<https://android.googlesource.com/kernel/>

# TODO

上一次奔溃日志打印

如何dump内核工具

[高通平台Android源码分析之Linux内核设备树(DT - Device Tree)](http://huaqianlee.github.io/2015/08/19/Android/%E9%AB%98%E9%80%9A%E5%B9%B3%E5%8F%B0Android%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8BLinux%E5%86%85%E6%A0%B8%E8%AE%BE%E5%A4%87%E6%A0%91-DT-Device-Tree-dts%E6%96%87%E4%BB%B6/)

Lowmemorykiller

制定嵌入式专家计划

Liun,git源码分析和使用修改

硬件条件

<https://elixir.bootlin.com/linux/latest/source/drivers/usb/core/usb.c>

<https://www.jianshu.com/p/36b488863bc0>

可以先看一下android的智能指针和linux进程间通信（只看posix，不要看systemV）

UNIX网络编程卷2：进程间通信（第2版）史蒂文斯

<http://ybin.cc/tools/clion-for-linux-driver-developer/>

<https://blog.csdn.net/godbreak/article/details/81225393>

如何阅读Linux源码(转)

<https://blog.csdn.net/qq_37468954/article/details/77483013>

集群服务器解决方案

https://mp.weixin.qq.com/s/0qj\_ukCPS1aCU\_aA1D-Ihw

<https://mp.weixin.qq.com/s/eGYzXAwI6VdkDkFgs_E-rg>

# 概述

## 嵌入式架构工程师

<https://zhuanlan.zhihu.com/p/27958152>

如何用一个实例来探讨嵌入式软件架构设计

<https://blog.csdn.net/dwx1005526886/article/details/80877338>

**嵌入式应用软件架构设计POS**

<https://blog.csdn.net/yyz_1987/article/details/77008252>

笔者从事嵌入式软件开发有6，7个年头，bsp，驱动，应用软件，[Android](https://link.zhihu.com/?target=http%3A//lib.csdn.net/base/android" \t "_blank) hall，framework等都有涉猎。平时除了关注嵌入式行业的发展，也多少对Web，后台服务端，分布式等方向的技术有一些关注。

目前国内的[嵌入式开发](https://link.zhihu.com/?target=http%3A//lib.csdn.net/base/embeddeddevelopment)主要分为嵌入式底层开发和嵌入式应用开发，嵌入式的底层开发一般叫做驱动开发，或者bsp开发，有时也有称之为[linux](https://link.zhihu.com/?target=http%3A//lib.csdn.net/base/linux" \t "_blank)内核开发，名字听着都很高大上的感觉。

这么高大上的名字为什么没有架构师呢：linux kernel的架构师是linus等一众linux kernel开发维护者，因为本身linux kernel或者[操作系统](https://link.zhihu.com/?target=http%3A//lib.csdn.net/base/operatingsystem" \t "_blank)就是一个通用的平台，解决通用的问题，linux开源届的大牛都已经制定好了架构规则

那嵌入式bsp开发人员都在做什么：

，就是解些稳定性的bug了（这里对具体工作不详细描述了，调试外设只会增加一些经验，增加广度，对提高深度贡献不大，只是按不会调试-》会调试-》调试的快这个路线发展，而解稳定性问题确实是需要一些积累经验）

而嵌入式上的应用开发，一般业务逻辑比较简单，被很多人忽略

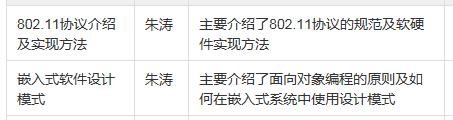
至此感觉嵌入式行业的确不需要架构师，被互联网行业的鄙视也没什么大惊小怪的。

还是不要“妄想”做Linux kernel的架构师了

其实给客户做嵌入式Linux应用层解决方案和SDK的公司，就需要架构师了，但是项目一旦bring up起来，就成型了，不会轻易改架构

## MCU工程师





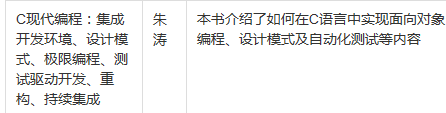
## LINUx专题





## 学习资料







## kernel 内核学习路线资料

### 计算机基础

1, 关于操作系统理论的最初级的知识。不需要通读并理解《操作系统概念》《现代操作系统》等巨著，但总要知道分时（time-shared）和实时（real-time）的区别是什么，进程是个什么东西，CPU和系统总线、内存的关系（很粗略即可），等等。

2, 关于C语言。不需要已经很精通C语言，只要能熟练编写C程序，能看懂链表、散列表等数据结构的C实现，用过gcc编译器，就可以了。当然，如果已经精通C语言显然是大占便宜的。

3, 关于CPU的知识。这块儿可以在学习内核过程中补，但这样的话你就需要看讲解很详细的书，比方后面将会提到的《情景分析》。你是否熟悉 Intel 80386 CPU？尝试着回答这几个问题来判断一下：1）说出80386的中断门和陷阱门的区别；2）说出保护模式与实模式的区别；3）多处理器机器上，普通的读-改-写回一块内存这样的动作，为什么需要特殊的手段来保护。等等。讲解基于其它CPU的Linux内核的书，目前好象只有一本《IA64Linux内核：设计与实现》──也还是Intel的，其它都是讲解基于IA32的。

### **了解Linux内核机制**

概要设计文档：Linux Kernel Developmen/Linux内核设计与实现(原书第3版)2011-05

拉芙 (Robert Love)、 陈莉君

此书是当今首屈一指的入门最佳图书。作者是为2.6内核加入了抢占的人，对调度部分非常精通，而调度是整个系统的核心，因此本书是很权威的。这本书讲解浅显易懂，全书没有列举一条汇编语句，但是给出了整个Linux操作系统2.6内核的概观，使你能通过阅读迅速获得一个overview。而且对内核中较为混乱的部分（如下半部），它的讲解是最透彻的。对没怎么深入内核的人来说，这是强烈推荐的一本书。

