# 整体框架

Linux抽象了看门狗框架去管理看门狗驱动，以前的看门狗驱动需要自己实现包含open/write/ioctl等操作的设备操作集file\_operations，然后向系统注册一个misc设备，现在在看门狗框架的管理下，我们只需实现看门狗的操作集然后向系统注册看门狗设备即可。linux内核下看门狗驱动可以分为三层：

1. watchdog\_dev.c中定义的接口层，统一了与用户空间交互的接口。
2. watchdog\_core.c为核心层，实现了看门狗设备的注册过程。
3. 设备层则是具体的看门狗设备，需要实现具体功能。

# 看门狗数据结构

## 看门狗设备

看门狗设备由如下结构进行描述：

struct watchdog\_device {

int id;//看门狗ID，看门狗设备注册时自动申请。id为0时既会添加cdev /dev/watchdog0，也会添加老的miscdev /dev/watchdog。

struct device \*parent;//父设备，可为NULL

const struct attribute\_group \*\*groups;//sysfs属性

const struct watchdog\_info \*info;//看门狗附加信息

const struct watchdog\_ops \*ops;//看门狗操作集

const struct watchdog\_governor \*gov;//预超时调控器

unsigned int bootstatus;//启动时的看门狗状态

unsigned int timeout;//看门狗超时时间（单位：s）

unsigned int pretimeout;//预超时时间（单位：s），预超时指的是超时时间前多少时间，如timeout=5，pretimeout=3,则应该在2s时报告预超时。

unsigned int min\_timeout;//最小超时时间

unsigned int max\_timeout;//最大超时时间，如果设置了max\_hw\_heartbeat\_ms，则不使用此参数

unsigned int min\_hw\_heartbeat\_ms;//硬件能容忍的最低心跳间隔，通常为0

unsigned int max\_hw\_heartbeat\_ms;//最大硬件心跳，单位ms，如果timeout大于max\_hw\_heartbeat\_ms，也就是硬件处理不了足够大的超时时间，框架程序会向看门狗驱动发送心跳，直到看门狗超时为止。如果驱动没有实现stop功能，则这个值必须设置。

struct notifier\_block reboot\_nb;//框架内部使用，利用notifier机制在reboot时停止调用了 watchdog\_stop\_on\_reboot的看门狗。

struct notifier\_block restart\_nb;//框架内部使用，如果看门狗能够重启设备，实现了ops->restart，会利用notifier机制在restart时调用。

void \*driver\_data;//私有数据，通过 watchdog\_set\_drvdata/ watchdog\_get\_drvdata设置与访问

struct watchdog\_core\_data \*wd\_data;//看门狗核心内部数据，注册看门狗设备时自动申请，内部使用

unsigned long status;//看门狗状态，不同位对应不同含义

struct list\_head deferred;//延期注册时使用

};

其中的状态status位含义如下：

WDOG\_ACTIVE：看门狗是否处于活动状态，如果已处于活动状态，那么用户空间需要向设备发送心跳请求。

WDOG\_NO\_WAY\_OUT：设置此位后，看门狗将无法停止。

WDOG\_STOP\_ON\_REBOOT：重启时看门狗停止。

WDOG\_HW\_RUNNING：表示硬件看门狗正在运行，如果看门狗硬件无法停止，则必须设置此位，如果看门狗设备在系统启动后在设备打开前已运行，也应该设置此位，内核能保证看门狗keepalive。

## 看门狗操作集

看门狗操作集定义如下：

struct watchdog\_ops {

struct module \*owner;

/\* mandatory operations \*/

int (\*start)(struct watchdog\_device \*);

int (\*stop)(struct watchdog\_device \*);

/\* optional operations \*/

int (\*ping)(struct watchdog\_device \*);

unsigned int (\*status)(struct watchdog\_device \*);

int (\*set\_timeout)(struct watchdog\_device \*, unsigned int);

int (\*set\_pretimeout)(struct watchdog\_device \*, unsigned int);

unsigned int (\*get\_timeleft)(struct watchdog\_device \*);

int (\*restart)(struct watchdog\_device \*, unsigned long, void \*);

long (\*ioctl)(struct watchdog\_device \*, unsigned int, unsigned long);

};

我们在实现设备驱动时主要就是负责填充此结构，这个结构中有必须要实现的，也有可选的。

**必须实现：**

owner：必须指定，可在看门狗active时锁死模块，防止看门狗正在使用时卸载模块导致系统崩溃。

start：启动看门狗。

**可选实现：**

stop：停止看门狗，如果没有实现则必须设置max\_hw\_heartbeat\_ms。

ping：向看门狗设备发送keepalive ping，如果看门狗设备不支持此功能，看门狗核心会使用start来重启看门狗，同样可以达到看门狗keepalive的目的。

status：报告看门狗状态，使用WDIOF\_\*状态进行报告，WDIOF\_MAGICCLOSE 和 WDIOF\_KEEPALIVEPING 由看门狗核心报告，如果此函数没有实现，内核在报告看门狗状态时就会改为报告bootstatus 。

set\_timeout：更改看门狗超时时间。如果设备驱动中在set\_timeout中除了获取超时时间，不干其他事情，则可不用实现此函数，因为内核中会将超时时间保存在watchdog\_device->timeout中。需要watchdog\_info->options设置WDIOF\_SETTIMEOUT才支持设置超时时间。

set\_pretimeout：更改看门狗预超时时间，这个值需要小于timeout，内核会自动检查，同样如果除了需要获取值外不用做其他操作，则无需实现此函数，内核会自动保存到watchdog\_device->pretimeout中。需要watchdog\_info->options设置WDIOF\_PRETIMEOUT才支持设置预超时时间。

get\_timeleft：获取超时的剩余时间。

ioctl：自己实现的ioctl，可以部分或全部覆盖内核已经实现的ioctl命令，如果命令不支持，应该返回-ENOIOCTLCMD，此时内核会执行自己实现的命令。

