Table of Contents

power_forecast

配置环境

- gcc
- make
- gcc android source code

gcc 4.8.2

编译gcc需要安装的库

```
yum install gmp gmp-devel
yum install mpfr-devel mpfr
yum install libmpc libmpc-devel
```

steps

```
download gcc4.8.2.tar.gz
tar -zxvf gcc4.8.2.tar.gz
cd gcc4.8.2/
./configure --prefix=/usr/share/gcc4.8.2/
make
sudo make install
```

qa

/usr/include/gnu/stubs.h:7:27: error: gnu/stubs-32.h: 找不到文件

> yum install yum install install glibc-devel.i686

config multi gcc

```
alternatives --install /usr/bin/gcc gcc /usr/share/gcc4.8.2/bin/gcc 60
alternatives --config gcc
```

make -v 3.8.2

steps

```
download make3.8.2.tar.gz
tar -zxvf make.tar.gz
./configure --prefix=/usr/share/make3.8.2/
make
make install
```

config multi make

```
sudo alternatives --config make
sudo alternatives --install /usr/bin/make make  /usr/share/make3.8.2/bin/make  60
```

Compile Android source code

编译安卓源代码

```
yum install glibc.i686  bison flex  gpref
source build/envsetup.sh
lunch
make -j4

编译成功后配置环境变量
为了使用模拟器
export PATH=$PATH:android-4.4.4_r1/out/host/linux-x86/bin

为编译kernel
export PATH=$PATH:android-4.4.4_r1/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/bin
```

运行模拟器

```
cd android/
source build/envsetup.sh
lunch
#没有上面两条指令可能无法启动模拟器
emulator -partition-size 1024
或者
emulator -partition-size 1024 -kernel ../goldfish/arch/arm/boot/zImage
```

> android 模拟器启动需要四个文件，分别是zImage ,system.img , userdata.img ramdisk.img source build/envsetup.sh && lunch 1 可以找到后面三个文件，故上述命令只制定了内核zImage,或者可以使用如下命令

//emulator -kernel ./prebuilts/qemu-kernel/arm/kernel-qemu-armv7 -sysdir ./out/target/product/generic/ -system system.img -data userdata.img -partition-size 1024

> http://blog.csdn.net/flydream0/article/details/7070392

Problem

```
frameworks/base/api/current.txt:8: error 9: Removed public constructor Manifest.permissio
n()
The errors were resolved by re-downloading the source code.So if any of you encountered t
he same error, I would suggest you to re-download the source.The error should be gone.
```

compile goldfish

```
cd goldfish
make goldfish_armv7_defconfig 默然编译选项,执行后会生成.config文件
更多现有的config文件在arch/arm/configs
make -j4
运行emulator 使用编译好的内核goldfish/arch/arm/boot/zImage
可以运行make menuconfig 配置编译选项
```

enable kprobe

- make menuconfig
- enable loading modle
- in general setup enable kprobe
- make -j4

insufficient permissions for device

设置usb权限

```
$ lsusb
Bus 001 Device 002: ID 8087:0020 Intel Corp. Integrated Rate Matching Hub
Bus 002 Device 002: ID 8087:0020 Intel Corp. Integrated Rate Matching Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 003: ID 04b3:310c IBM Corp. Wheel Mouse
Bus 002 Device 004: ID 24ae:2000
Bus 002 Device 006: ID 2a45:0c02 Meizu Corp. MX Phone (MTP & ADB)
```

```
vim /etc/udev/rules.d/70-android.rules
加入下面的内容
SUBSYSTEM=="usb",ATTRS{idVendor}=="2a45",ATTRS{idProduct}=="0c02",MODE="0666"
```

> get idVendor and idProduct from output of lsusb

重启udev

```
ubuntu: service udev restart
centos: systemctl restart systemd-udev-trigger.service
```

重启adb server

```
adb kill-server
adb shell
```

内核模块

内核模块

hello world

cd goldfish/drivers mkdir hello

./hello/hello.c

touch ./hello/hello.c

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
static int __init hello_init(void)  {
    printk(KERN_ERR "Hello world init\n");
    return 0;
}

static void __exit hello_exit(void)  {
    printk(KERN_ERR "Hello world exit\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

./hello/Makefile

touch ./hello/Makefile

```
#可选模块
obj-$(CONFIG_HELLO) += hello.o

#内建模块
#obj-y := hello.o
```

./hello/Kconfig

touch ./hello/Kconfig

```
config HELLO
    tristate "Fake Register Driver"
    default n
    help
    This is the freg driver for android system.
```