### **研究Linux内核源码**

详细设计：Understanding the Linux Kernel 英文原版/深入理解LINUX内核(第3版)(涵盖2.6版) [Daniel](http://search.dangdang.com/?key2=Daniel&medium=01&category_path=01.00.00.00.00.00) P. [Bovet](http://search.dangdang.com/?key2=Bovet&medium=01&category_path=01.00.00.00.00.00" \t "_blank),[MarcoCesati](http://search.dangdang.com/?key2=MarcoCesati&medium=01&category_path=01.00.00.00.00.00)

《Linux内核源代码情景分析》上、下。毛德操、胡希明著，浙江大学出版社，上册￥80,下册￥70. 评说：

本书是基于2.4.0内核的，比较早，也没听说会出第二版。上册讲解内存管理、中断、异常与系统调用、进程控制、文件系统与传统Unix IPC；下册讲解socket、设备驱动、SMP和引导。关于这套书的评价褒贬不一，我个人认为其深度是同类著作中最优秀的。本书基于Intel IA32体系，由于厚度大，很多体系上的知识都捎带讲解了，所以如果你想深入了解内核的工作机制而又不非常熟悉Intel CPU的体系构造，本书是最合适的。缺点是：版本较老，没有TCP/IP协议栈部分（它讲的socket只是Unix域协议的），图表太少，不适合初学者入门。还有就是对学生朋友来说，可能书价偏高，这样的话可以考虑先买上册，因为上册是核心部分，下册一大部分都在讲具体PCI/ISA/USB设备的驱动

### **设备驱动方向学习**

LINUX设备驱动程序(第3版)2010-09：[Jonathan](http://search.dangdang.com/?key2=Jonathan&medium=01&category_path=01.00.00.00.00.00" \t "_blank) [Corbet](http://search.dangdang.com/?key2=Corbet&medium=01&category_path=01.00.00.00.00.00)

### **嵌入式开发**

现在Linux相关的工作，多集中在一些嵌入式开发领域，arm，mips等，要学习以下这些体系架构的的资料，了解CPU的设计和工作方式。 ARM就看对应的芯片手册，讲的很细致。MIPS就看 《see mips run》，有一二两版，两版内容有些差异，推荐都看。

### **网络方向学习**

学些Linux网络部分就学习《深入理解LINUX网络技术内幕》。这本书把Linux的网络部分讲的非常清晰透彻。但是通常不做这方面的工作研究，也不用研究这么深，毕竟现在国内相关职位较

LINUX内核源代码情景分析(上册)2001-09毛德操、 胡希明

### 补充一点经验

不要认为Linux很庞大，很复杂，就觉的很难学。任何东西认真学下来都是能学会的，看你的恒心和毅力了。另外，不要走弯路，不要看 市面上讲什么Linux0.11的那些书，直接学你要学的东西。就像学C语言看什么谭浩强一样，弯路走了，力气没少花，还严重影响学习效果。

### 参考

<https://blog.csdn.net/im_Louis_LIU/article/details/51809683>

## Ubuntu

Ubuntu 是 debian 的后代，debian 是非常好的操作系统，Ubuntu 跟它的区别就是更加的人性化，省去了一些比较 hack 的部分。Ubuntu 的着重发力点是 桌面操作系统，它确实也是做得非常不错和被广泛使用的一个 Linux 桌面版。由于 debian 系跟 redhat 一直都是 linux 发型版的 top2 厂商，Ubuntu 本身又有了不少的积累，所有 Ubuntu 的服务器版也是非常好用，甚至比 CentOS 更推荐使用，这个版本的名称是 Ubuntu Server。

在 Ubuntu 的发型版本号里，有一类是带 lts，例如最新的 Ubuntu Lts 18.04 这个代号说明了此版本是长期支持的版本，有几年的支持时间，是更加稳定的版本。

AP Application Processor）：应用进程

CP: 通讯进程

# 源码了目录结构

arch 这个子目录包含了此核心源代码所支持的硬件体系结构相关的核心代码。如对于X86平台就是i386。

  include 这个目录包括了核心的大多数include文件。另外对于每种支持的体系结构分别有一个子目录。

  init 此目录包含核心启动代码。

  mm 此目录包含了所有的内存管理代码。与具体硬件体系结构相关的内存管理代码位于arch/\*/mm目录下，如对应于X86的就是arch/i386/mm/fault.c 。

  drivers 系统中所有的设备驱动都位于此目录中。它又进一步划分成几类设备驱动，每一种也有对应的子目录，如声卡的驱动对应于drivers/sound。

  ipc 此目录包含了核心的进程间通讯代码。

  modules 此目录包含已建好可动态加载的模块。

  fs Linux支持的文件系统代码。不同的文件系统有不同的子目录对应，如ext2文件系统对应的就是ext2子目录。

  kernel 主要核心代码。同时与处理器结构相关代码都放在arch/\*/kernel目录下。

  net 核心的网络部分代码。里面的每个子目录对应于网络的一个方面。

  lib 此目录包含了核心的库代码。与处理器结构相关库代码被放在arch/\*/lib/目录下。

  scripts此目录包含用于配置核心的脚本文件。

# 系统目录结构

 Linux下的文件系统为树形结构，入口为/ 树形结构下的文件目录： 无论哪个版本的Linux系统，都有这些目录，这些目录应该是标准的。各个Linux发行版本会存在一些小小的差异，但总体来说，还是大体差不多。

**1. /** 文件系统的入口，最高一级目录；

**2. /bin** 基础系统所需要的命令位于此目录，是最小系统所需要的命令，如：ls, cp, mkdir等。

这个目录中的文件都是可执行的，一般的用户都可以使用。

**3. /boot** 包含Linux内核及系统引导程序所需要的文件，比如 vmlinuz initrd.img 文件都位于这个目录中。在一般情况下，GRUB或LILO系统引导管理器也位于这个目录；

**4. /dev** 设备文件存储目录，比如声卡、磁盘... ... 这是一个非常有趣的目录，是Linux文件系统的一个闪亮的特性 - 所有对象都是文件或目录。仔细观察这个目录你会发现hda1, hda2等, 它们代表系统主硬盘的不同分区。

