# 电源管理

对于消费类电子产品（嵌入式系统）来说，因为由电池供电，因此功耗（电源）管理相当重要。Linux功耗管理非常复杂，牵扯到系统级的待机、频率电压变换、系统空闲时的处理以及每个设备驱动对系统待机的支持和每个设备的运行时功耗管理等。

本书将与节能相关的管理工作称为功耗管理，而将对设备供电（电池、交流电和USB供电）的管理称为电源管理，电源管理可视为功耗管理的一部分。

## 1 cpufreq子系统

内核cpufreq子系统用于调节处理器工作频率，从而达到节能的目的。cpufreq子系统通过sysfs文件系统对向用户进程提供操作接口，用户可通过此接口设置CPU的调频策略或直接设置CPU的工作频率。CPU工作频率的改变由前一小节介绍的CLOCK模块（时钟硬件）执行。

### 1.1概述

内核实现了cpufreq子系统框架用于调整CPU频率，用户可通过sysfs文件系统对CPU频率进行调整。cpufreq子系统代码位于/drivers/cpufreq/目录下。

#### ■驱动框架

cpufreq子系统框图可用下图表示：



每个CPU核关联到一个cpufreq策略（调频策略），cpufreq驱动用于调整时钟硬件提供给CPU的时钟频率。用户通过sysfs文件系统中cpufreq策略的属性文件设置调频策略或频率值，经过cpufreq驱动设置时钟硬件的输出频率。

Linux内核cpufreq子系统驱动框架如下图所示：



上图中cpufreq\_governor结构体实例（由双链表管量）表示调频策略的类型，内核调频策略类型如下：

cpufreq\_performance：CPU以最高频率运行，以获得最佳的性能。

cpufreq\_powersave：CPU以最低频率运行，以达到最节能的效果。

cpufreq\_ondemand：CPU平时以低速方式运行，当系统负载提高时按需自动提高频率。

cpufreq\_conservative：与cpufreq\_ondemand相似，区别在于动态频率在变更频率时采用渐进的方式。

cpufreq\_userspace：由用户进程设置CPU的运行频率（root用户）。

cpufreq\_governor实例由全局双链表管理，实例在/drivers/cpufreq/cpufreq\_xxx.c文件内实现并注册，可通过配置选项选择编译哪些策略实例到内核以及默认的策略。

cpufreq驱动由cpufreq\_driver结构体表示，实例由处理器（平台）相关代码实现，并向内核注册。驱动实例主要是实现对时钟硬件输出频率的设置，例如：target()函数用于按参数指定的频率值设置时钟硬件，target\_index()函数用于按频率列表项中指定的频率设置时钟硬件。

cpufreq\_driver实例需要向内核注册，在注册的过程中将为CPU核创建和初始化策略cpufreq\_policy实例，关联默认的cpufreq\_governor实例，并调用此实例的governor()函数，在此函数中将调用注册驱动中的函数设置硬件时钟输出频率。

cpufreq\_policy实例通过kobject实例成员导出到sysfs文件系统，用户可通过其属性文件执行设置调频策略、查看或设置CPU频率等操作。

#### ■初始化

cpufreq子系统初始化函数cpufreq\_core\_init()位于/drivers/cpufreq/cpufreq.c内，代码如下：

static int \_\_init cpufreq\_core\_init(void)

{

if (cpufreq\_disabled())

return -ENODEV;

**cpufreq\_global\_kobject = kobject\_create()**; /\*创建kobject实例\*/

/\*被添加后创建/sys/devices/system/cpu/cpufreq/目录，此目录下保存cpufreq全局属性\*/

BUG\_ON(!cpufreq\_global\_kobject);

register\_syscore\_ops(&**cpufreq\_syscore\_ops**); /\*注册syscore\_ops实例\*/

return 0;

}

core\_initcall(cpufreq\_core\_init);

初始化函数内创建kobject实例，此实例导出到sysfs文件下的目录为/sys/devices/system/cpu/cpufreq/（在第一次向此实例添加属性时添加实例到sysfs）。cpufreq\_sysfs\_create\_file(const struct attribute \*attr)接口函数用于在此目录下添加属性文件，表示cpufreq子系统的全局属性。