./Kconfig

```
source "drivers/hello/Kconfig"
```

./Makefile

```
obj-$(CONFIG_HELLO)+=hello/
```

make menuconfig 的时候把该项选上,然后make

也可以在hello/Makefile中 obj-y := hello.o,此时hello/Kconfig可以为空, .Makefile中obj-y+=hello/ .Kconfig为空,这种配置方法默认是编译进内核的

动态加载

mkdir hello touch hello/hello.c

hello/hello.c

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
static int __init hello_init(void)  {
    printk(KERN_ERR "Hello world init\n");
    return 0;
}

static void __exit hello_exit(void)  {
    printk(KERN_ERR "Hello world exit\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

hello/Makefile

touch hello/Makefile

```
obj-m := hello.o

#内核在哪
KID := /home/cwd/fedora/cwd/paper/goldfish
PWD := $(shell pwd)
ARCH =arm
CROSS_COMPILE=arm-eabi-
cc=$(CROSS_COMPILE)gcc
LD=$(CROSS_COMPILE)ld

all:
    make -C $(KID) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) modules

clean:
    rm -rf *.o .cmd *.ko *.mod.c .tmp_versions
```

> KID表示内核源码目录，这种方式适用于嵌入式开发的交叉编译，KID目录中包含了内核驱动模块所需要的各种头文件及依赖。若在PC机开发内核模块则应使用
>
> ```
> #KERN_DIR = /usr/src/$(shell uname -r)
> #KERN_DIR = /lib/modules/$(shell uname -r)/build
> ```
>
> -C表示 指定进入指定的目录即KID，是内核源代码目录，调用该目录顶层下的Makefile,目标为modules。M=$(shell pwd)选项让该Makefile在构造modules目标之前返回到模块源代码目录并在当前目录生成obj-m指定的hello.o目标模块。
>
> ### make

adb push hello.ko /data/

insmod hello.ko

cat /proc/kmsg

rmmod hello.ko

读写文件

fiel_operation.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
static char buf[] = "hello";
static char buf1[10];

int __init hello_init(void)
{
    struct file *fp;
    mm_segment_t fs;
    loff_t pos;
    printk("hello enter\n");
    fp = filp_open("test_file", O_RDWR | O_CREAT, 0644);
    if (IS_ERR(fp)) {
        printk("create file error\n");
        return -1;
    }
    fs = get_fs();
    set_fs(KERNEL_DS);
    pos = 0;
    vfs_write(fp, buf, sizeof(buf), &pos);
    pos = 0;
    vfs_read(fp, buf1, sizeof(buf), &pos);
    printk("read: %s\n", buf1);
    filp_close(fp, NULL);
    set_fs(fs);
    return 0;
}
void __exit hello_exit(void)
{
    printk("hello exit\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
```

Makefile

```
obj-m := file_operation.o

KID := /home/cwd/fedora/cwd/paper/goldfish
PWD := $(shell pwd)
ARCH =arm
CROSS_COMPILE=arm-eabi-
cc=$(CROSS_COMPILE)gcc
LD=$(CROSS_COMPILE)ld


all:
    make -C $(KID) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) modules

clean:
    rm -rf *.o .cmd *.ko *.mod.c .tmp_versions
```

make && adb push && insmod

jprobe

jprobe_example.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>


static long jdo_fork(unsigned long clone_flags, unsigned long stack_start,
          struct pt_regs *regs, unsigned long stack_size,
          int __user *parent_tidptr, int __user *child_tidptr)
{
    printk(KERN_INFO "jprobe: clone_flags = 0x%lx, stack_size = 0x%lx,"
           " regs = 0x%p\n",
           clone_flags, stack_size, regs);

    /* Always end with a call to jprobe_return(). */
    jprobe_return();
    return 0;
}

static struct jprobe my_jprobe = {
    .entry          = jdo_fork,
    .kp = {
        .symbol_name    = "do_fork",
    },
};

static int __init jprobe_init(void)
{
    int ret;

    ret = register_jprobe(&my_jprobe);
    if (ret < 0) {
        printk(KERN_INFO "register_jprobe failed, returned %d\n", ret);
        return -1;
    }
    printk(KERN_INFO "Planted jprobe at %p, handler addr %p\n",
           my_jprobe.kp.addr, my_jprobe.entry);
    return 0;
}

static void __exit jprobe_exit(void)
{
    unregister_jprobe(&my_jprobe);
    printk(KERN_INFO "jprobe at %p unregistered\n", my_jprobe.kp.addr);
}

module_init(jprobe_init)
module_exit(jprobe_exit)
MODULE_LICENSE("GPL");
```