/dev/cdrom和/dev/fd0代表你的CDROM驱动器和floppy驱动器。看上去可能有些奇怪，但比较文件和硬件的特性这却是十分合理的。它们都可以读出和写入。例如/dev/dsp，这个文件代表你的扬声器。那么写入这个文件的数据都回传送到喇叭。试一试 'cat /etc/lilo.conf > /dev/dsp' 你会听到一些声音。这是你的lilo.conf 文件的声音！同样，向 /dev/ttyS0 ( COM 1 ) 读出或写入数据你可以和接到上面的设备进行通讯。

**5. /etc** 存放系统程序或者一般工具的配置文件。

如安装了apache2之后，配置文件在/etc/apache2/目录下。

/etc/init.d 这个目录是用来存放系统或服务器以System V模式启动的脚本，这在以System V模式启动或初始化的系统中常见。

如apache2的/etc/init.d apache2 start|stop|restart MySQL为/etc/init.d mysql start|stop|restart

**6. /home** 普通用户默认存放目录 Linux 是多用户环境，所以每一个用户都有一个只有自己可以访问的目录（当然管理员也可以访问）。它们以 /home/username 的方式存在。这个目录也保存一些应用对于这个用户的配置，比如 IRC, X 等。

**7. /lib** 库文件存放目录这里包含了系统程序所需要的所有共享库文件，类似于 Windows 的共享库 DLL 文件。

**8. /lost+found** 在ext2或ext3文件系统中，当系统意外崩溃或机器意外关机，而产生一些文件碎片放在这里。当系统启动的过程中fsck工具会检查这里，并修复已经损坏的文件系统。 有时系统发生问题，有很多的文件被移到这个目录中，可能会用手工的方式来修复，或移到文件到原来的位置上。

Linux 应该正确的关机。但有时你的系统也可能崩溃掉或突然断电使系统意外关机。那么启动的时候 fsck 将会进行长时间的文件系统检查。Fsck 会检测并试图恢复所发现的不正确的文件。被恢复的文件会放置在这个目录中。所恢复的文件也许并不完整或并不合理，但毕竟提供了一些恢复数据的机会。

**9. /media**即插即用型存储设备的挂载点自动在这个目录下创建，比如USB盘系统自动挂载后，会在这个目录下产生一个目录 ；CDROM/DVD自动挂载后，也会在这个目录中创建一个目录，类似cdrom 的目录。这个只有在最新的发行套件上才有.

**10. /mnt** /mnt 这个目录一般是用于存放挂载储存设备的挂载目录的，比如有cdrom等目录。有时我们可以把让系统开机自动挂载文件系统，把挂载点放在这里也是可以的。比如光驱可以挂载到/mnt/cdrom 。

这是一个普通的加载目录，在这里你可以加载你的文件系统或设备。

加载是使一个文件系统对于系统可用的过程。

在加载后你的文件可以在加载目录下访问。

这个目录通常包含加载目录或用于加载软驱和光驱的子目录。

如果需要，你也可以在这里建立其它的加载目录。

对于加载目录的位置并没有强制性的要求，你可以在系统的任意位置建立加载目录。

建立 /mnt 只是为了使系统更工整的惯例。

**11. /opt**表示的是可选择的意思，有些软件包也会被安装在这里，也就是自定义软件包，比如在Fedora Core 5.0中，OpenOffice就是安装在这里。有些我们自己编译的软件包，就可以安装在这个目录中；通过源码包安装的软件，可以通过 ./configure --prefix=/opt/，将软件安装到opt目录。

这个目录包含所有默认系统安装之外的软件和添加的包。

**12. /proc** 操作系统运行时，进程（正在运行中的程序）信息及内核信息（比如cpu、硬盘分区、内存信息等）存放在这里。/proc目录是伪装的文件系统proc的挂载目录，proc并不是真正的文件系统。

这是系统中极为特殊的一个目录，实际上任何分区上都不存在这个目录。它实际是个实时的、驻留在内存中的文件系统。

**13. /root**Linux超级权限用户root的家目录；

**14. /sbin**大多是涉及系统管理的命令的存放，是超级权限用户root的可执行命令存放地，普通用户无权限执行这个目录下的命令；

这个目录和

/usr/sbin; /usr/X11R6/sbin或/usr/local/sbin 目录是相似的； 我们记住就行了，凡是目录sbin中包含的都是root权限才能执行的。

**15. /tmp** 临时文件目录，有时用户运行程序的时候，会产生临时文件。 /tmp就用来存放临时文件的。/var/tmp目录和这个目录相似。

许多程序在这里建立lock文件和存储临时数据。有些系统会在启动或关机时清空此目录。

**16. /usr** 这个是系统存放程序的目录，比如命令、帮助文件等。

这个目录下有很多的文件和目录。

当我们安装一个Linux发行版官方提供的软件包时，大多安装在这里。

如果有涉及服务器配置文件的，会把配置文件安装在/etc目录中。

/usr目录下包括：

涉及字体目录/usr/share/fonts ，

帮助目录 /usr/share/man或/usr/share/doc，

普通用户可执行文件目录/usr/bin 或/usr/local/bin 或/usr/X11R6/bin ，

超级权限用户root的可执行命令存放目录，比如 /usr/sbin 或/usr/X11R6/sbin或/usr/local/sbin 等；

还有程序的头文件存放目录/usr/include。

/usr/bin 这个目录是可执行程序的目录，普通用户就有权限执行； 当我们从系统自带的软件包安装一个程序时，他的可执行文件大多会放在这个目录。

/usr/sbin 这个目录也是可执行程序的目录，但大多存放涉及系统管理的命令。只有root权限才能执行；相似目录是/sbin 或/usr/local/sbin或/usr/X11R6/sbin等；