初始化函数内还注册了syscore\_ops结构体实例cpufreq\_syscore\_ops，syscore\_ops结构体定义在头文件/include/linux/syscore\_ops.h内：

struct syscore\_ops {

struct list\_head **node**; /\*双链表节点\*/

int (\*suspend)(void); /\*syscore\_suspend()函数中调用\*/

void (\*resume)(void); /\*syscore\_resume()函数中调用\*/

void (\*shutdown)(void); /\*syscore\_shutdown()函数中调用\*/

};

syscore\_ops结构体用于注册系统睡眠、唤醒、关机等时机需要执行的工作。

内核在/drivers/base/syscore.c文件内定义了全局双链表用于管理syscore\_ops实例：

static LIST\_HEAD(syscore\_ops\_list);

注册函数 register\_syscore\_ops()只是简单地将实例添加到全局双链表的末尾：

void register\_syscore\_ops(struct syscore\_ops \*ops)

{

mutex\_lock(&syscore\_ops\_lock);

**list\_add\_tail(&ops->node, &syscore\_ops\_list);**

mutex\_unlock(&syscore\_ops\_lock);

}

cpufreq\_core\_init()函数中注册的cpufreq\_syscore\_ops实例定义如下：

static struct syscore\_ops cpufreq\_syscore\_ops = {

.shutdown = **cpufreq\_suspend**, /\*对每个策略调用cpufreq\_driver->suspend(policy)函数\*/

};

在系统睡眠等时机调用的syscore\_suspend()/syscore\_resume()等函数中将调用syscore\_ops\_list双链表中实例中的相应函数。

### 1.2 cpufreq策略

内核定义了cpufreq\_governor结构体表示调频策略类型，策略类型由cpufreq子系统实现并注册。具体CPU核的调频策略由cpufreq\_policy结构体表示，它关联到某一调频策略类型。在向内核注册cpufreq\_driver实例时将为CPU创建cpufreq\_policy实例，并关联到默认的调频策略类型。

用户可通过配置选项选择编译哪几个策略类型到内核以及新创建的cpufreq\_policy实例关联的默认调频策略类型。

#### ■策略类型

调频策略类型由cpufreq\_governor结构体表示，定义在/include/linux/cpufreq.h头文件内：

struct cpufreq\_governor {

char **name**[CPUFREQ\_NAME\_LEN]; /\*名称，用于标识调频策略类型\*/

int initialized;

int (\***governor**) (struct cpufreq\_policy \*policy,unsigned int **event**); /\*执行策略的函数\*/

ssize\_t (\***show\_setspeed**) (struct cpufreq\_policy \*policy,char \*buf); /\*显示频率\*/

int (\***store\_setspeed**) (struct cpufreq\_policy \*policy,unsigned int freq); /\*设置频率\*/

unsigned int max\_transition\_latency; /\*延迟时间ns\*/

struct list\_head **governor\_list**; /\*将实例添加到全局双链表cpufreq\_governor\_list\*/

struct module \*owner;

};

cpufreq\_governor结构体中governor()函数在CPU策略有事件发生时调用，事件类型定义如下：

#define CPUFREQ\_GOV\_START 1 /\*CPU策略开始使用此策略类型\*/

#define CPUFREQ\_GOV\_STOP 2 /\*CPU策略停止使用此策略类型\*/

#define CPUFREQ\_GOV\_LIMITS 3 /\*策略限制\*/

#define CPUFREQ\_GOV\_POLICY\_INIT 4 /\*初始化\*/

#define CPUFREQ\_GOV\_POLICY\_EXIT 5 /\*退出策略类型，在STOP事件后\*/

内核定义了全局双链表cpufreq\_governor\_list用于管理所有注册的cpufreq\_governor实例：

static LIST\_HEAD(cpufreq\_governor\_list); /\*/drivers/cpufreq/cpufreq.c\*/

注册cpufreq\_governor实例的函数位于/drivers/cpufreq/cpufreq.c文件内，代码如下：

int **cpufreq\_register\_governor**(struct cpufreq\_governor \*governor)

