      chown system system /sys/class/leds/lcd-backlight/brightness
Init.rc (vendor\marvell\littleton):      chown system system /sys/class/leds/button-backlight/brightness
```

5.4 获取，显示电池状态

电池状态（正在充电（AC））：

Status.java
String statusString;
mBatteryStatus.setSummary(statusString);

```
        public static final int BATTERY_STATUS_UNKNOWN = 1;
        public static final int BATTERY_STATUS_CHARGING = 2;
        public static final int BATTERY_STATUS_DISCHARGING = 3;
        public static final int BATTERY_STATUS_NOT_CHARGING = 4;
        public static final int BATTERY_STATUS_FULL = 5;

        // values for "health" field in the ACTION_BATTERY_CHANGED Intent
        public static final int BATTERY_HEALTH_UNKNOWN = 1;
        public static final int BATTERY_HEALTH_GOOD = 2;
        public static final int BATTERY_HEALTH_OVERHEAT = 3;
        public static final int BATTERY_HEALTH_DEAD = 4;
        public static final int BATTERY_HEALTH_OVER_VOLTAGE = 5;
        public static final int BATTERY_HEALTH_UNSPECIFIED_FAILURE = 6;
```

public static final int BATTERY_PLUGGED_AC = 1; 电源充电
public static final int BATTERY_PLUGGED_USB = 2; USB 充电

BatteryInfo.java (packages\apps\settings\src\com\android\settings)

电池级别（50％）
BatteryService.java (frameworks\base\services\java\com\android\server)
电池服务器：
构造函数：

```
public BatteryService(Context context)
   mUEventObserver.startObserving("SUBSYSTEM=power_supply");
```

－－－－－－－－－－
UEventObserver.java (frameworks\base\core\java\android\os)

```
void startObserving(String match)
   ensureThreadStarted();
      sThread = new UEventThread();
      sThread.start();
   sThread.addObserver(match, this);
```

－－－－－－－－－－

```
   update()
      native_update();
      sendIntent();

private final void sendIntent()
   Intent intent = new Intent(Intent.ACTION_BATTERY_CHANGED);
   intent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
   ...
   intent.putExtra("status", mBatteryStatus);
   intent.putExtra("health", mBatteryHealth);
   intent.putExtra("present", mBatteryPresent);
   intent.putExtra("level", mBatteryLevel);
   intent.putExtra("scale", BATTERY_SCALE);
   intent.putExtra("icon-small", icon);
   intent.putExtra("plugged", mPlugType);
   intent.putExtra("voltage", mBatteryVoltage);
```

```
    intent.putExtra("temperature", mBatteryTemperature);
    intent.putExtra("technology", mBatteryTechnology);
ActivityManagerNative.broadcastStickyIntent(intent, null);
```

把读取的电池信息通过广播信息发送给所有的应用程序。

native_update 本地调用的是文件 com_android_server_BatteryService.cpp (frameworks\base\services\jni) 中的函数：

```
static void android_server_BatteryService_update(JNIEnv* env, jobject obj)
{
    setBooleanField(env, obj, AC_ONLINE_PATH, gFieldIds.mAcOnline);
    setBooleanField(env, obj, USB_ONLINE_PATH, gFieldIds.mUsbOnline);
    setBooleanField(env, obj, BATTERY_PRESENT_PATH, gFieldIds.mBatteryPresent);

    setIntField(env, obj, BATTERY_CAPACITY_PATH, gFieldIds.mBatteryLevel);
    setIntField(env, obj, BATTERY_VOLTAGE_PATH, gFieldIds.mBatteryVoltage);
    setIntField(env, obj, BATTERY_TEMPERATURE_PATH, gFieldIds.mBatteryTemperature);

    const int SIZE = 128;
    char buf[SIZE];

    if (readFromFile(BATTERY_STATUS_PATH, buf, SIZE) > 0)
        env->SetIntField(obj, gFieldIds.mBatteryStatus, getBatteryStatus(buf));

    if (readFromFile(BATTERY_HEALTH_PATH, buf, SIZE) > 0)
        env->SetIntField(obj, gFieldIds.mBatteryHealth, getBatteryHealth(buf));

    if (readFromFile(BATTERY_TECHNOLOGY_PATH, buf, SIZE) > 0)
        env->SetObjectField(obj, gFieldIds.mBatteryTechnology, env->NewStringUTF(buf));
}
#define AC_ONLINE_PATH "/sys/class/power_supply/ac/online"
#define USB_ONLINE_PATH "/sys/class/power_supply/usb/online"
#define BATTERY_STATUS_PATH "/sys/class/power_supply/battery/status"
#define BATTERY_HEALTH_PATH "/sys/class/power_supply/battery/health"
#define BATTERY_PRESENT_PATH "/sys/class/power_supply/battery/present"
#define BATTERY_CAPACITY_PATH "/sys/class/power_supply/battery/capacity"
#define BATTERY_VOLTAGE_PATH "/sys/class/power_supply/battery/batt_vol"
#define BATTERY_TEMPERATURE_PATH "/sys/class/power_supply/battery/batt_temp"
#define BATTERY_TECHNOLOGY_PATH "/sys/class/power_supply/battery/technology"
```

==========

==========

6. linux 系统启动流程分析

6.1 桌面操作系统启动流程（redhat，federa，ubuntu）

6.2 小型嵌入式系统启动流程

6.3 android 系统启动流程

==========

6.1 桌面操作系统启动流程（redhat，federa，ubuntu）

ubuntu 从 6.10 开始逐步用 upstart 代替原来的 sysinit，进行服务进程的管理。为了对原有的 init 实现向后兼容，

目前 ubuntu 中与 init 相关的几个目录和应用程序，可以方便后面的论述。这些目录和程序包括：
init
telinit　//字面理解 tell init
runlevel
/etc/event.d/
/etc/init.d/
/etc/rcX.d/
首先是/etc/event.d/目录，这是 upstart 的核心，upstart 不同于原有的 init 的地方就在于它引入了 event 机制。Event 机制通俗的
讲就是将所有进程的触发、停止等等都看作 event(事件)。/etc/event.d/中就存放了目前 upstart 需要识别的 event。这其中主要有三种
rc-default, rcX(x=0,1,...6,S)以及 ttyX。这 rc-default 就类似于 inittab 文件，它就是设置默认运行级别的 ，需要运行程序的
脚本，而 ttyX 则是设置伪终端数目的，也就是你 Ctrl+Alt+F(1~6)调出的那个 Console。我们以 rc2 为例，cat rc2:
rc-default
start on stopped rcS
telinit 2
所以会依次执行 /etc/event.d/rcS /etc/event.d/rc2
它们又会分别执行：
exec /etc/init.d/rc S
exec /etc/init.d/rc 2

这样，我们就可以自然地过渡到下一个重要的目录，/etc/init.d/了。
/etc/init.d/中存放的是服务(services)或者任务(tasks)的执行脚本。可以这么说，只要你安装了一个程序(特别是服务程序 daemon)，
它可以在系统启动的时候运行，那么它必定会在/etc/init.d/中有一个脚本文件。
执行了一个 exec /etc/init.d/rc 2 的命令。也就是说，给/etc/init.d/rc 脚本传递了一个参数"2"，让它执行。
rc 脚本(很长，耐心点)，能看到这样的一段：
Now run the START scripts for this runlevel.
Run all scripts with the same level in parallel
.....
for s in /etc/rc$runlevel.d/S*
.....
将会开始执行/etc/rc2.d/下 S 开头的脚本。这就过渡到下一个目录/etc/rc2.d/了。
/etc/rc2.d 都是一些到/etc/init.d/中脚本的符号链接。不同的是在开头加上了 S 和一个数字，S 表示在启动时运行，数字则表示执行的先后顺序。
/etc/rcS.d/S35mountall.sh
K08vmware
S19vmware
S20nfs-common
S20nfs-kernel-server
S20samba
S20xinetd
S30gdm
S98usplash
S99rc.local
总结：

这样一来，upstart 管理的 ubuntu 启动过程应该就清楚了。梳理一下：

1,内核启动 init

2,init 找到/etc/event.d/rc-default 文件，确定默认的运行级别(X)

3,触发相应的 runlevel 事件，开始运行/etc/event.d/rcX

4,rcX 运行/etc/init.d/rc，传入参数 X

5,/etc/init.d/rc 脚本进行一系列设置，最后运行相应的/etc/rcX.d/中的脚本

6,/etc/rcX.d/中的脚本按事先设定的优先级依次启动，直至最后给出登录画面(启动 X 服务器和 GDM)

理解了这些，手动配置开机服务的启动与否就很简单了。Ubutnu 默认的启动级别是 2，不想启动的程序，只要把相应的符号链接从/etc/rc2.d/中删去即可

注意：

想 redat ， federa 这些系统，他们用的是 sysvinit ，有 /etc/inittab 文件，里面定义了 ：

id:5:initdefault:

si::sysinit:/etc/init.d/rcS

init 直接解析 id:5:initdefault 字段，然后执行 /etc/rc5.d/ 下面的脚本

==========

参考文档：

linux 教程:upstart 和 ubuntu 启动过程原理介绍

http://www.zhiweinet.com/jiaocheng/2009-06/12500.htm

6.2 小型嵌入式系统启动流程

小型嵌入式的 init 通常使用 busybox 中自带的，

6.3 android 系统启动流程

参考文档：

init 是内核进入文件系统后第一个运行的程序，我们可以在 linux 的命令行中进行指定，如果没指定，内核将会到/sbin/, /bin/ 等目录下

查找默认的 init，如果没有找到那么就报告出错。

init 源代码分析

init 的 mian 函数在文件：./system/core/init/init.c 中，init 会一步步完成下面的任务：

1.初始化 log 系统

2.解析/init.rc 和/init.%hardware%.rc 文件

3. 执行 early-init action in the two files parsed in step 2.

4. 设备初始化，例如：在 /dev 下面创建所有设备节点，下载 firmwares.

5. 初始化属性服务器，Actually the property system is working as a share memory. Logically it looks like a registry under Windows system.

6. 执行 init action in the two files parsed in step 2.

7. 开启 属性服务。

8. 执行 early-boot and boot actions in the two files parsed in step 2.

9. 执行 Execute property action in the two files parsed in step 2.

10. 进入一个无限循环 to wait for device/property set/child process exit events.例如,如果 SD 卡被插入，init 会收到一个设备插入事件，

它会为这个设备创建节点。系统中比较重要的进程都是由 init 来 fork 的，所以如果他们他谁崩溃了，那么 init 将会收到一个 SIGCHLD 信号，把这个信号转化

为子进程退出事件， 所以在 loop 中，init 会操作进程退出事件并且执行 *.rc 文件中定义的命令。
例如，在 init.rc 中，因为有：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
```