/usr/local 这个目录一般是用来存放用户自编译安装软件的存放目录；一般是通过源码包安装的软件，如果没有特别指定安装目录的话，一般是安装在这个目录中。这个目录下面有子目录。

/usr/lib 和/lib 目录相似，是库文件的存储目录；

/usr/share 系统共用的东西存放地，比如 /usr/share/fonts 是字体目录，是用户都共用的吧。

/usr/share/doc和/usr/share/man帮助文件，也是共用的吧；

/usr/src 是内核源码存放的目录，比如下面有内核源码目录，比如 linux 、linux-2.xxx.xx 目录等。有的系统也会把源码软件包安装在这里。比如Fedora/Redhat，当我们安装file.src.rpm的时候，这些软件包会安装在 /usr/src/redhat相应的目录中。请参考： 《file.src.rpm 使用方法的简单介绍》 。另外Fedhat 4.0 5.0，他的内核源码包的目录位于/usr/src/kernels目录下的某个目录中（只有安装后才会生成相应目录）；

**17. /var** 这个目录的内容是经常变动的，看名字就知道，我们可以理解为vary的缩写，/var下有/var/log这是用来存放系统日志的目录。

/var/www目录是定义Apache服务器站点存放目录；/var/lib 用来存放一些库文件，比如MySQL的，以及MySQL数据库的的存放地；

/var/log 系统日志存放，分析日志要看这个目录的东西；

/var/spool 打印机、邮件、代理服务器等假脱机目录；

参考:

1.Linux 文件系统的目录结构

<http://blog.sina.com.cn/s/blog_662234020101ebfd.html>

# 内核源码目录结构

|  |  |
| --- | --- |
| /arch | 不同CPU架构下的核心代码。其中的每一个子目录都代表Linux支持的CPU架构 |
| /block | 块设备通用函数 |
| /crypto | 常见的加密算法的C语言实现代码，譬如crc32、md5、sha1等 |
| /Documentation | 说明文档，对每个目录的具体作用进行说明 |
| /drivers | 内核中所有设备的驱动程序，其中的每一个子目录对应一种设备驱动 |
| /firmware | 固件代码 |
| /fs | Linux支持的文件系统代码，及各种类型的文件的操作代码。每个子目录都代表Linux支持的一种文件系统类型 |
| /include | 内核编译通用的头文件 |
| /init | 内核初始化的核心代码 |
| /ipc | 内核中进程间的通信代码 |
| /kernel | 内核的核心代码，此目录下实现了大多数Linux系统的内核函数。与处理器架构相关的内核代码在/kernel/$ARCH/kernel |
| /lib | 内核共用的函数库，与处理器架构相关的库在/kernel/$ARCH/lib |
| /mm | 内存管理代码，譬如页式存储管理内存的分配和释放等。与具体处理器架构相关的内存管理代码位于/arch/$ARCH/mm目录下 |
| /net | 网络通信相关代码 |
| /samples | 示例代码 |
| /scripts | 用于内核配置的脚本文件，用于实现内核配置的图形界面 |
| /security | 安全性相关的代码 |
| /tools | Linux中的常用工具 |
| /usr | 内核启动相关的代码 |
| /virt | 内核虚拟机相关的代码 |

# DTS

在kernel里面多了一种dts文件，因为当初自学Linux时和在第一家公司做物联网模型时都是用的比较老的内核，内核代码还比较混乱，没有采用dts这种方便简洁的格式。后面才知道这是因为Linus的一句”this whole arm thing is a fucking pain in ass“促进改革的，记得Linux早期代码里面板级细节都是在C文件中描述的，代码就显得十分臃肿和混乱。如此优化之后就显得简洁多了，并且也更易于学习、移植。

修复

# 内核日志

<https://www.zhihu.com/question/31598183>

# 网络编程

## **I/O模式**

　　对于**一次IO访问（这回以read举例）**，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的缓冲区，最后交给进程。所以说，**当一个read操作发生时，它会经历两个阶段：**  
　　1. 等待数据准备 (Waiting for the data to be ready)  
　　2. 将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)

正式因为这两个阶段，**linux系统产生了下面五种网络模式的方案：**  
　　-- 阻塞 I/O（blocking IO）  
　　-- 非阻塞 I/O（nonblocking IO）  
　　-- I/O 多路复用（ IO multiplexing）  
　　-- 信号驱动 I/O（ signal driven IO）  
　　-- 异步 I/O（asynchronous IO）

注：由于signal driven IO在实际中并不常用，所以我这只提及剩下的四种IO 模型。

<https://www.cnblogs.com/zingp/p/6863170.html>

### I/O多路复用

　I/O多路复用实际上就是用select, poll, epoll监听多个io对象，当io对象有变化（有数据）的时候就通知用户进程。好处就是单个进程可以处理多个socket。当然具体区别我们后面再讨论，现在先来看下I/O多路复用的流程

（1）当用户进程调用了select，那么整个进程会被block；

      （2）而同时，kernel会“监视”所有select负责的socket；

　　（3）当任何一个socket中的数据准备好了，select就会返回；

　　（4）这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。

　　所以，I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的**任意一个进入读就绪状态**，select()**函数就可以返回**。

　　这个图和blocking IO的图其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个connection。

　　所以，**如果处理的连接数不是很高的话**，使用**select/epoll的web server不一定**比使用**多线程 + 阻塞 IO**的web server性能更好，可能延迟还更大。

　　select/epoll的优势**并不是**对于单个连接能处理得**更快**，而是在于能处理**更多**的连接。）

　　在IO multiplexing Model中，实际中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。

# IPC

Linux已经拥有的进程间通信IPC手段包括(Internet Process Connection)： 管道（Pipe）、信号（Signal）和跟踪（Trace）、插口（Socket）、报文队列（Message）、共享内存（Share Memory）和信号量（Semaphore）。本文详细介绍Binder作为Android主要IPC方式的优势。