{

int err;

...

mutex\_lock(&cpufreq\_governor\_mutex);

**governor->initialized = 0**;

err = -EBUSY;

if (!**find\_governor(governor->name)**) { /\*由名称在全局双链表中查找cpufreq\_governor实例\*/

err = 0;

**list\_add**(&governor->governor\_list, &cpufreq\_governor\_list);

/\*未找到相同实例则将其添加到双链表头部\*/

}

mutex\_unlock(&cpufreq\_governor\_mutex);

return err;

}

注册函数比较简单，首先在cpufreq\_governor\_list双链表中查找是否存在相同名称的实例，如果存在则函数返回，如果不存在则将实例插入到全局双链表的头部，函数返回。

cpufreq子系统定义了5个策略类型实例：

cpufreq\_performance：CPU以最高频率运行，以获得最佳的性能。

cpufreq\_powersave：CPU以最低频率运行，以达到最节能的效果。

cpufreq\_ondemand：CPU平时以低速方式运行，当系统负载提高时按需自动提高频率。

cpufreq\_conservative：与cpufreq\_ondemand相似，区别在于动态频率在变更频率时采用渐进的方式。

cpufreq\_userspace：由用户进程设置CPU的运行频率（root用户）。

cpufreq\_governor实例在/drivers/cpufreq/cpufreq\_xxx.c文件内实现并注册，用户可通过配置选项选择编译哪些策略类型文件。

#### ■CPU策略

CPU调频策略由cpufreq\_policy结构体表示，它是与硬件无关的，每个CPU核关联一个调频策略。不同的CPU核可关联相同的cpufreq\_policy实例。全局双链表cpufreq\_policy\_list管理着所有的cpufreq\_policy实例，如下图所示。



cpufreq\_policy实例通过kobject结构体成员导出到sysfs文件系统/sys/devices/system/cpu/cpuX/cpufreq/，其中cpuX表示CPU核（X表示编号），kobject实例属性导出到此目录下。频率列表表示CPU可设置的频率（可选，由cpufreq\_driver实现）。

cpufreq\_policy结构体定义在/include/linux/cpufreq.h头文件：

struct cpufreq\_policy {

cpumask\_var\_t cpus; /\* Online CPUs only（在线CPU位图）\*/

cpumask\_var\_t related\_cpus; /\* Online + Offline CPUs （策略适用的CPU位图）\*/

cpumask\_var\_t real\_cpus; /\* Related and present CPUs（CPU位图）\*/

unsigned int shared\_type; /\* ACPI: ANY or ALL affected CPUs should set cpufreq \*/

unsigned int **cpu**; /\*管理本实例的CPU核编号\*/

struct clk \***clk**; /\*硬件时钟，见前一小节\*/

**struct cpufreq\_cpuinfo cpuinfo**; /\*CPU信息，见下文\*/

unsigned int min; /\*策略限制的最小频率，kHz \*/

unsigned int max; /\*策略限制的最大频率，kHz \*/

unsigned int cur; /\*当前频率，kHz\*/

unsigned int **restore\_freq**; /\*频率调整前的频率\*/

unsigned int suspend\_freq; /\* freq to set during suspend \*/

unsigned int policy; /\*\*/

unsigned int last\_policy; /\* policy before unplug \*/

struct cpufreq\_governor \***governor**; /\*调频策略类型\*/

void \*governor\_data;

bool governor\_enabled; /\* governor start/stop flag \*/

char last\_governor[CPUFREQ\_NAME\_LEN]; /\* last governor used \*/

struct work\_struct update;

/\* if update\_policy() needs to be called, but you're in IRQ context \*/

struct cpufreq\_real\_policy **user\_policy;**

struct cpufreq\_frequency\_table **\*freq\_table**; /\*频率表，指向cpufreq\_frequency\_table数组\*/

struct list\_head  **policy\_list**; /\*将实例添加到全局双链表cpufreq\_policy\_list\*/

**struct kobject kobj;** /\*导出到sysfs的kobject实例\*/

struct completion kobj\_unregister;

struct rw\_semaphore rwsem;