所以，如果 zygote 因为启动某些服务导致异常退出后，init 将会重新去启动它。

```
int main(int argc, char **argv)
{
    ...
    //需要在后面的程序中看打印信息的话，需要屏蔽 open_devnull_stdio()函数
    open_devnull_stdio();
    ...
    //初始化 log 系统
    log_init();
    //解析/init.rc 和/init.%hardware%.rc 文件
    parse_config_file("/init.rc");
    ...
    snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
    parse_config_file(tmp);
    ...
    //执行  early-init action in the two files parsed in step 2.
    action_for_each_trigger("early-init", action_add_queue_tail);
    drain_action_queue();
    ...
    /* execute all the boot actions to get us started */
    /*  执行  init action in the two files parsed in step 2 */
    action_for_each_trigger("init", action_add_queue_tail);
    drain_action_queue();
    ...
    /*  执行  early-boot and boot actions in the two files parsed in step 2 */
    action_for_each_trigger("early-boot", action_add_queue_tail);
    action_for_each_trigger("boot", action_add_queue_tail);
    drain_action_queue();

    /* run all property triggers based on current state of the properties */
    queue_all_property_triggers();
    drain_action_queue();

    /* enable property triggers */
    property_triggers_enabled = 1;
    ...
    for(;;) {
        int nr, timeout = -1;
    ...
        drain_action_queue();
        restart_processes();

        if (process_needs_restart) {
            timeout = (process_needs_restart - gettime()) * 1000;
```

```
            if (timeout < 0)
                timeout = 0;
        }
    ...
        nr = poll(ufds, 3, timeout);
        if (nr <= 0)
            continue;

        if (ufds[2].revents == POLLIN) {
            /* we got a SIGCHLD - reap and restart as needed */
            read(signal_recv_fd, tmp, sizeof(tmp));
            while (!wait_for_one_process(0))
                ;
            continue;
        }

        if (ufds[0].revents == POLLIN)
            handle_device_fd(device_fd);

        if (ufds[1].revents == POLLIN)
    {
            handle_property_set_fd(property_set_fd);
    }
    }

    return 0;
}
```

解析 init.rc 脚本

init.rc 脚本的具体语法可以参考下面文档

http://www.kandroid.org/android_pdk/bring_up.html

init.rc 脚本语法

Android 初始化語言由四大类声明组成：行为类(Actions),命令类(Commands)，服务类(Services),选项类(Options).

初始化语言以行为单位，由以空格间隔的语言符号組成。C 风格的反斜杠转义符可以用来插入空白到语言符号。双引号也可以用来防止

文本被空格分成多个语言符号。当反斜杠在行末时，作为换行符。

* 以#开始(前面允许空格)的行为注释。

* Actions 和 Services 隐含声明一个新的段落。所有该段落下 Commands 或 Options 的声明属于该段落。第一段落前的 Commands

或 Options 被忽略。

* Actions 和 Services 拥有唯一的命名。在他们之后声明相同命名的类将被当作错误并忽略。

Actions 是一系列命令的命名。Actions 拥有一个触发器(trigger)用来决定 action 何時执行。当一个 action 在符合触发条件被执行时，

如果它还没被加入到待执行队列中的话，则加入到队列最后。

队列中的 action 依次执行，action 中的命令也依次执行。Init 在执行命令的中间处理其他活动(设备创建/销毁,property 设置，进程重启)。

Actions 的表现形式:

```
on <trigger>
   <command>
   <command>
```

<command>
重要的数据结构
两个列表，一个队列。
static list_declare(service_list);
static list_declare(action_list);
static list_declare(action_queue);
*.rc 脚本中所有 service 关键字定义的服务将会添加到 service_list 列表中。
*.rc 脚本中所有 on 关键开头的项将会被会添加到 action_list 列表中。
每个 action 列表项都有一个列表，此列表用来保存该段落下的 Commands

脚本解析过程
parse_config_file("/init.rc")

```
int parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);
    if (!data) return -1;

    parse_config(fn, data);
    DUMP();
    return 0;
}
static void parse_config(const char *fn, char *s)
 {
    ...
    case T_NEWLINE:
        if (nargs) {
            int kw = lookup_keyword(args[0]);
            if (kw_is(kw, SECTION)) {
                state.parse_line(&state, 0, 0);
                parse_new_section(&state, kw, nargs, args);
            } else {
                state.parse_line(&state, nargs, args);
            }
            nargs = 0;
        }
  ...
}
```