## 套接字

## RIL

# kernel/init/Kconfig

当执行#make menuconfig时会出现内核的配置界面，所有配置工具都是通过读取"arch/$(ARCH)Kconfig"文件来生成配置界面，这个文件就是所有配置的总入口，它会包含其他目录的Kconfig

注：对于uboot来讲，配置的总入口在uboot/Kconfig

Kconfig的作用：Kconfig用来配置内核，它就是各种配置界面的源文件，内核的配置工具读取各个Kconfig文件，生成配置界面供开发人员配置内核，最后生成配置文件**.config**

# arch\arm64\configs

https://www.cnblogs.com/leaven/archive/2010/05/13/1734466.html

# REF

# Fan

|  |  |  |
| --- | --- | --- |
| T | B | Speed |
| [0,53> |  | 0 |
| [53,58> | [0,80> | 0 |
|  | [80,100] | Max/2 |
| [58,63> | [0,50> | 0 |
|  | [50,80> | Max/2 |
|  | [80,100] | Max |

80du 从何而来，

怎么判断产品get\_broard\_ID

这么多宏定义，可读性非常差！怎么维护

风扇逻辑还是有问题。

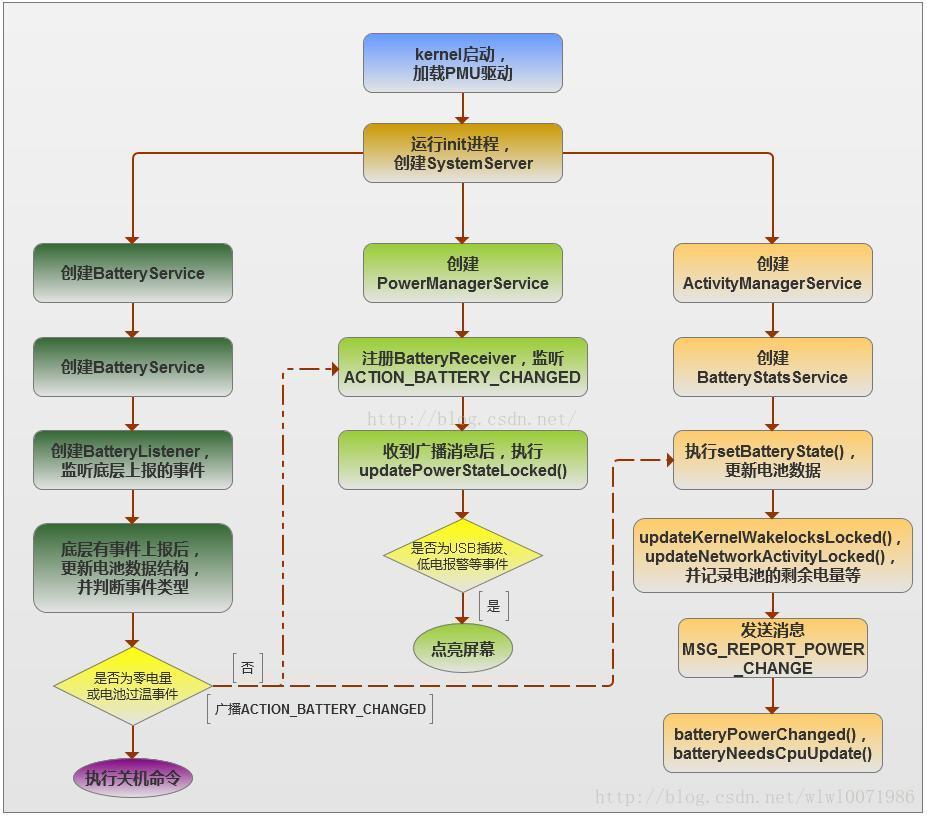
Bat:

# 电池管理

## Android 电池服务

Android电池服务，用来监听内核上报的电池事件，并将最新的电池数据上报给系统，系统收到新数据后会去更新电池显示状态、剩余电量等信息。如果收到过温报警和低电报警，系统会自动触发关机流程，保护电池和机器不受到危害。

android电池服务的启动和运行流程：



Android电池服务的源码结构

         Framework\base\services\[**Java**](http://lib.csdn.net/base/javase)\com\android\server  
        ├── SystemServer.[**Java**](http://lib.csdn.net/base/java)  
                 创建BatteryServices、PowerManagerService、ActivityManagerService  
        ├── BatterySevices.java  
                 监听底层上报的battery事件，广播电池发生改变的消息

         Framework\base\services\java\com\android\server\am  
        ├── ActivityManagerService.java  
                 创建BatteryStatsService  
        ├── BatteryStatsService.java  
                 统计和记录电池参数的信息

         Framework\base\services\java\com\android\server\power  
        ├── PowerManagerService.java  
                 监听电池发生变化的广播消息，并调节系统的电源状态，例如亮屏