/\* Synchronization for frequency transitions \*/

bool transition\_ongoing; /\* Tracks transition status \*/

spinlock\_t transition\_lock;

**wait\_queue\_head\_t transition\_wait;**

**struct task\_struct \*transition\_task;** /\* Task which is doing the transition \*/

/\* cpufreq-stats \*/

**struct cpufreq\_stats \*stats;**

void \***driver\_data**; /\*cpufreq驱动私有数据\*/

};

cpufreq\_policy结构体主要成员简介如下：

●**cpuinfo：**cpufreq\_cpuinfo结构体实例，表示CPU信息，结构体定义如下：

struct cpufreq\_cpuinfo {

unsigned int max\_freq; /\*最高频率，通常同策略最大频率\*/

unsigned int min\_freq; /\*最低频率，通常同策略最小频率\*/

unsigned int transition\_latency; /\*转换延迟，纳秒\*/

};

●**user\_policy：**cpufreq\_real\_policy结构体实例，结构体定义如下：

struct cpufreq\_real\_policy {

unsigned int min; /\*最小频率，kHz \*/

unsigned int max; /\*最大频率，kHz \*/

unsigned int policy;

struct cpufreq\_governor \*governor; /\*指向频率管理者\*/

};

**●freq\_table：**指向频率列表，指向cpufreq\_frequency\_table结构体数组，结构体定义如下：

struct cpufreq\_frequency\_table {

unsigned int flags; /\*标记， CPUFREQ\_BOOST\_FREQ (1 << 0)\*/

unsigned int driver\_data; /\*驱动私有数据\*/

unsigned int **frequency**; /\*频率值， kHz \*/

};

**●stats：**指向cpufreq\_stats结构体，结构体定义在/drivers/cpufreq/cpufreq\_stats.c文件内：

struct cpufreq\_stats {

unsigned int total\_trans;

unsigned long long last\_time;

unsigned int max\_state;

unsigned int state\_num;

unsigned int last\_index;

u64 \*time\_in\_state;

unsigned int \*freq\_table;

#ifdef CONFIG\_CPU\_FREQ\_STAT\_DETAILS

unsigned int \*trans\_table;

#endif

};

**●policy\_list：**双链表成员，将实例添加到全局双链表cpufreq\_policy\_list。

#### ■创建CPU策略

在注册驱动cpufreq\_driver实例时，将遍历系统内CPU，如果CPU尚未关联到策略，将为其创建策略

cpufreq\_policy实例，创建函数定义如下（/drivers/cpufreq/cpufreq.c）：

static struct cpufreq\_policy \***cpufreq\_policy\_alloc**(struct device \*dev)

/\*dev：表示CPU设备的device实例（内嵌在cpu结构体）\*/

{

struct cpufreq\_policy \*policy;

int ret;

policy = **kzalloc(sizeof(\*policy), GFP\_KERNEL)**; /\*分配实例\*/

...

if (!alloc\_cpumask\_var(&policy->cpus, GFP\_KERNEL)) /\*分配CPU位图\*/

goto err\_free\_policy;

if (!zalloc\_cpumask\_var(&policy->related\_cpus, GFP\_KERNEL))

goto err\_free\_cpumask;

if (!zalloc\_cpumask\_var(&policy->real\_cpus, GFP\_KERNEL))

goto err\_free\_rcpumask;

ret = kobject\_init\_and\_add(&**policy->kobj**, &**ktype\_cpufreq**, &**dev->kobj**, "**cpufreq**");

/\*初始化并添加kobject实例，父节点为表示CPU设备的kobject实例\*/

/\*创建/sys/devices/system/cpu/cpuX/cpufreq/目录，属性为目录下文件\*/

...

INIT\_LIST\_HEAD(&policy->policy\_list);

init\_rwsem(&policy->rwsem);

spin\_lock\_init(&policy->transition\_lock);

init\_waitqueue\_head(&policy->transition\_wait);

init\_completion(&policy->kobj\_unregister);

INIT\_WORK(&policy->update, handle\_update);

**policy->cpu = dev->id**; /\*CPU核编号\*/

policy->kobj\_cpu = dev->id;

return policy;

...