parse_config 会逐行对脚本进行解析，如果关键字类型为 SECTION ，那么将会执行 parse_new_section()
类型为 SECTION 的关键字有： on 和 sevice
关键字类型定义在 Parser.c (system\core\init) 文件中
Parser.c (system\core\init)

```
#define SECTION 0x01
#define COMMAND 0x02
#define OPTION  0x04
关键字              属性
capability,   OPTION,   0, 0)
class,          OPTION,   0, 0)
class_start, COMMAND, 1, do_class_start)
class_stop,   COMMAND, 1, do_class_stop)
```

```
console,        OPTION,    0, 0)
critical,       OPTION,    0, 0)
disabled,       OPTION,    0, 0)
domainname,    COMMAND, 1, do_domainname)
exec,           COMMAND, 1, do_exec)
export,         COMMAND, 2, do_export)
group,          OPTION,    0, 0)
hostname,       COMMAND, 1, do_hostname)
ifup,           COMMAND, 1, do_ifup)
insmod,         COMMAND, 1, do_insmod)
import,         COMMAND, 1, do_import)
keycodes,       OPTION,    0, 0)
mkdir,          COMMAND, 1, do_mkdir)
mount,          COMMAND, 3, do_mount)
on,             SECTION, 0, 0)
oneshot,        OPTION,    0, 0)
onrestart,     OPTION,    0, 0)
restart,        COMMAND, 1, do_restart)
service,        SECTION, 0, 0)
setenv,         OPTION,    2, 0)
setkey,         COMMAND, 0, do_setkey)
setprop,        COMMAND, 2, do_setprop)
setrlimit,     COMMAND, 3, do_setrlimit)
socket,         OPTION,    0, 0)
start,          COMMAND, 1, do_start)
stop,           COMMAND, 1, do_stop)
trigger,        COMMAND, 1, do_trigger)
symlink,        COMMAND, 1, do_symlink)
sysclktz,       COMMAND, 1, do_sysclktz)
user,           OPTION,    0, 0)
write,          COMMAND, 2, do_write)
chown,          COMMAND, 2, do_chown)
chmod,          COMMAND, 2, do_chmod)
loglevel,       COMMAND, 1, do_loglevel)
device,         COMMAND, 4, do_device)
```

parse_new_section()中再分别对 service 或者 on 关键字开头的内容进行解析。

```
    ...
    case K_service:
        state->context = parse_service(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on:
        state->context = parse_action(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_action;
            return;
        }
        break;
    }
```

```
    ...

对 on 关键字开头的内容进行解析
static void *parse_action(struct parse_state *state, int nargs, char **args)
{
    ...
    act = calloc(1, sizeof(*act));
    act->name = args[1];
    list_init(&act->commands);
    list_add_tail(&action_list, &act->alist);
    ...
}

对 service 关键字开头的内容进行解析
static void *parse_service(struct parse_state *state, int nargs, char **args)
{
    struct service *svc;
    if (nargs < 3) {
        parse_error(state, "services must have a name and a program\n");
        return 0;
    }
    if (!valid_name(args[1])) {
        parse_error(state, "invalid service name '%s'\n", args[1]);
        return 0;
    }
    //如果服务已经存在 service_list 列表中将会被忽略
    svc = service_find_by_name(args[1]);
    if (svc) {
        parse_error(state, "ignored duplicate definition of service '%s'\n", args[1]);
        return 0;
    }

    nargs -= 2;
    svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);
    if (!svc) {
        parse_error(state, "out of memory\n");
        return 0;
    }
    svc->name = args[1];
    svc->classname = "default";
    memcpy(svc->args, args + 2, sizeof(char*) * nargs);
    svc->args[nargs] = 0;
    svc->nargs = nargs;
    svc->onrestart.name = "onrestart";
    list_init(&svc->onrestart.commands);
    //添加该服务到 service_list 列表
    list_add_tail(&service_list, &svc->slist);
    return svc;
}
服务的表现形式:
service <name> <pathname> [ <argument> ]*
<option>
<option>
```

...

申请一个 service 结构体,然后挂接到 service_list 链表上,name 为服务的名称 pathname 为执行的命令 argument
为命令的参数。之后的 option 用来控制这个 service 结构体的属性,parse_line_service 会对 service 关键字后的
内容进行解析并填充到 service 结构中 ， 当遇到下一个 service 或者 on 关键字的时候此 service 选项解析结束。

例如：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
```

服务名称为： zygote
启动该服务执行的命令： /system/bin/app_process
命令的参数： -Xzygote /system/bin --zygote --start-system-server
socket zygote stream 666： 创建一个名为：/dev/socket/zygote 的 socket ， 类型为：stream

当*.rc 文件解析完成以后：
action_list 列表项目如下：
on init
on boot
on property:ro.kernel.qemu=1
on property:persist.service.adb.enable=1
on property:persist.service.adb.enable=0
init.marvell.rc 文件
on early-init
on init
on early-boot
on boot

service_list 列表中的项有：
service console
service adbd
service servicemanager
service mountd
service debuggerd
service ril-daemon
service zygote
service media
service bootsound
service dbus
service hcid
service hfag
service hsag
service installd
service flash_recovery

设备初始化
early-init 初始化
初始化属性服务器

在 init.c 的 main 函数中启动状态服务器。
property_set_fd = start_property_service();
状态读取函数：
Property_service.c (system\core\init)
const char* property_get(const char *name)
Properties.c (system\core\libcutils)
int property_get(const char *key, char *value, const char *default_value)
状态设置函数：
Property_service.c (system\core\init)
int property_set(const char *name, const char *value)
Properties.c (system\core\libcutils)
int property_set(const char *key, const char *value)

在终端模式下我们可以通过执行命令 setprop <key> <value>
setprop 工具源代码所在文件： Setprop.c (system\core\toolbox)
Getprop.c (system\core\toolbox): property_get(argv[1], value, default_value);
Property_service.c (system\core\init)
中定义的状态读取和设置函数仅供 init 进程调用，

handle_property_set_fd(property_set_fd);
  property_set() //Property_service.c (system\core\init)
    property_changed(name, value) //Init.c (system\core\init)
      queue_property_triggers(name, value)
      drain_action_queue()
只要属性一改变就会被触发，然后执行相应的命令：
例如：
在 init.rc 文件中有
on property:persist.service.adb.enable=1
  start adbd
on property:persist.service.adb.enable=0
  stop adbd
所以如果在终端下输入：
setprop property:persist.service.adb.enable 1 或者 0
那么将会开启或者关闭 adbd 程序。

执行 action_list 中的命令：
从 action_list 中取出 act->name 为 early-init 的列表项，再调用 action_add_queue_tail(act)将其插入到
队列 action_queue 尾部。drain_action_queue() 从 action_list 队列中取出队列项 ，然后执行 act->commands
列表中的所有命令。
所以从 ./system/core/init/init.c mian()函数的程序片段：
action_for_each_trigger("early-init", action_add_queue_tail);
drain_action_queue();
action_for_each_trigger("init", action_add_queue_tail);
drain_action_queue();
action_for_each_trigger("early-boot", action_add_queue_tail);
action_for_each_trigger("boot", action_add_queue_tail);

```
drain_action_queue();
/* run all property triggers based on current state of the properties */
queue_all_property_triggers();
drain_action_queue();
```

可以看出，在解析完 init.rc init.marvell.rc 文件后，action 命令执行顺序为：
执行 act->name 为 early-init，act->commands 列表中的所有命令
执行 act->name 为 init， act->commands 列表中的所有命令
执行 act->name 为 early-boot，act->commands 列表中的所有命令
执行 act->name 为 boot， act->commands 列表中的所有命令

关键的几个命令：
class_start default 启动所有 service 关键字定义的服务。
class_start 在 act->name 为 boot 的 act->commands 列表中，所以当 class_start 被触发后，实际上调用的是函数 do_class_start（）