Makefile

```
obj-m := jprobe_example.o


KID := /home/cwd/fedora/cwd/paper/goldfish
PWD := $(shell pwd)
ARCH =arm
CROSS_COMPILE=arm-eabi-
cc=$(CROSS_COMPILE)gcc
LD=$(CROSS_COMPILE)ld


all:
    make -C $(KID) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) modules

clean:
    rm -rf *.o .cmd *.ko *.mod.c .tmp_versions
```

adb push insmod cat /proc/kmsg

more note

```
/home/cwd/fedora/cwd/note
```

为Nexus5编译Android固件

初始化编译环境

```
cd android4.4.4
. build/envsetup.sh
```

加载机型

lunch 选择 aosp_hammerhead-userdebug

device	code name	build configuration
Nexus 6	shamu	aosp_shamu-userdebug
Nexus Player	fugu	aosp_fugu-userdebug
Nexus 9	volantis (flounder)	aosp_flounder-userdebug
Nexus 5 (GSM/LTE)	hammerhead	aosp_hammerhead-userdebug
Nexus 7 (Wi-Fi)	razor (flo)	aosp_flo-userdebug
Nexus 7 (Mobile)	razorg (deb)	aosp_deb-userdebug
Nexus 10	mantaray (manta)	full_manta-userdebug
Nexus 4	occam (mako)	full_mako-userdebug
Nexus 7 (Wi-Fi)	nakasi (grouper)	full_grouper-userdebug
Nexus 7 (Mobile)	nakasig (tilapia)	full_tilapia-userdebug
Galaxy Nexus (GSM/HSPA+)	yakju (maguro)	full_maguro-userdebug
Galaxy Nexus (Verizon)	mysid (toro)	aosp_toro-userdebug
Galaxy Nexus (Experimental)	mysidspr (toroplus)	aosp_toroplus-userdebug
PandaBoard (Archived)	panda	aosp_panda-userdebug
Motorola Xoom (U.S. Wi-Fi)	wingray	full_wingray-userdebug
Nexus S	soju (crespo)	full_crespo-userdebug
Nexus S 4G	sojus (crespo4g)	full_crespo4g-userdebug3.2

生成驱动目录

需要安装驱动，否则一直停留在google的黑色界面

HARDWARE COMPONENT	COMPANY	DOWNLOAD	MD5 CHECKSUM	
NFC, Bluetooth, Wi-Fi	Broadcom	下载	2c398994e37093df51b105d63f0eb611	991346159c9
Camera, Sensors, Audio	LG	下载	74cf8235e6bb04da28b2ff738b13eee9	175dd5bae81
Graphics, GSM, Camera, GPS, Sensors, Media, DSP, USB	Qualcomm	下载	0a43395e175d3de3dc312d8abdcb4f20	007cf9d49f0

> 下载完成后,解压出来是三个**.sh**文件,放到**Android**源码目录下面,然后执行.会将相关驱动放到**vender**目录下面.
>
> 针对不同的**android**版本有不同的驱动文件,参考下面的下载官网

下载官网

执行编译命令

make -j4

刷机命令

> 刷机的时候会默认刷入一个内核镜像，名字是boot,可以从fastboot -w flashall中的输出看到 Nexus5关机状态下,长按音量下+电源,即可进入recovery模式, 然后在源码根目录下执行下面命令:
>
> ```
> source  build/envsetup.sh
> lunch
> fastboot -w flashall
> ```
>
> 或者
>
> ```
> **root权限**
> adb shell
> reboot bootloader
> source  build/envsetup.sh
> lunch
> fastboot -w flashall
> ```

adb fastboot这些命令在编译android源代码后加入了环境变量

problem

如果遇到开机一听停留在Google的显示界面，那么则没有安装驱动，安装相应的驱动就可以成功开机，驱动的安装方法参考上文，或者文章结尾的参考链接

参考

msm

android对应虚拟上面的内核是goldfish，适用于N5等一些列手机的内核是msm,msm中有很多分支. N5的code name 是hammerhead，在msm分支中适用于N5 android4.4 的分支是android-msm-hammerhead-3.4-kitkat-mr1，kitkat是android4.4的 code name

download msm

git clone https://android.googlesource.com/kernel/msm.git msm cd msm git checkout android-msm-hammerhead-3.4-kitkat-mr1

compile msm

```
cd android4.4.4
source build/envsetup.sh
lunch
```

> 上面source lunch貌似不是必须的 修改内核Makefile文件
>
> ```
> #ARCH           ?= $(SUBARCH)
> #CROSS_COMPILE  ?= $(CONFIG_CROSS_COMPILE:"%"=%)
> ARCH            ?= arm
> CROSS_COMPILE   ?= arm-eabi-
> ```
>
> ```
> make menuconfig 或者make hammerhead_defconfig
> make -j4
> ```