}

cpufreq\_policy实例kobj成员关联的kobj\_type类型为**ktype\_cpufreq，**其下属性（文件）是cpufreq子系统的用户操作接口，后面将详细介绍。

#### ■CPU策略初始化

通用初始化cpufreq\_policy实例的函数为**cpufreq\_generic\_init**()，通常在cpufreq\_driver实例的init()函数内调用，函数定义如下（/drivers/cpufreq/cpufreq.c）：

int **cpufreq\_generic\_init**(struct cpufreq\_policy \*policy,

struct cpufreq\_frequency\_table \***table**,unsigned int transition\_latency)

/\*policy：指向cpufreq\_policy实例，table：指向频率列表（由驱动实现）\*/

{

int ret;

ret = **cpufreq\_table\_validate\_and\_show**(policy, table); /\*/drivers/cpufreq/freq\_table.c\*/

...

policy->cpuinfo.transition\_latency = **transition\_latency**;

cpumask\_setall(policy->cpus);

return 0;

}

cpufreq\_table\_validate\_and\_show()函数定义如下：

int cpufreq\_table\_validate\_and\_show(struct cpufreq\_policy \*policy,struct cpufreq\_frequency\_table \*table)

{

int ret = **cpufreq\_frequency\_table\_cpuinfo**(policy, table);

/\*设置cpuinfo成员，/drivers/cpufreq/freq\_table.c\*/

if (!ret)

**policy->freq\_table = table**; /\*指向频率列表\*/

return ret;

}

#### ■cpufreq通知

内核在/drivers/cpufreq/cpufreq.c文件内定义并初始化了通知链：

static BLOCKING\_NOTIFIER\_HEAD(**cpufreq\_policy\_notifier\_list**);

static struct srcu\_notifier\_head **cpufreq\_transition\_notifier\_list**;

内核其它子系统可调用接口函数**cpufreq\_register\_notifier**(struct notifier\_block \*nb, unsigned int **list**)向通知链注册通知，在CPU策略发生变化时将调用执行通知链中注册的通知。list参数表示将通知注册到上面哪一个通知链中，取值定义如下（/include/linux/cpufreq.h）：

#define CPUFREQ\_TRANSITION\_NOTIFIER (0)

#define CPUFREQ\_POLICY\_NOTIFIER (1)

通知处理函数中action事件参数取值如下：

/\* Transition notifiers \*/

#define CPUFREQ\_PRECHANGE (0)

#define CPUFREQ\_POSTCHANGE (1)

/\* Policy Notifiers \*/

#define CPUFREQ\_ADJUST (0)

#define CPUFREQ\_INCOMPATIBLE (1)

#define CPUFREQ\_NOTIFY (2)

#define CPUFREQ\_START (3)

#define CPUFREQ\_CREATE\_POLICY (4)

#define CPUFREQ\_REMOVE\_POLICY (5)

### 1.3 cpufreq驱动

cpufreq驱动由cpufreq\_driver结构体表示，此结构体中主要包含对底层时钟硬件进行操作的函数指针。平台（处理器）相关代码需要实现此结构体实例，并调用接口函数**cpufreq\_register\_driver()**向内核注册。在注册函数中将向CPU总线注册子系统接口，注册函数中将对总线下每个CPU设备调用子系统接口中的add\_dev()函数为各CPU创建策略cpufreq\_policy实例（尚未关联策略的CPU），并调用cpufreq\_driver实例中的init()函数初始化创建的cpufreq\_policy实例。

#### ■数据结构

cpufreq驱动由cpufreq\_driver结构体表示，定义如下（/include/linux/cpufreq.h）：

struct cpufreq\_driver {

char name[CPUFREQ\_NAME\_LEN]; /\*驱动名称\*/

u8 **flags**; /\*标记\*/

void \*driver\_data;