```
int do_class_start(int nargs, char **args)
{
        /* Starting a class does not start services
         * which are explicitly disabled.  They must
         * be started individually.
         */
    service_for_each_class(args[1], service_start_if_not_disabled);
    return 0;
}
void service_for_each_class(const char *classname,
                            void (*func)(struct service *svc))
{
    struct listnode *node;
    struct service *svc;
    list_for_each(node, &service_list) {
        svc = node_to_item(node, struct service, slist);
        if (!strcmp(svc->classname, classname)) {
            func(svc);
        }
    }
}
```

因为在调用 parse_service() 添加服务列表的时候，所有服务 svc->classname 默认取值："default"，
所以 service_list 中的所有服务将会被执行。
参考文档：
http://blog.chinaunix.net/u1/38994/showart_1775465.html
http://blog.chinaunix.net/u1/38994/showart_1168440.html
浅析 kernel 启动的第 1 个用户进程 init 如何解读 init.rc 脚本
http://blog.chinaunix.net/u1/38994/showart_1168440.html

Zygote 服务概论：
Zygote 是 android 系统中最重要的一个服务，它将一步一步完成下面的任务：
start Android Java Runtime and start system server. It's the most important service. The source is in device/servers/app.

1. 创建 JAVA 虚拟机
2. 为 JAVA 虚拟机注册 android 本地函数
3. 调用 com.android.internal.os.ZygoteInit 类中的 main 函数，
android/com/android/internal/os/ZygoteInit.java.

a) 装载 ZygoteInit 类
b) 注册 zygote socket
c) 装载 preload classes(the default file is device/java/android/preloaded-classes)
d) 装载 Load preload 资源
e) 调用 Zygote::forkSystemServer (定义在./dalvik/vm/InternalNative.c)来 fork 一个新的进程，在新进程中调用
com.android.server.SystemServer 的 main 函数。

a) 装载 libandroid_servers.so 库
bb) 调用 JNI native init1 函数
（device/libs/android_servers/com_android_server_SystemServers）
Load libandroid_servers.so
Call JNI native init1 function implemented in
device/libs/android_servers/com_android_server_SystemServers.
It only calls system_init implemented in device/servers/system/library/system_init.cpp.
If running on simulator, instantiate AudioFlinger, MediaPlayerService and CameraService here.
Call init2 function in JAVA class named com.android.server.SystemServer, whose source is in device/java/services/com/android/server. This function is very critical for Android because it start all of
Android JAVA services.
If not running on simulator, call IPCThreadState::self()->joinThreadPool() to enter into service dispatcher.

SystemServer::init2 将会启动一个新的线程来启动下面的所有 JAVA 服务：
Core 服务:

1. Starting Power Manager(电源管理）
2. Creating Activity Manager（活动服务）
3. Starting Telephony Registry（电话注册服务）
4. Starting Package Manager（包管理器）
5. Set Activity Manager Service as System Process
6. Starting Context Manager
7. Starting System Context Providers
8. Starting Battery Service（电池服务）
9. Starting Alarm Manager（闹钟服务）
10. Starting Sensor Service
11. Starting Window Manager（启动窗口管理器）
12. Starting Bluetooth Service（蓝牙服务）
13. Starting Mount Service

其他 services：

1. Starting Status Bar Service（状态服务）
2. Starting Hardware Service（硬件服务）
3. Starting NetStat Service（网络状态服务）
4. Starting Connectivity Service
5. Starting Notification Manager
6. Starting DeviceStorageMonitor Service
7. Starting Location Manager

8. Starting Search Service（查询服务）
9. Starting Clipboard Service
10. Starting Checkin Service
11. Starting Wallpaper Service
12. Starting Audio Service
13. Starting HeadsetObserver
14. Starting AdbSettingsObserver

最后 SystemServer::init2 将会调用 ActivityManagerService.systemReady 通过发送 Intent.CATEGORY_HOME intent 来启动第一个 activity.还有另外一种启动 system server 的方法是：
通过名为 system_server 的程序（源代码：device/servers/system/system_main.cpp）它也是通过调用 system_init 来启动 system services，这时候就有个问题：为什么 android 有两种方式启动 system services？
我的猜想是：
My guess is that directly start system_server may have synchronous problem with zygote because system_server will call JNI to start SystemServer::init2, while at that time zygote may not start JAVA VM yet. So Android uses another method. After zynote is initialized, fork a new process to start system services.

Zygote 服务启动的详细过程：
通过启动服务列表的 app_process 进程，实际上进入的是
App_main.cpp (frameworks\base\cmds\app_process)
main()
根据 init.rc 中的 --zygote --start-system-server
分别调用的是
runtime.start("com.android.internal.os.ZygoteInit",startSystemServer);
或者
runtime.start();
start()函数在 AndroidRuntime.cpp (frameworks\base\core\jni)文件中
从打印信息：
D/AndroidRuntime(56): >>>>>>>>>>>>>> AndroidRuntime START <<<<<<<<<<<<<<
也可以看出调用的是：
AndroidRuntime::start(const char* className, const bool startSystemServer)
JNI_CreateJavaVM()
startReg()
LOGD("--- registering native functions ---\n");
startMeth = env->GetStaticMethodID(startClass, "main","([Ljava/lang/String;)V");
从上面的调用可以看出一类引用的过程都是从 main 方法
所以接着调用了 ZygoteInit 类的 main 方法
main 方法主要完成：
1.Register zygote socket, Registers a server socket for zygote command connections
2.Load preload classes(the default file is device/java/android/preloaded-classes).
3.Load preload resources, Load in commonly used resources, so they can be shared across processes.
4.Start SystemServer, Prepare the arguments and fork for the system server process.
具体执行过程如下：
ZygoteInit.java (frameworks\base\core\java\com\android\internal\os)中的 mian
main()
registerZygoteSocket()

```
preloadClasses()
  loadLibrary()
    Log.i(TAG, "Preloading classes...");
  Runtime.loadLibrary
    Dalvik_java_lang_Runtime_nativeLoad()
      dvmLoadNativeCode()
        LOGD("Trying to load lib %s %p\n", pathName, classLoader);
        System.loadLibrary("media_jni");
preloadResources();
startSystemServer()
  Zygote.forkSystemServer(parsedArgs.uid, parsedArgs.gid,parsedArgs.gids, debugFlags, null);
  //Zygote.java (dalvik\libcore\dalvik\src\main\java\dalvik\system)
    forkSystemServer()
      forkAndSpecialize()     //Zygote.java (dalvik\libcore\dalvik\src\main\java\dalvik\system)
        Dalvik_dalvik_system_Zygote_forkAndSpecialize() //dalvik_system_Zygote.c
(dalvik\vm\native)
          Dalvik_dalvik_system_Zygote_forkAndSpecialize()
            setSignalHandler()
            fork()
    handleSystemServerProcess() //handleChildProc(parsedArgs, descriptors, newStderr);
      closeServerSocket();
    RuntimeInit.zygoteInit(parsedArgs.remainingArgs);
      zygoteInit()              //RuntimeInit.java
(frameworks\base\core\java\com\android\internal\os)
        zygoteInitNative()
          invokeStaticMain()
            System.loadLibrary("android_servers");
            //com.android.server.SystemServer startSystemServer()  函数中
            m = cl.getMethod("main", new Class[] { String[].class });
              //执行的是 SystemServer  类的 main 函数  SystemServer.java
(frameworks\base\services\java\com\android\server)
              init1() //SystemServer.java (frameworks\base\services\java\com\android\server)

                //init1()实际上是调用 android_server_SystemServer_init1(JNIEnv* env,
jobject clazz)
                //com_android_server_SystemServer.cpp (frameworks\base\services\jni)
              android_server_SystemServer_init1()//JNI  调用
                system_init() //System_init.cpp (frameworks\base\cmds\system_server\library)
                  // Start the SurfaceFlinger
                  SurfaceFlinger::instantiate();
                  //Start the AudioFlinger media playback    camera service
                    AudioFlinger::instantiate();
                    MediaPlayerService::instantiate();
                    CameraService::instantiate();
                    //调用  SystemServer  类的 init2
                    runtime->callStatic("com/android/server/SystemServer", "init2");
                    init2()//SystemServer.java
(frameworks\base\services\java\com\android\server)
                    ServerThread()
                      run()//在 run 中启动电源管理,蓝牙,等核心服务以及状态,查找等其他
服务
```

((ActivityManagerService)ServiceManager.getService("activity")).setWindowManager(wm);
...
ActivityManagerNative.getDefault().systemReady();
runSelectLoopMode();
done = peers.get(index).runOnce();
forkAndSpecialize() //Zygote.java (dalvik\libcore\dalvik\src\main\java\dalvik\system)
Dalvik_dalvik_system_Zygote_forkAndSpecialize() //dalvik_system_Zygote.c (dalvik\vm\native)
forkAndSpecializeCommon()
setSignalHandler()
RETURN_INT(pid);
closeServerSocket();

见附 A
主進程 runSelectLoopMode()
5.Runs the zygote process's select loop runSelectLoopMode(), Accepts new connections as they happen, and
reads commands from connections one spawn-request's worth at a time.
如果运行正常，则 zygote 进程会在 runSelectLoopMode()中循环：

zygote 被 siganl(11)终止
在 dalvik_system_Zygote.c (dalvik\vm\native)
的 static void sigchldHandler(int s) 函数中打印：
"Process %d terminated by signal (%d)\n",
"Exit zygote because system server (%d) has terminated\n",

startSystemServer() ZygoteInit.java (frameworks\base\core\java\com\android\internal\os)
SystemServer 的 mian()函数会调用
SystemServer.java (frameworks\base\services\java\com\android\server)中的 init1()函数。
init1()实际执行的是 com_android_server_SystemServer.cpp (frameworks\base\services\jni)
中的 android_server_SystemServer_init1()。
android_server_SystemServer_init1()调用的是
System_init.cpp (frameworks\base\cmds\system_server\library) 中的 system_init()函数
system_init()函数定义如下：