成功之后会生成zImage-dtb

生成boot.img

```
mkbooting --kernel zImage-dtb --ramdisk  android4.4.4/out/target/product/hammerhead/ramdi
sk.img --cmdline "console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31
 maxcpus=2 msm_watchdog_v2.enable=1" --base 0x00000000 --pagesize 2048 --ramdisk_offset 0
x02900000 --tags_offset 0x02700000 --output my_boot.img
```

关于上述参数请参考这里

刷机

```
adb shell
reboot bootloader

主机需要root权限,否则会显示wait for device
fastboot boot my_boot.img

如果上面运行无误后,则可以执行fastboot flash boot my_boot.img
fastboot reboot

**fastboot boot my_boot.img 只是测试my_boot.img是否可用，不会写入,如果用fastboot flash boot bo
ot my_boot.img则会写入**
```

加载jprobe_example.ko 模块

```
adb push jprobe_example.ko /sdcard/
adb shell
su
insmod jprobe_example.ko
lsmod
cat /proc/kmsg
```

reference:

- 1
- 2 文章mkbooting --kernel有误

jprobe

Cpufreq

CPU 电源状态（C State）和 CPU/设备性能状态（P State）

在开始 CPUfreq 讨论之前，我们先来看看 CPU 电源状态和 CPU/设备性能状态。

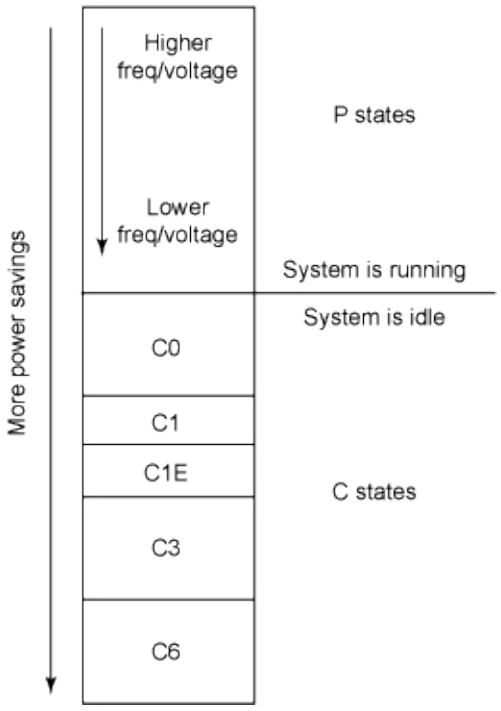
CPU 电源状态：几乎全是空闲

CPU 电源状态（不包括处理器运行时的 C0 状态）是空闲状态，处理器将解锁并关闭组件来节省电能。CPU 电源状态程度越深，采取的电能节省措施就越多 — 比如说停止处理器时钟或停止外部中断请求。这些状态帮助空闲中的系统节省电能。

此外，C1E 模式（或称作 Enhanced C1 或 C1 Enhanced Mode）也可以帮助空闲系统节省电能。同样通过降低电压和频率，C1E 尝试比传统 C1 状态（只会停止时钟信号）提供更大的电能节省。事实上，C1E 能够比任何 CPUfreq 调控器更快地降低电压/频率。

并非所有处理器都有这些选项，但是要使用 C 电源状态和 CIE，请确保启用了 BIOS 选项 CPU C State 和 C1E（或者类似的选项），以便于在空闲时实现更大的电能节省。一些系统支持 C3 甚至 C6 尝试休眠状态。

记住，CPU 电源状态程度越深，节省的电能就越多。



参考 Documentf翻译

Problem

Problem

unload module

delete_module failed (errno 38)

这个问题主要是没有配置模块卸载 解决办法如下：

```
make menuconfig
enter enable loadable module support and select module unloading
```