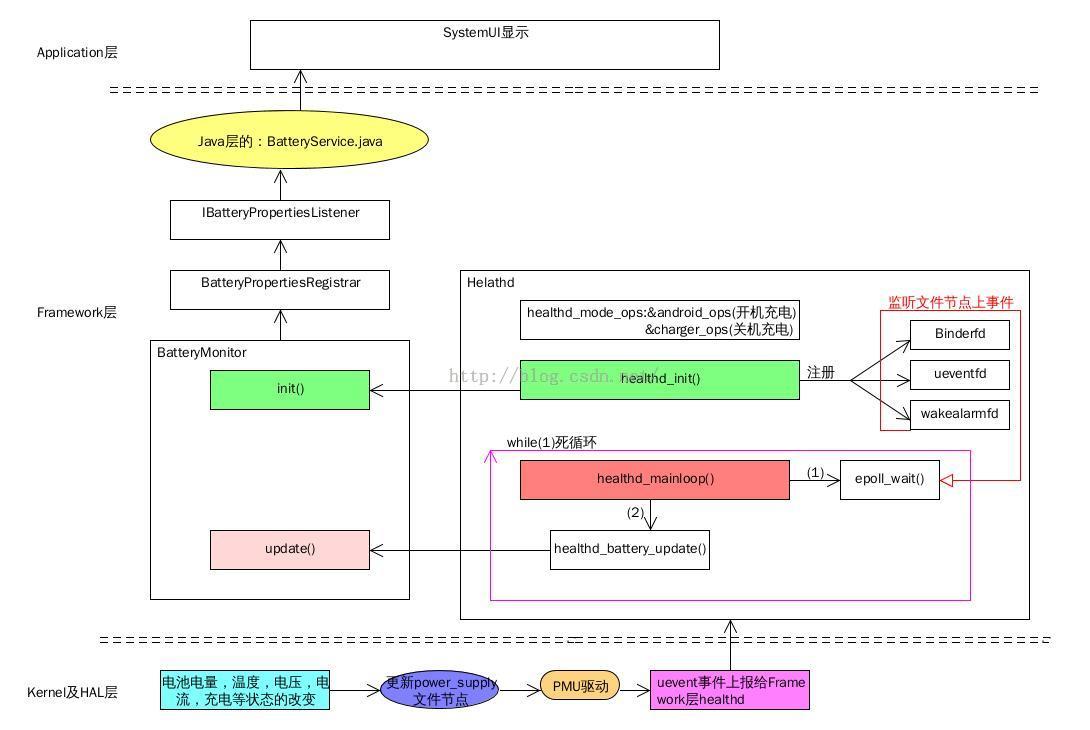
         Framework\base\core\java\com\internal\os\  
        ├── BatteryStatsImpl.java  
                 统计和记录电池参数的信息，并通知其他模块

         System\core\healthd  
        ├── healthd.cpp  
                 创建uevent socket，监听内核上报的内核事件  
        ├── BatteryMonitor.cpp  
                 初始化本地电池**[数据结构](http://lib.csdn.net/base/datastructure" \t "_blank" \o "算法与数据结构知识库)**，将power\_supply路径下属性节点路径填充进去，  
        ├── BatteryMonitor.h  
        ├── BatteryPropertiesRegistrar.cpp  
                 创建电池属性监听器，并将其注册到Android的系统服务中  
        ├── BatteryPropertiesRegistrar.h

## 二、Healthd

该模型向下监听来自底层的电池事件，向上传递电池数据信息给Framework层的BatteryService用来计算电池电量相关信息，

BatteryService通过传递来的数据来计算电池电量等信息，因此healthd在电池管理系统中起着承上启下的作用。



healthd的具体调用流程[深入分析android5.1 healthd](http://blog.csdn.net/kc58236582/article/details/47300413" \t "_blank)这篇文章讲得很清楚。

## 三、驱动

Android电源管理底层用的是**[Linux](http://lib.csdn.net/base/linux" \t "_blank" \o "Linux知识库)**

power\_supply框架，内核提供给电池驱动的接口是结构体power\_supply结构体。

**[cpp]** [view plain](http://blog.csdn.net/bhj1119/article/details/52947344) [copy](http://blog.csdn.net/bhj1119/article/details/52947344)

1. **struct** power\_supply {
2. **const** **char** \*name;
3. **enum** power\_supply\_type type;
4. **enum** power\_supply\_property \*properties;
5. **size\_t** num\_properties;
7. **char** \*\*supplied\_to;
8. **size\_t** num\_supplicants;
10. **int** (\*get\_property)(**struct** power\_supply \*psy,
11. **enum** power\_supply\_property psp,
12. **union** power\_supply\_propval \*val);
13. **int** (\*set\_property)(**struct** power\_supply \*psy,
14. **enum** power\_supply\_property psp,
15. **const** **union** power\_supply\_propval \*val);
16. **int** (\*property\_is\_writeable)(**struct** power\_supply \*psy,
17. **enum** power\_supply\_property psp);
18. **void** (\*external\_power\_changed)(**struct** power\_supply \*psy);
19. **void** (\*set\_charged)(**struct** power\_supply \*psy);
21. /\* For APM emulation, think legacy userspace. \*/
22. **int** use\_for\_apm;
24. /\* Driver private data \*/
25. **void** \*drv\_data;//add by bhj
27. /\* private \*/
28. **struct** device \*dev;
29. **struct** work\_struct changed\_work;
30. spinlock\_t changed\_lock;
31. **bool** changed;
32. **struct** wake\_lock work\_wake\_lock;
33. **struct** delayed\_work deferred\_register\_work;
35. #ifdef CONFIG\_LEDS\_TRIGGERS
36. **struct** led\_trigger \*charging\_full\_trig;
37. **char** \*charging\_full\_trig\_name;
38. **struct** led\_trigger \*charging\_trig;
39. **char** \*charging\_trig\_name;
40. **struct** led\_trigger \*full\_trig;
41. **char** \*full\_trig\_name;
42. **struct** led\_trigger \*online\_trig;
43. **char** \*online\_trig\_name;
44. **struct** led\_trigger \*charging\_blink\_full\_solid\_trig;
45. **char** \*charging\_blink\_full\_solid\_trig\_name;
46. #endif
47. };</span>