```
extern "C" status_t system_init()
{
    ...
    sp<IServiceManager> sm = defaultServiceManager();
    ...
    property_get("system_init.startsurfaceflinger", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        //读取属性服务器,开启启动 SurfaceFlinger 服务
        //接着会开始显示机器人图标
        //BootAnimation.cpp (frameworks\base\libs\surfaceflinger):status_t BootAnimation::readyToRun()
        SurfaceFlinger::instantiate();
```

```
        }
        //在模拟器上 audioflinger 等几个服务与设备上的启动过程不一样，所以
        //我们在这里启动他们。

        if (!proc->supportsProcesses()) {
            //启动 AudioFlinger，media playback service，camera service 服务
            AudioFlinger::instantiate();
            MediaPlayerService::instantiate();
            CameraService::instantiate();
        }
        //现在开始运行 the Android runtime ，我们这样做的目的是因为必须在 core system
services
        //起来以后才能 Android runtime initialization，其他服务在调用他们自己的 main()时，都会
        //调用 Android runtime
        //before calling the init function.
        LOGI("System server: starting Android runtime.\n");
        AndroidRuntime* runtime = AndroidRuntime::getRuntime();
        LOGI("System server: starting Android services.\n");
        //调用 SystemServer.java (frameworks\base\services\java\com\android\server)
        //中的 init2 函数
        runtime->callStatic("com/android/server/SystemServer", "init2");

        // If running in our own process, just go into the thread
        // pool.    Otherwise, call the initialization finished
        // func to let this process continue its initilization.
        if (proc->supportsProcesses()) {
            LOGI("System server: entering thread pool.\n");
            ProcessState::self()->startThreadPool();
            IPCThreadState::self()->joinThreadPool();
            LOGI("System server: exiting thread pool.\n");
        }
        return NO_ERROR;
}
```

System server: entering thread pool 表明已经进入服务线程 ServerThread
在 ServerThread 类的 run 服务中开启核心服务：

```
        @Override
        public void run() {
            EventLog.writeEvent(LOG_BOOT_PROGRESS_SYSTEM_RUN,
                    SystemClock.uptimeMillis());

            ActivityManagerService.prepareTraceFile(false);        // create dir

            Looper.prepare();
        //设置线程的优先级
            android.os.Process.setThreadPriority(
                        android.os.Process.THREAD_PRIORITY_FOREGROUND);
        ...
            //关键（核心）服务
            try {
```

```
            Log.i(TAG, "Starting Power Manager.");
            Log.i(TAG, "Starting     activity Manager.");
            Log.i(TAG, "Starting telephony registry");
            Log.i(TAG, "Starting Package Manager.");
            Log.i(TAG, "    tarting Content Manager.");
            Log.i(TAG, "Starting System Content Providers.");
            Log.i(TAG, "Starting Battery Service.");
            Log.i(TAG, "Starting Alarm Manager.");
            Log.i(TAG, "Starting Sensor Service.");
            Log.i(TAG, "Starting Window Manager.");
            Log.i(TAG, "Starting Bluetooth Service.");
            //如果是模拟器，那么跳过蓝牙服务。
            // Skip Bluetooth if we have an emulator kernel
         //其他的服务
            Log.i(TAG, "Starting Status Bar Service.");
            Log.i(TAG, "Starting Clipboard Service.");
            Log.i(TAG, "Starting Input Method Service.");
            Log.i(TAG, "Starting Hardware Service.");
            Log.i(TAG, "Starting NetStat Service.");
            Log.i(TAG, "Starting Connectivity Service.");
            Log.i(TAG, "Starting Notification Manager.");
            // MountService must start after NotificationManagerService
            Log.i(TAG, "Starting Mount Service.");
        Log.i(TAG, "Starting DeviceStorageMonitor service");
            Log.i(TAG, "Starting Location Manager.");
            Log.i(TAG, "Starting Search Service.");
            ...
            if (INCLUDE_DEMO) {
                Log.i(TAG, "Installing demo data...");
                (new DemoThread(context)).start();
            }
            try {
                Log.i(TAG, "Starting Checkin Service.");
                Intent intent = new Intent().setComponent(new ComponentName(
                        "com.google.android.server.checkin",
                        "com.google.android.server.checkin.CheckinService"));
                if (context.startService(intent) == null) {
                    Log.w(TAG, "Using fallback Checkin Service.");
                    ServiceManager.addService("checkin", new
FallbackCheckinService(context));
                }
            } catch (Throwable e) {
                Log.e(TAG, "Failure starting Checkin Service", e);
            }


            Log.i(TAG, "Starting Wallpaper Service");
        Log.i(TAG, "Starting Audio Service");
            Log.i(TAG, "Starting HeadsetObserver");
            Log.i(TAG, "Starting AppWidget Service");
        ...
            try {
                com.android.server.status.StatusBarPolicy.installIcons(context, statusBar);
```

```
                } catch (Throwable e) {
                    Log.e(TAG, "Failure installing status bar icons", e);
                }
            }

            // make sure the ADB_ENABLED setting value matches the secure property value
            Settings.Secure.putInt(mContentResolver, Settings.Secure.ADB_ENABLED,
                    "1".equals(SystemProperties.get("persist.service.adb.enable")) ? 1 : 0);

            // register observer to listen for settings changes

mContentResolver.registerContentObserver(Settings.Secure.getUriFor(Settings.Secure.ADB_ENABL
ED),
                    false, new AdbSettingsObserver());

            // It is now time to start up the app processes...
            boolean safeMode = wm.detectSafeMode();
            if (statusBar != null) {
                statusBar.systemReady();
            }
            if (imm != null) {
                imm.systemReady();
            }
            wm.systemReady();
            power.systemReady();
            try {
                pm.systemReady();
            } catch (RemoteException e) {
            }
            if (appWidget != null) {
                appWidget.systemReady(safeMode);
            }

            // After making the following code, third party code may be running...
            try {
                ActivityManagerNative.getDefault().systemReady();
            } catch (RemoteException e) {
            }

            Watchdog.getInstance().start();

            Looper.loop();
            Log.d(TAG, "System ServerThread is exiting!");
        }


startActivity()
   mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);


ActivityManagerService.java 3136p (frameworks\base\services\java\com\android\server\am)
   startActivity()
```

```
        startActivityLocked() //3184
        int res = startActivityLocked(caller, intent, resolvedType,grantedUriPermissions, grantedMode,
aInfo,
                                      resultTo, resultWho, requestCode, -1, -1,
                                      onlyIfNeeded, componentSpecified);


public abstract class ActivityManagerNative extends Binder implements IActivityManager
ActivityManagerService.java 1071p    (frameworks\base\services\java\com\android\server\am)
ActivityManagerService.main()
   //ActivityManagerService.java 7375p (frameworks\base\services\java\com\android\server\am)
   m.startRunning(null, null, null, null);
      //ActivityManagerService.java 7421p (frameworks\base\services\java\com\android\server\am)
      systemReady();