# Magic Close 特性

如果设备关闭时直接将看门狗关闭，那么如果将看门狗守护进程kill掉时，此时系统应该重启但是并不会，为了避免这种情况，可以在watchdog\_info->options中设置WDIOF\_MAGICCLOSE，此时必须在关闭设备前向设备发送特定的魔术字符“V”，才能在关闭设备时将看门狗关闭，使用此特性可以保证看门狗守护进程异常退出时一定能让看门狗触发复位。

# 相关API说明

void watchdog\_set\_drvdata(struct watchdog\_device \*wdd, void \*data);

设置私有数据。

void \*watchdog\_get\_drvdata(struct watchdog\_device \*wdd);

获取私有数据。

void watchdog\_set\_nowayout(struct watchdog\_device \*wdd,

int nowayout);

设置看门狗为无法停止。

int watchdog\_init\_timeout(struct watchdog\_device \*wdd,

unsigned int timeout\_parm,

struct device \*dev);

初始化超时时间，如果不调用API进行初始化，也可以直接在注册看门狗设备前将wdd->timeout成员直接赋值，如果不进行赋值也不进行初始化，那么看门狗的超时时间就是0，那么看门狗在使能时就会立马触发复位。

void watchdog\_stop\_on\_reboot(struct watchdog\_device \*wdd);

重启时禁用看门狗。

void watchdog\_set\_restart\_priority(struct watchdog\_device \*wdd, int priority);

更改重启处理程序优先级，优先级值越大，优先级越高，越能先于其他重启处理。

void watchdog\_notify\_pretimeout(struct watchdog\_device \*wdd);

发出预超时通知，可在中断上下文调用，如果启用了看门狗预超时调控器

（CONFIG\_WATCHDOG\_PRETIMEOUT\_GOV），则由看门狗的预超时调控器进行操作。如果未启用，则只是将通知消息打印到内核日志。

# 预超时调控器

# 看门狗ioctl命令

ioctl(fd, WDIOC\_KEEPALIVE, 0);

喂狗，与写设备效果一致，参数内核会忽略。

ioctl(fd, WDIOC\_SETTIMEOUT, &timeout);

设置超时时间。

ioctl(fd, WDIOC\_GETTIMEOUT, &timeout);

获取超时时间。

ioctl(fd, WDIOC\_SETPRETIMEOUT, &pretimeout);

设置预超时时间，为0时禁用。

ioctl(fd, WDIOC\_GETPRETIMEOUT, &timeout);

获取预超时时间。

ioctl(fd, WDIOC\_GETTIMELEFT, &timeleft);

获取看门狗复位剩余时间。

struct watchdog\_info ident;

ioctl(fd, WDIOC\_GETSUPPORT, &ident);

获取看门狗信息。ident中的options字段位含义如下：

WDIOF\_OVERHEAT：上次因为CPU过热导致看门狗重启设备。

WDIOF\_FAFAULT：看门狗监测的风扇出现故障。

WDIOF\_EXTERN1：

WDIOF\_EXTERN2：外部监测引脚被触发。

WDIOF\_POWERUNDER：电源欠压

WDIOF\_CARDRESET：上次重启是因为看门狗触发复位

WDIOF\_POWEROVER：电源过压

WDIOF\_KEEPALIVEPING：自上次查询，看门狗被喂过

WDIOF\_SETTIMEOUT：//可以设置超时时间

WDIOF\_PRETIMEOUT：//支持预超时

ioctl(fd, WDIOC\_GETTEMP, &temperature);

温度读取，单位为华氏度

ioctl(fd, WDIOC\_SETOPTIONS, &options);

设置选项：

WDIOS\_DISABLECARD 关闭看门狗定时器

WDIOS\_ENABLECAR 打开看门狗定时器

# 示例说明

内核中已经提供了软件看门狗，具体实现请参考softdog.c，其只能提供一个软件狗，我在此基础上进行了更改，可以根据模块输入参数softdog\_num决定注册软件看门狗的个数，方便应用层多进程、多线程的守护，当然用一个看门狗去守护多线程、多进程的方法也有很多。

测试程序使用默认的超时时间10s，预超时时间为3s，开始5s持续喂狗，5s后停止喂狗后退出，系统会提示此时看门狗并未退出，系统复位前3s预超时调控器会进行提示。执行./test\_watchdog /dev/watchdog1，结果如下：

watchdog default timeout:10s

[ 1267.674634] watchdog: watchdog1: watchdog did not stop!

[ 1274.720666] watchdog1: pretimeout event

[ 1277.920651] Initiating system reboot

如果想在程序退出时停止看门狗，需向看门狗写入魔术字符‘V’。