 内核主要通过get\_property这个函数指针来获得驱动中的有关电池的信息，而这个函数在内核中只给出了声明，我们在写驱动的

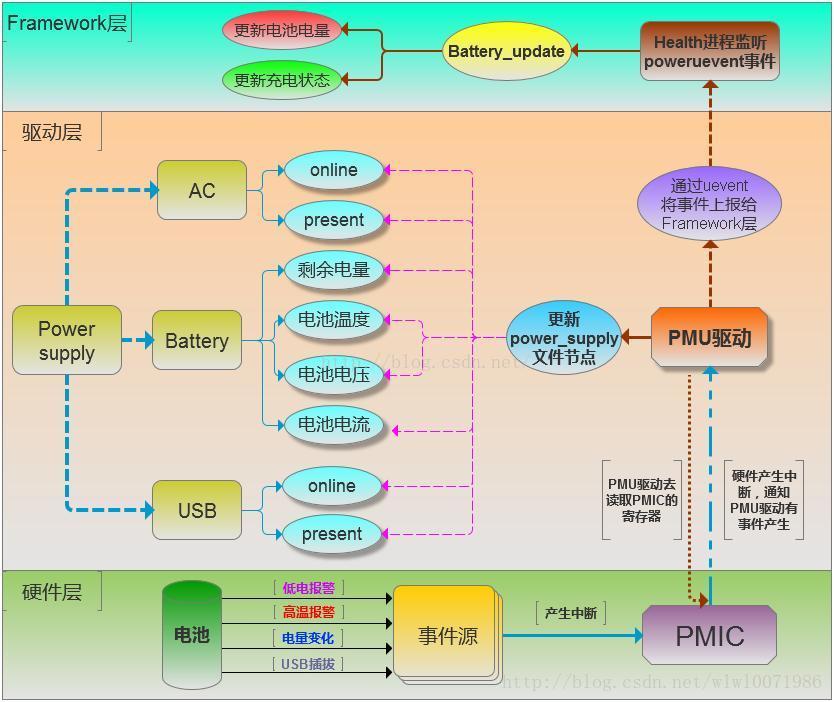
时候要自己实现这个函数，即将自己写的函数赋值给这个函数指针，当内核需要驱动中电源信息的时候就回调这个get\_property函数

。另外，我们写驱动程序的时候又要给用户提供接口，内核中提供给用户的接口就是sysfs，通过读取sysfs文件系统中文件内容，就

可以得到电源的信息。内核主要通过两个文件power\_supply\_class.c和power\_supply\_core.c，我们调用其中的函数就可以把电源（电

池，USB或AC）的信息展现给用户，有关电源的属性写在/sys/class/powersupply文件夹下（此文件夹为程序运行后所生成的）。

电池系统从底层向Framework层上报数据的流程：



## REF

[Android 电池管理系统](http://blog.csdn.net/lqxandroid2012/article/details/70050071)

[Android BatteryStatsHelper深入理解(and5.1)](http://blog.csdn.net/kc58236582/article/details/47273507)

[Android的文件系统结构](http://blog.csdn.net/conowen/article/details/7251057)

# AlarmManager

### [认领AlarmManager源码解析](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&cad=rja&uact=8&ved=0ahUKEwjtpdipvOLVAhUBi5QKHcTUBPcQFgg3MAM&url=https%3A%2F%2Fgithub.com%2FLittleFriendsGroup%2FAndroidSdkSourceAnalysis%2Fissues%2F68&usg=AFQjCNEuaO5NC5s40NH1uMuWKzH_74Eg_w)

<https://github.com/LittleFriendsGroup/AndroidSdkSourceAnalysis>

setings时间设置原理

## 什么是Alarm

Alarm是**[android](http://lib.csdn.net/base/android" \t "_blank" \o "Android知识库)**提供的用于完成闹钟式定时任务的类，系统通过AlarmManager来管理所有的Alarm，Alarm支持一次性定时任务和循环定时任务，使用方式：

1. AlarmManager alarmMgr = (AlarmManager) getSystemService(Context.ALARM\_SERVICE);
2. Intent intent = new
3. Intent("android.intent.action.POWER\_USAGE\_SUMMARY");
4. alarmMgr.set(AlarmManager.ELAPSED\_REALTIME
5. , SystemClock.elapsedRealtime() + 5 \* 1000//5S后打开电池界面
6. , PendingIntent.getActivity(getApplicationContext(),0, intent, PendingIntent.FLAG\_UPDATE\_CURRENT));

Alarm和Timer以及Handler在定时任务上的区别

**相同点**：三者都可以完成定时任务，都支持一次性定时和循环定时（注：Handler可以间接支持循环定时任务）

**不同点**：Handler和Timer在定时上是类似的，二者在系统休眠的情况下无法正常工作，定时任务不会按时触发。Alarm在系统休眠的情况下可以正常工作，并且还可以决定是否唤醒系统，同时Alarm在自身不启动的情况下仍能正常收到定时任务提醒，但是当系统重启或者应用被杀死的情况下，Alarm定时任务会被取消。另外，从Android4.4开始，Alarm事件默认采用非精准方式，即定时任务可能会有小范围的提前或延后，当然我们可以强制采用精准方式，而在此之前，Alarm事件都是精准方式。

## Alarm与Binder的交互

code：AlarmManager#set

**[java]** [view plain](http://blog.csdn.net/singwhatiwanna/article/details/18448997) [copy](http://blog.csdn.net/singwhatiwanna/article/details/18448997)

1. **public** **void** set(**int** type, **long** triggerAtMillis, PendingIntent operation) {
2. setImpl(type, triggerAtMillis, legacyExactLength(), 0, operation, **null**);
3. }
5. **public** **void** set(**int** type, **long** triggerAtMillis, **long** windowMillis, **long** intervalMillis,
6. PendingIntent operation, WorkSource workSource) {
7. setImpl(type, triggerAtMillis, windowMillis, intervalMillis, operation, workSource);
8. }
10. **private** **void** setImpl(**int** type, **long** triggerAtMillis, **long** windowMillis, **long** intervalMillis,
11. PendingIntent operation, WorkSource workSource) {
12. **if** (triggerAtMillis < 0) {
13. /\* NOTYET
14. if (mAlwaysExact) {
15. // Fatal error for KLP+ apps to use negative trigger times
16. throw new IllegalArgumentException("Invalid alarm trigger time "
17. + triggerAtMillis);
18. }
19. \*/
20. triggerAtMillis = 0;
21. }
23. **try** {
24. //定时任务实际上都有mService来完成，也就是说AlarmManager只是一个空壳
25. //从下面的构造方法可以看出，这个mService是IAlarmManager类型的，而IAlarmManager是一个接口
26. //如果大家了解AIDL就应该知道IAlarmManager应该是一个AIDL接口
27. mService.set(type, triggerAtMillis, windowMillis, intervalMillis, operation,
28. workSource);
29. } **catch** (RemoteException ex) {
30. }
31. }
33. AlarmManager(IAlarmManager service, Context ctx) {
34. mService = service;
36. **final** **int** sdkVersion = ctx.getApplicationInfo().targetSdkVersion;
37. mAlwaysExact = (sdkVersion < Build.VERSION\_CODES.KITKAT);
38. }