ActivityManagerService.java 3136p (frameworks\base\services\java\com\android\server\am)
   startActivity(IApplicationThread caller,Intent intent,...)
      int startActivityLocked(caller, intent,...)       //3184L  定义：2691L
         void startActivityLocked()            //3132L  定义：2445L
            resumeTopActivityLocked(null); //2562p  定义：2176L
            if(next=NULL)
            {
               intent.addCategory(Intent.CATEGORY_HOME);
               startActivityLocked(null, intent, null, null, 0, aInfo,null, null, 0, 0, 0, false, false);
            }
            else
            {
               startSpecificActivityLocked(next, true, false); //2439L  定义：1628L
                  realStartActivityLocked() //1640L       定义：1524L
                  //1651L  定义：1654L
                  startProcessLocked(r.processName, r.info.applicationInfo, true, 0,"activity",
r.intent.getComponent());
                     //1717L  定义：1721L
                     startProcessLocked(app, hostingType, hostingNameStr);
                        //1768L    定义：Process.java 222L(frameworks\base\core\java\android\os)
                        int pid = Process.start("android.app.ActivityThread",...)
                           startViaZygote(processClass, niceName, uid, gid, gids,debugFlags, zygoteArgs);

                              pid = zygoteSendArgsAndGetPid(argsForZygote);
                                 sZygoteWriter.write(Integer.toString(args.size()));
            }