/\*以下两个函数是必须实现的\*/

int (\***init**)(struct cpufreq\_policy \*policy); /\*初始化函数，必须实现\*/

int (\***verify**)(struct cpufreq\_policy \*policy); /\*对策略进行有效性验证，必须实现\*/

/\*以下三个函数中，最少要实现一个。实现了setpolicy()则不能实现后面两个函数，

\*后面两个函数若实现了其中之一，则不能实现setpolicy()。\*/

int (\*setpolicy)(struct cpufreq\_policy \*policy); /\*设置策略\*/

int (\***target**)(struct cpufreq\_policy \*policy,unsigned int target\_freq,unsigned int relation);

/\*将频率调整到一个指定的值\*/

int (\***target\_index**)(struct cpufreq\_policy \*policy,unsigned int **index**);

/\*按频率列表项值设置时钟频率\*/

/\*以两个函数必须同时实现，或同时不实现\*/

unsigned int (\*get\_intermediate)(struct cpufreq\_policy \*policy,unsigned int index);

int (\*target\_intermediate)(struct cpufreq\_policy \*policy,unsigned int index);

unsigned int (\***get**)(unsigned int cpu); /\*获取CPU当前时钟频率\*/

/\*可选函数\*/

int (\*bios\_limit)(int cpu, unsigned int \*limit);

int (\*exit)(struct cpufreq\_policy \*policy);

void (\*stop\_cpu)(struct cpufreq\_policy \*policy);

int (\*suspend)(struct cpufreq\_policy \*policy);

int (\*resume)(struct cpufreq\_policy \*policy);

/\* Will be called after the driver is fully initialized \*/

void (\*ready)(struct cpufreq\_policy \*policy); /\*策略初始化最后调用此函数\*/

**struct freq\_attr \*\*attr**; /\*属性，添加到策略目录下\*/

/\* platform specific boost support code \*/

bool boost\_supported;

bool boost\_enabled;

int (\*set\_boost)(int state);

};

cpufreq\_driver结构体主要成员简介如下：

**●flags：**标记成员，取值定义如下：

#define CPUFREQ\_STICKY (1 << 0)/\*cpufreq\_driver实例不会被移出，除非所有init()函数失败\*/

#define CPUFREQ\_CONST\_LOOPS (1 << 1) /\*不影响loops\_per\_jiffy等全局变量\*/

#define CPUFREQ\_PM\_NO\_WARN (1 << 2)

#define CPUFREQ\_HAVE\_GOVERNOR\_PER\_POLICY (1 << 3)

#define CPUFREQ\_ASYNC\_NOTIFICATION (1 << 4)

#define CPUFREQ\_NEED\_INITIAL\_FREQ\_CHECK (1 << 5)

/\*检查CPU是否运行在频率列表中某一频率\*/

●**init()：**注册cpufreq\_driver实例函时，在为CPU创建cpufreq\_policy实例后，调用此函数对其进行初始化，设置频率表等。

●**verify()：**用于对用户设置的策略进行有效性验证和数据修正。

●**setpolicy()**：设置CPU策略，实现这个函数，CPU一般具备在一个范围内自动调整频率的能力，大多数CPU还不具备此能力，因此也不实现此函数。

●**target()：**将CPU频率调整到一个指定的值。target\_freq表示目标频率，实际的操作中是将频率设置到最接近target\_freq的值（不一定能准确的设为target\_freq），参数relation用于暗示实际频率值应大于参数target\_freq值，还是小于此值。relation取值定义在/include/linux/cpufreq.h头文件：

#define CPUFREQ\_RELATION\_L 0 /\*实际频率大于等于目标频率\*/

#define CPUFREQ\_RELATION\_H 1 /\*实际频率小于等于目标频率\*/

#define CPUFREQ\_RELATION\_C 2 /\*实际频率最接近目标频率\*/

●**target\_index()：**设置频率值，参数index表示频率列表中的索引值，即按频率列表中某个频率设置CPU频率。策略的频率列表在cpufreq\_driver->init()函数中设置。

●**get()：**获取指定CPU的当前频率。

●**attr：**指向freq\_attr结构体指针数组，此属性将添加到CPU策略目录下，作为策略的属性文件。

内核定义了全局指针变量cpufreq\_driver用于指向当前注册的cpufreq\_driver实例，系统内某一时刻只能有一个注册的cpufreq\_driver实例，若有多个cpufreq\_driver实例，在注册本实例前应注销前一个注册的实例。