禁用cpu core

固定cpu频率

> If you have selected the "userspace" governor which allows you toset the CPU operating frequency to a specific value, you can read out the current frequency in scaling_setspeed. By "echoing" a new frequency into this you can change the speed of the CPU, but only within the limits of scaling_min_freq and scaling_max_freq.

```
cd /sys/devices/system/cpu/cpu0/cpufreq/cpu0/
cat scaling_available_governors
cat scaling_available_frequencies
echo userspace scaling_governor
echo xxxx > scaling_cur_freq
```

android与linux内核关系

- 出处

Android Version |API Level |Linux Kernel in AOSP |----------|----------|----------| 1.5 Cupcake |3 |2.6.27 1.6 Donut |4 |2.6.29 2.0/1 Eclair |5-7 |2.6.29 2.2.x Froyo |8 |2.6.32 2.3.x Gingerbread |9, 10 |2.6.35 3.x.x Honeycomb |11-13 |2.6.36 4.0.x Ice Cream San|14, 15 |3.0.1 4.1.x Jelly Bean |16 |3.0.31 4.2.x Jelly Bean |17 |3.4.0 4.3 Jelly Bean |18 |3.4.39 4.4 Kit Kat |19, 20 |3.10 5.x Lollipop |21, 22 |3.16.1 6.0 Marshmallow |23 |3.18.10

android 环境

更新**android sdk**

在android_sdk中的tools文件夹下android这个文件，./android会出现sdk manager 窗口,可以设置代理来更新。

获取源代码

download repo tool

```
mkdir ~/bin
PATH=~/bin:$PATH
curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+x ~/bin/repo
```

使用每月更新的初始化包

> 由于首次同步需要下载 24GB 数据，过程中任何网络故障都可能造成同步失败，我们强烈建议您使用初始化包进行初始化。 下载 https://mirrors.tuna.tsinghua.edu.cn/aosp-monthly/aosp-latest.tar，下载完成后记得根据 checksum.txt 的内容校验一下。 由于所有代码都是从隐藏的 .repo 目录中 checkout 出来的，所以我们只保留了 .repo 目录，下载后解压 再 repo sync 一遍即可得 到完整的目录。 使用方法如下:
>
> ```
> wget https://mirrors.tuna.tsinghua.edu.cn/aosp-monthly/aosp-latest.tar # 下载初始化包
> tar xf aosp-latest.tar
> cd AOSP   # 解压得到的 AOSP 工程目录
> # 这时 ls 的话什么也看不到，因为只有一个隐藏的 .repo 目录
> repo sync # 正常同步一遍即可得到完整目录
> # 或 repo sync -l 仅checkout代码
> ```
>
> 此后，每次只需运行 repo sync 即可保持同步。 我们强烈建议您保持每天同步，并尽量选择凌晨等低峰时间

checkout branch

> repo help init
>
> ```
> repo init -b xxxversion
> repo sync
> ```
>
> android版本列表

获取powerfile.xml

获取的方法

> download framework-res.apk, 然后使用apktool工具反编译 该文件位于手机 /system/framework/framework-res.apk

```
adb pull /system/framework/framework-res.apk

apktool d framework-res.apk
```

install apktool

app

重要类的初始化

BatteryStatsHelper

```
//BatteryStatsHelper mStatsHelper =  new BatteryStatsHelper(activity, true);
BatteryStatsHelper mStatsHelper =  new BatteryStatsHelper(this);
```

PowerProfile

```
PowerProfile powerProfile = mStatsHelper.getPowerProfile();
```

BatteryStats

```
BatteryStats stats = mStatsHelper.getStats();
```

BatterySipper

```
final List<BatterySipper> usageList = getCoalescedUsageList(USE_FAKE_DATA ? getFakeStats(
) : mStatsHelper.getUsageList());
final int numSippers = usageList.size();
for (int i = 0; i < numSippers; i++)
{
    final BatterySipper sipper = usageList.get(i);
}
```

```
BatterySipper sipper = new BatterySipper(DrainType.APP,new FakeUid(UserHandle.getSharedAp
pGid(Process.FIRST_APPLICATION_UID)), 10.0f);
sipper.packageWithHighestDrain = "dex2oat";
```

UserHandle

```
UserHandle userHandle = new UserHandle(UserHandle.getUserId(sipper.getUid()));
```

NDK_PROJECT_PATH=null

1. build.gradle文件中添加

```
android {
    ...

    sourceSets.main {
        jni.srcDirs = []
        jniLibs.srcDir 'src/main/libs'
    }

}
```

   i. src/main/jni目录下添加空文件

```
util.c
empty.cpp
```

battery_walkthrough

```
https://github.com/google/battery-historian
> adb kill-server
> adb devices
> adb shell dumpsys batterystats --reset
<disconnect and play with app>...<reconnect>
> adb devices

>adb shell dumpsys batterystats > batterystats.txt

> python historian.py batterystats.txt > batterystats.html
```