   runSelectLoopMode();
      done = peers.get(index).runOnce();
         forkAndSpecialize()      //Zygote.java (dalvik\libcore\dalvik\src\main\java\dalvik\system)
            Dalvik_dalvik_system_Zygote_forkAndSpecialize() //dalvik_system_Zygote.c
(dalvik\vm\native)
               forkAndSpecializeCommon()
                  setSignalHandler()
               RETURN_INT(pid);
```

ActivityThread main()
ActivityThread attach() //ActivityThread.java 3870p
(frameworks\base\core\java\android\app)
mgr.attachApplication(mAppThread)
//ActivityManagerService.java 4677p
(frameworks\base\services\java\com\android\server\am)
attachApplication()
//ActivityManagerService.java 4677p
(frameworks\base\services\java\com\android\server\am)
attachApplicationLocked()
if (realStartActivityLocked(hr, app, true, true)) //ActivityManagerService.java
4609p

//(frameworks\base\services\java\com\android\server\am)
realStartActivityLocked()
//ActivityManagerService.java
(frameworks\base\services\java\com\android\server\am)
app.thread.scheduleLaunchActivity(new Intent(r.intent), r,r.info, r.icicle, results,
newIntents, \
!andResume,isNextTransitionForward());
scheduleLaunchActivity()
queueOrSendMessage(H.LAUNCH_ACTIVITY, r);
ActivityThread.H.handleMessage()
handleLaunchActivity() //ActivityThread.java
(frameworks\base\core\java\android\app)
performLaunchActivity() //ActivityThread.java
(frameworks\base\core\java\android\app)
activity = mInstrumentation.newActivity(cl,
component.getClassName(), r.intent);

//

init 守护进程：
//andriod init 函数启动过程分析：
在 main 循环中会重复调用
drain_action_queue();
restart_processes();
static void restart_processes()
{
process_needs_restart = 0;
service_for_each_flags(SVC_RESTARTING,
restart_service_if_needed);
}
通过循环检测服务列表 service_list 中每个服务的 svc->flags 标记，如果为
SVC_RESTARTING，
那么在满足条件的情况下调用：restart_service_if_needed
通过 service_start 来再次启动该服务。

ActivityManagerService.main

I/SystemServer(45): Starting Power Manager.
I/ServiceManager(26): service 'SurfaceFlinger' died
D/Zygote (30): Process 45 terminated by signal (11)
I/Zygote (30): Exit zygote because system server (45) has terminated
通过错误信息发现程序在调用 SurfaceFlinger 服务的时候被中止。
Service_manager.c (frameworks\base\cmds\servicemanager):
LOGI("service '%s' died\n", str8(si->name));
Binder.c (frameworks\base\cmds\servicemanager):
death->func(bs, death->ptr);

Binder.c (kernel\drivers\misc)中的函数
binder_thread_read()
struct binder_work *w;
switch (w->type)
为 BINDER_WORK_DEAD_BINDER 的时候
binder_parse()中
当 cmd 为 BR_DEAD_BINDER 的时候
执行 death->func(bs, death->ptr)
因为函数
int do_add_service(struct binder_state *bs,
                   uint16_t *s, unsigned len,
                   void *ptr, unsigned uid)
的 si->death.func = svcinfo_death;
所以 death->func(bs, death->ptr) 实际上执行的是
svcinfo_death()//Service_manager.c (frameworks\base\cmds\servicemanager)

==========

==========

7. linux 下 svn 使用指南
1.1 服务器端配置说明
1.1.3 配置用户和权限
1.1.4 导入工程到仓库中
1.2 客户端操作指南及使用规范
1.2.1 检出工作拷贝
1.2.2 svn update 更新别人做的更改
1.2.2.1 svn update 获取最新版本
1.2.2.2 svn update -r 获取特定的版本
1.2.3 svn st 查看文件状态信息
1.2.4 svn log 查看 log 信息
1.2.5 svn diff 查看文件修改详情

1.2.6 svn list 显示版本库的文件列表
1.2.8 svn add 增加目录或者文件
1.2.9 svn delete 删除目录或者文件
1.2.10 svn revert 取消本地修改
1.2.11 svn commit 提交本地做的更改
1.2.12 文件更新,提交时的冲突处理
1.2.13 打标签
1.2.14 清除缓存的认证信息，重新输入用户名和密码

==========

1.1 服务器端配置说明
1.1.1 ubuntu-8.10 svn 服务器安装
sudo apt-get install subversion
1.1.2 建立版本库（Repository）
运行 Subversion 服务器需要首先要建立一个版本库（Repository），可以看作服务器上存放数据的数据库，在安装了 Subversion 服务器之后，可以直接运行
cd path_to_svn_root 例如：/home/svn
svnadmin create --fs-type=fsfs smartphone
--fs-type 指定仓库类型,可以为 fsfs 或 bdb 如果没有指定默认创建为 fsfs 类型 smartphone 为仓库名称

1.1.3 配置用户和权限
修改 path_to_svn_repos/conf/svnserve.conf 文件，打开下面配置项

#anon-access = read
anon-access = none
auth-access = write
password-db = passwd
authz-db = authz
anon-access 应设置等于 none ,否则没有 log 信息

修改 path_to_svn_repos/conf/passwd 文件，添加用户和密码

[users]
wanghui=wanghui
…
1.1.4 导入工程到仓库中
svn import smartphone/ svn://192.168.2.148/smartphone
1.1.5 运行 svn 服务器
svnserve -d -r path_to_svn_root 例如：/home/svn
1.2 客户端操作指南及使用规范
以我们服务器上 android 源代码为例，介绍 svn 常用操作。
1.2.1 检出工作拷贝
检出工作拷贝到 ~/svn/cupcake-jiangping
使用 svn co url
cd ~/svn
svn co svn://192.168.2.148/smartphone/td0901/trunk/cupcake-jianping cupcake-jianping

1.2.2 svn update 更新别人做的更改
1.2.2.1 svn update 获取最新版本
svn update cupcake-jiangping
或者进入目录更新
cd cupcake-jiangping
svn update
如果负责的应用与系统的关联性不是很大，通常不建议频繁进行更新。
1.2.2.2 svn update -r 获取特定的版本
直接在某目录下执行 svn update 获取当前目录下所有文件的最新版本，如果我们只需要获取某个文件或者目录的特定版本，可以通过-r 和 名称进行指定：
svn update –r 5 cupcake-jianping/packages/apps/Phone/src/com/android/phone/xxxx.java
1.2.3 svn st 查看文件状态信息
M cupcake-jianping/packages/apps/Phone/src/com/android/phone/xxxx.java
? cupcake-jianping/packages/apps/Phone/src/com/android/phone/yyyy.java
M 表明文件已经有修改
? 表明文件没有受版本控制

1.2.4 svn log 查看 log 信息
svn log -r 查看所有版本的 log 信息
svn log -r 5 查看某一版本的 log 信息
svn log -r 5:19 查看某区间一系列版本的 log 信息
如果要查看 log 的详细信息可以加上 –v 选项，如：
svn log –v -r 5
1.2.5 svn diff 查看文件修改详情
显示单个文件或者某目录下所有文件的修改详情
svn diff 有三种不同的用法
1. 检查本地修改
2. 比较工作拷贝与版本库
3. 比较版本库与版本库
不使用任何参数调用时，svn diff 将会比较你的工作文件与缓存在.svn 的“原始”拷贝,如：
svn diff cupcake-jianping/packages/apps/Phone
svn diff cupcake-jianping/packages/apps/Phone/src/com/android/phone/zzzz.java
如果传递一个—revision –r 参数，你的工作拷贝会与指定的版本比较。
svn diff -r 3 cupcake-jianping/packages/apps/Phone
如果通过--revision –r 传递两个通过冒号分开的版本号，这两个版本会进行比较。
svn diff -r 2:3 cupcake-jianping/packages/apps/Phone
如果你在本机没有工作拷贝，还是可以比较版本库的修订版本，只需要在命令行中输入合适的 URL：
svn diff -r 33 svn://192.168.2.148/smartphone/td0901/trunk/cupcake-jianping/packages/apps/Phone
1.2.6 svn list 显示版本库的文件列表
svn list svn://192.168.2.148/smartphone/td0901
design/
hedoc/
pm/
release/
tag/
trunk/
svn list svn://192.168.2.148/smartphone/td0901/trunk
3src/

boot-a1/
cupcake-jianping/
linux-2.6.28-a1/
svn list 类似本机的 ls 命令，它查看的是服务器端的目录结构。

1.2.7 svn info 查看版本库信息
cd ~/svn/cupcake-jianping
svn infos
路径: .
URL: svn://192.168.2.148/smartphone/td0901/trunk/cupcake-jianping
版本库根: svn://192.168.2.148/smartphone
版本库 UUID: 1fac82c5-1665-442c-a8d6-2b3dd850438a
版本: 146
节点种类: 目录
调度: 正常
最后修改的作者: tangligang
最后修改的版本: 145
最后修改的时间: 2009-07-31 15:40:50 +0800 (五, 2009-07-31)

1.2.8 svn add 增加目录或者文件
svn add cupcake-jianping/packages/apps/Phone/src/com/android/phone/xxxx
svn add cupcake-jianping/packages/apps/Phone/src/com/android/phone/yyyy.java

1.2.9 svn delete 删除目录或者文件
svn delete cupcake-jianping/packages/apps/Phone/src/com/android/phone/xxxx
svn delete cupcake-jianping/packages/apps/Phone/src/com/android/phone/yyyy.java
在进行删除操作的时候要非常小心，假设我们要添加一个文件：
cupcake-jianping/packages/apps/Phone/src/com/android/phone/yyyy.java
但是在提交之前我们发现并不需要这个文件，这时候我们经常通过 svn delete 来撤销之前添加的文件：
svn delete cupcake-jianping/packages/apps/Phone/src/com/android/phone/yyyy.java
这样操作的后果往往导致本地的文件 yyyy.java 被误删除掉，所以我们正确的做法是：
svnn delete cupcake-jianping/packages/apps/Phone/src/com/android/phone/yyyy.java –keep-local