说明：我对代码进行了注释，从注释可以看出，现在我们需要去找到这个mService，其实我已经帮大家找到了，它就是AlarmManagerService，看下它的类的声明：

class AlarmManagerService extends IAlarmManager.Stub

很显然，AlarmManagerService的确实现了IAlarmManager接口，为什么是显然呢？因为按照AIDL的规范，IAlarmManager.Stub是按照如下这种方式声明的：

**[java]** [view plain](http://blog.csdn.net/singwhatiwanna/article/details/18448997) [copy](http://blog.csdn.net/singwhatiwanna/article/details/18448997)

1. **public** **static** **abstract** **class** Stub **extends** Binder **implements** IAlarmManager {
3. **public** **static** IAlarmManager asInterface(IBinder obj)
4. ...
5. }

可见这个Stub类就是一个普通的Binder，只不过它实现了IAlarmManager接口。它还有一个静态方法asInterface，这个方法很有用，通过它，我们就可以将IBinder对象转换成IAlarmManager的实例，进而通过实例来调用其方法。

code：SystemServer

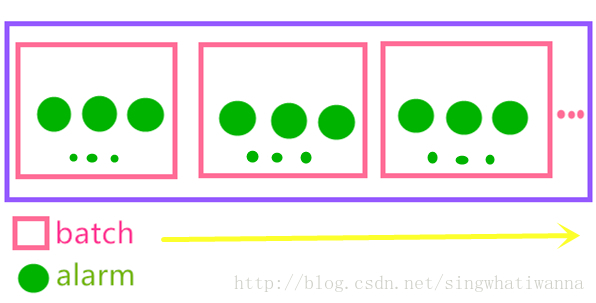
1. //main方法，由底层调用
2. **public** **static** **void** main(String[] args) {
3. //这里是关键，ServerThread被创建，同时其initAndLoop被调用
4. ServerThread thr = **new** ServerThread();
5. thr.initAndLoop();
6. }

code：ServerThread#initAndLoop

1. **public** **void** initAndLoop() {
2. //主线程Looper被创建
3. Looper.prepareMainLooper();      android.os.Process.setThreadPriority(
4. android.os.Process.THREAD\_PRIORITY\_FOREGROUND);
5. BinderInternal.disableBackgroundScheduling(**true**);
6. android.os.Process.setCanSelfBackground(**false**);
7. ...此处省略
8. //下面是各种Binder服务，从名字我们应该能够大致看出它们所对应的Manager
9. AlarmManagerService alarm = **null**;
10. Slog.i(TAG, "Alarm Manager");
11. //这里AlarmManager对应的Binder服务被创建
12. alarm = **new** AlarmManagerService(context);
13. //将AlarmManagerService加入ServiceManager中统一管理
14. ServiceManager.addService(Context.ALARM\_SERVICE, alarm);
15. ActivityManagerService.self().setWindowManager(wm);
16. }

## Alarm机制分析

在分析源码之前，我先来描述下Alarm的工作原理：从Android4.4开始，Alarm默认为非精准模式，除非显示指定采用精准模式。在非精准模式下，Alarm是批量提醒的，每个alarm根据其触发时间和最大触发时间的不同会被加入到不同的batch中，同一个batch的不同alarm是同时发生的，这样就无法实现精准闹钟，官方的解释是批量处理可以减少设备被唤醒次数以及节约电量，不过针对精准闹钟，官方预留的方法是setExact和setWindow，二者都是通过将时间窗口定义为0来实现精准闹钟的，因为时间窗口为0，意味着触发时间和最大触发时间是一样的，因为典型的情况下：最大触发时间= 触发时间 + 时间窗口。同时所有的batch是按开始时间升序排列的，在一个batch内部，不同的闹钟也是按触发时间升序排列的，所以闹钟的唤醒顺序是按照batch的排序依次触发的，而同一个batch中的alarm是同时触发的，可以用下面这个示意图来描述：

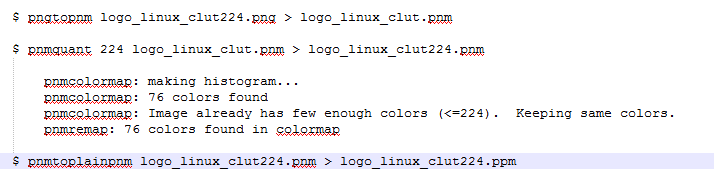


上图是示意图，系统中可以有多个batch，每个batch中可以有多个alarm。下面我们分析一下AlarmManagerService中的代码。其入口方法为set，set又调用了setImplLocked，所以我们直接看setImplLocked。

code：AlarmManagerService#setImplLocked

<http://blog.csdn.net/singwhatiwanna/article/details/18448997>

# 修改内核logo



<http://www.52rd.com/blog/Detail_RD.Blog_wuyaya_67898.html>

# TASK

[Linux内核（Kernel-3.18） - Linux Input 子系统分析](http://zhoujinjian.cc/2018/04/01/Linux%E5%86%85%E6%A0%B8%EF%BC%88Kernel-3-18%EF%BC%89-Input-%E5%AD%90%E7%B3%BB%E7%BB%9F%E5%88%86%E6%9E%90-i-wonder/)