1.2.10 svn revert 取消本地修改
1．当你发现对某个文件的所有修改都是错误的，或许你根本不应该修改这个文件，或者是从开头重新修改会更加容易的时候可以用这个命令。
2．通过 svn add 添加了一个项目，如果想取消可以通过该命令。
1.2.11 svn commit 提交本地做的更改
通常只对自己负责的模块进行提交，如果负责电话模块，那么提交命令如下：
svn commit cupcake-jiangping/packages/apps/Phone
在提交之前建议用命令：
svn st cupcake-jiangping/packages/apps/Phone 查看状态
M cupcake-jianping/packages/apps/Phone/src/com/android/phone/xxxx.java
? cupcake-jianping/packages/apps/Phone/src/com/android/phone/yyyy.java
M 表明文件已经有修改
？ 表明文件没有受版本控制
1． 如果有“？”存在，并且该文件或者目录是自己添加并且是工程的一部分，那么在提交之前必须先执行 svn add 操作：svn add cupcake-

jianping/packages/apps/Phone/src/com/android/phone/yyyy.java ;
2． 提交之前也必须解决冲突，否则会提交失败。
3． 提交之前必须写 log

1.2.12 文件更新,提交时的冲突处理
$ svn update
U xxxx
G yyyy
C xxxx.c
1. 更新的时候如果前面的状态为：C 表示有冲突存在。
2. 工作拷贝里做过修改，且服务器版本库在修改前工作拷贝的版本后被提交过其他修改；那么 svn commit 首先会失败并要求 update，此时便会出现版本冲突的情况。
当你 Update 出现了冲突时，Subversion 会产生三个文件
filename.mine : 你更新前的文件，没有冲突标志，只是你最新更改的内容。
Filename.roldrev: 就是你在上次更新之后未作更改的版本。
Filename.rnewrev: 客户端从服务器刚刚收到的版本，这个文件对应版本库的 HEAD 版本。
冲突的文件内容，在冲突的地方将被使用“>>>>”标志出来，用户自己进行合并的取舍。
解决冲突之后，svn resolved path_to_name，Subversion 删除冲突所产生三个文件删除，此时你才可以进行提交。(也可以手动删除此三个文件。)

1.2.13 打标签
svn 的标签是通过 copy 命令完成，但是操作的路径必须是服务器的路径，打标签实际上类似于创建一个到特定版本的链接，如：
svn cp svn://192.168.2.148/smartphone/td0901/trunk/cupcake-jianping \
svn://192.168.2.148/smartphone/td0901/tags/cupcake-1.0.6
如果 svn://192.168.2.148/smartphone/td0901/trunk/cupcake-jianping 的当前版本为 5 ，\
那么 svn://192.168.2.148/smartphone/td0901/tags/cupcake-1.0.6 实际 \
上就是 svn://192.168.2.148/smartphone/td0901/trunk/cupcake-jianping 版本 5 的一个标签。

1.2.14 清除缓存的认证信息，重新输入用户名和密码
一个具有权限控制的 svn 版本库在第一次 checkout 工作拷贝的时候会要求输入用户名和密码:

认证领域: <svn://192.168.2.56:3690> 176512f1-51ee-4947-8c07-88c90ab77ac5
“$USER”的密码:
认证领域: <svn://192.168.2.56:3690> d3216b51-7915-4881-bf30-02e0672c61cd
用户名: xxxxx
“xxxxx”的密码:
这些信息被缓存在 ~/.subversion/auth/svn.simple/ 如果需要更换另一个用户登录，必须先清除缓存的认证信息：
rm ~/.subversion/auth/svn.simple/* -rf

1.3 为规避风险，建议遵守以下规范
1.3.1 提交前审查修改情况， 用命令 svn status 浏览所做的修改，svn diff 检查修改的详细信息
1.3.2 提交时，必须填写注释，注释内容清晰描述本次提交内容，变动信息。
1.3.3 做较大修改时，和项目组其他同事的工作相关时，必须通知对方。
1.3.4 纳入版本控制的项目必须定期提交，至少一周提交一次，避免意外事故导致代码丢失。
1.3.5 每次提交后，必须确认工程可正常运行，即 SVN 里保存的是可以正确运行的代码，否则恢复至稳定版本。

1.3.6 编译过程动态产生的东西不要提交到服务器
1.3.7 每次提交前先更新，这样能在提交前发现是否和别人的冲突

filelist=`find ./ -name "*.conf"`;svn add $filelist; svn commit $filelist
filelist=`find ./ -name "*.conf"`;svn delete $filelist --force --keep-local
==========

==========
8. LFS 相关
7.1 lfs 相关资源
7.2 LFS 问题解答
==========
LFS──Linux from Scratch，就是一种从网上直接下载源码，从头编译 LINUX 的安装方式。它不是发行版，只是一个菜谱，
告诉你到哪里去买菜(下载源码)，怎么把这些生东西(raw code) 作成符合自己口味的菜肴──个性化的 linux，不单单是
个性的桌面。
LFS 有什么优势呢？现在看来，它可以提供最快和最小的 Linux。但是最大的优势就是，安装 LFS 是菜鸟变成高手的捷径。
第一次安装，需要按照 LFS 文档安装，如果在此期间所有文档内容你都认真的阅读，保证你受益匪浅；然后发现很多地方可以
不按照别人的老路操作，这个时候用自己的方式参考第一次安装的经验，再一次建立 linux，完成的时候，你会发现你在 LinuxSir.Org
上已经再也不是菜鸟了。

7.1 lfs 相关资源
官方网站：
http://www.linuxfromscratch.org/
lfs 中文网站
http://lfs.linuxsir.org/main/
Linux From Scratch 版本 6.2
http://lamp.linux.gov.cn/Linux/LFS-6.2/index.html
Linux From Scratch 版本 6.4
http://www.bitctp.org/lfsbook-6.4/index.html
Linux 发行版 LFS 讨论区
http://www.linuxsir.org/bbs/forumdisplay.php?f=58

7.2 LFS 问题解答
构建 LFS 的过程中遇到一些问题，总体来说还算顺利，但是还有一些不明白的地方，这里总结一下：
1./etc/fstab 是否在开机就执行，是被谁调用执行的。
2.为什么系统启动之后就要自动挂载/proc 和/sys，这两个目录有什么作用；devpts 和 tmpfs 有什么作用。
参考章节：文件系统概述

3.关于文件系统：按照我的理解，文件系统是内核提供支持的，可以看作是一种协议，提供一种数据组织方式，每个设备必须有自己的文件系统。
不同文件系统的存储设备的数据组织形式不同。mke2fs -jv /dev/<xxx>默认在<xxx>上面创建 EXT3 的文件系统吗？既然这样的话为什么
我们还需要把<xxx>以 ext3 挂载到一个目录呢？如果不是 的话，又是创建什么文件系统呢？为什么第六章中挂载了虚拟内核文件系统之后才能
进入 chroot 环境呢？
参考章节：文件系统概述

4.虚拟文件系统.作用.什么?
虚拟内核文件系统（Virtual Kernel File Systems），是指那些是由内核产生但并不存在于硬盘上（存在于内存中）的文件系统，他们
被用来与内核进行通信。

5.符号链接 和硬链接的区别是什么？什么是符号链接？什么是硬链接？为什么 liinux 上都使用符号链接，而不是硬链接？linux 上很多地方
使用了链接，是为了组织清晰系统的结构和节省空间吗？
硬连接和软连接的区别, 硬连接和复制的区别?
硬连接记录的是目标的 inode；软连接记录的是目标的 path。
hard link 由于 inode 的缘故，只能在本分区中做 link；soft link 可以做跨分区的 link。硬连接因为记录的是 inode，所以不怕改名，
比如 ln aaa bbb, mv aaa ccc，这时 bbb 仍然可以访问;soft-link 就不行：source 的名字改变后，所有链接到这里的 soft-link
全部变为 broken。事实上，即使所有指向该 inode 的 hard-link 的文件名都变了，每一个仍然都可以访问。我想这是它最大的优点吧。
硬连接和复制的区别：
几个硬连接＝几个名字的同一个房子，这些名字可以相同或不同但地址（i-node）是一样的，所以硬连接被删除只是把相应名字抹去，只有最
后一个名字被抹去你才会找不到房子；而复制是建造一个一模一样的房子，当然地址（i-node）就不同的了。

6.工作平台中由 Glibc 提供的动态连接器与 Binutils 里面的标准连接器有什么区别?
参考章节： 链接器和加载器

7.$LFS/tools 目录的所有者是仅存在于宿主环境中的 lfs 用户。如果保留 $LFS/tools 目录，那么该目录内文件的所有者的 user ID 就
没有对应的账号 ？为什么没有帐户，难道不是 LFS?
查看 /etc/password /etc/group 两个文件 分别记录 用户和组的信息
如果用户名和用户 ID 组名和组 ID 的对应关系分别存在上面两个文件中，那么 ls －ls 的时候就可以查看到用户信息，而不再是 ID 等数字信息

8.系统的环境变量保存在哪个文件?
保存在 tty 中

9。配置参数脚本时[alias1] [alias2 ...]什么时候用到?
别名的意思
alias ls='ls --color=auto'
/etc/skel/.bashrc:81: #alias dir='dir --color=auto'

/etc/skel/.bashrc:82: #alias vdir='vdir --color=auto'
/etc/skel/.bashrc:84: #alias grep='grep --color=auto'
/etc/skel/.bashrc:85: #alias fgrep='fgrep --color=auto'
/etc/skel/.bashrc:86: #alias egrep='egrep --color=auto'
/etc/skel/.bashrc:89:# some more ls aliases
/etc/skel/.bashrc:90:#alias ll='ls -l'
/etc/skel/.bashrc:91:#alias la='ls -A'
/etc/skel/.bashrc:92:#alias l='ls -CF'
alias mohuifu='ls -l'

==========

9. linux 内核的初步理解
4. 编译内核
此处内核编译主要针对驱动组之外的同事
1> 设置工具链
内核的 linux-2.6.28-a1/Makefile 中设定了：
CROSS_COMPILE ?= arm-linux-
所以设置 PATH 环境变量，保证能找到正确的工具链
假设工具链位于： /usr/local/marvell-arm-linux-4.1.1/ 设置为：
export PATH:=/usr/local/marvell-arm-linux-4.1.1/bin/:$PATH

2> 更改编译选项（网络启动或者本机启动）
内核顶层目录执行：
make menuconfig
General setup --->
  [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
    () Initramfs source file(s) (NEW)
如果需要支持网络启动反选 [] Initial RAM filesystem and RAM disk (initramfs/initrd) support
如果需要支持本地启动选中 [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
设置 () Initramfs source file(s) (NEW) 为 root
拷贝 cupcake 编译结果 out/target/product/littleton/root/ 到内核顶层目录

3> 编译
内核顶层目录执行 make zImage
编译好的内核：
arch/arm/boot/zImage

initramfs 与 initrd
1. initrd 是一个单独的文件；initramfs 和 Linux 内核链接在一起(/usr 目录下的程序负责生成 initramfs 文档)。
2. initrd 是一个压缩的文件系统映像(可以是 ext2 等，需要内核的驱动)；initramfs 是类似 tar 的 cpio 压缩文档。
内核中的 cpio 解压缩代码很小，而且 init 数据在 boot 后可以丢弃。
3. initrd 运行的程序(initd，不是 init)进行部分 setup 后返回内核；initramfs 执行的 init 程序不返回内核
(如果/init 需要向内核传递控制权，可以再次安装在/目录下一个新的 root 设备并且启动一个新的 init 程序)。

编译脚本及系统变量

initramfs 与 initrd 的区别
1. initrd 是一个单独的文件；initramfs 和 Linux 内核链接在一起(/usr 目录下的程序负责生成 initramfs 文档)。
2. initrd 是一个压缩的文件系统映像(可以是 ext2 等，需要内核的驱动)；initramfs 是类似 tar 的 cpio 压缩文档。
内核中的 cpio 解压缩代码很小，而且 init 数据在 boot 后可以丢弃。
3. initrd 运行的程序(initd，不是 init)进行部分 setup 后返回内核；initramfs 执行的 init 程序不返回内核
(如果/init 需要向内核传递控制权，可以再次安装在/目录下一个新的 root 设备并且启动一个新的 init 程序)。
4. 切换到另一个 root 设备时，initrd 执行 pivot_root 后，卸载 ramdisk；initramfs 是 rootfs，既不能
pivot_root，也不能卸载。initramfs 会删掉 rootfs 的所有内容(find -xdev / -exec rm '{}' ';')，再次安装 root 到 rootfs(cd /newmount; mount --move . /; chroot .)，把 stdin/sdout/stderr 挂在新的/dev/console 上，重新执行 init。由于这是一个相当困难的实现过程(包括在使用一个命令之前把它删除)，所以
klibc 工具包引入一个帮助程序/utils/run_init.c 来执行上述过程。其他大部分工具包(包括 busybox) 把这个命令
称为"switch_root"。