#### ■注册函数

注册cpufreq\_driver实例的接口函数**cpufreq\_register\_driver()**定义如下（/drivers/cpufreq/cpufreq.c）：

int cpufreq\_register\_driver(struct cpufreq\_driver \*driver\_data)

{

unsigned long flags;

int ret;

if (cpufreq\_disabled()) /\*直接返回false\*/

return -ENODEV;

if (!driver\_data || !**driver\_data->verify** || !**driver\_data->init** ||

!(driver\_data->setpolicy || driver\_data->target\_index ||driver\_data->target) ||

(driver\_data->setpolicy && (driver\_data->target\_index ||driver\_data->target)) ||

(!!driver\_data->get\_intermediate != !!driver\_data->target\_intermediate))

return -EINVAL; /\*有效性判断\*/

...

write\_lock\_irqsave(&cpufreq\_driver\_lock, flags);

if (**cpufreq\_driver**) { /\*如果全局变量不为空，返回错误码\*/

write\_unlock\_irqrestore(&cpufreq\_driver\_lock, flags);

return -EEXIST;

}

**cpufreq\_driver** = **driver\_data**; /\*全局指针cpufreq\_driver指向注册实例\*/

write\_unlock\_irqrestore(&cpufreq\_driver\_lock, flags);

if (driver\_data->setpolicy)

driver\_data->flags |= **CPUFREQ\_CONST\_LOOPS**; /\*设置驱动标记位\*/

if (cpufreq\_boost\_supported()) { /\*返回cpufreq\_driver->boost\_supported\*/

if (!cpufreq\_driver->set\_boost)

cpufreq\_driver->set\_boost = cpufreq\_boost\_set\_sw;

ret = cpufreq\_sysfs\_create\_file(&boost.attr);

/\*在/sys/devices/system/cpu/cpufreq/目录下添加属性文件\*/

...

}

ret = **subsys\_interface\_register(&cpufreq\_interface)**;

/\*注册子系统接口（CPU总线），对每个CPU调用cpufreq\_interface.**add\_dev()**函数\*/

...

if (!(cpufreq\_driver->flags & CPUFREQ\_STICKY) &&list\_empty(&cpufreq\_policy\_list)) {

... /\*错误处理\*/

}

**register\_hotcpu\_notifier(&cpufreq\_cpu\_notifier)**; /\*注册通知到CPU热插拔通知链\*/

...

return 0;

...

}

注册cpufreq驱动函数首先检查cpufreq\_driver实例的有效性，然后对CPU总线（管理CPU设备）注册子系统接口**cpufreq\_interface**，子系统接口将为CPU创建并初始化策略（见下文），最后在CPU热插拔通知链中注册cpufreq\_cpu\_notifier通知。

下面先介绍CPU热插拔通知cpufreq\_cpu\_notifier中所做的工作，通知实例定义如下：

static struct notifier\_block \_\_refdata cpufreq\_cpu\_notifier = {

.notifier\_call = **cpufreq\_cpu\_callback**, /\*回调函数\*/

};

CPU热插拔时的回调函数cpufreq\_cpu\_callback()定义如下：

static int cpufreq\_cpu\_callback(struct notifier\_block \*nfb,unsigned long action, void \*hcpu)

{

unsigned int cpu = (unsigned long)hcpu;

struct device \*dev;

dev = **get\_cpu\_device(cpu)**; /\*表示CPU设备的device实例\*/

if (dev) {

switch (action & ~CPU\_TASKS\_FROZEN) {

case CPU\_ONLINE:

**cpufreq\_add\_dev(dev, NULL)**; /\*插入CPU，创建并初始化策略，见下文\*/

break;

case CPU\_DOWN\_PREPARE:

\_\_cpufreq\_remove\_dev\_prepare(dev);

break;

case CPU\_POST\_DEAD:

\_\_cpufreq\_remove\_dev\_finish(dev);

break;

case CPU\_DOWN\_FAILED:

cpufreq\_add\_dev(dev, NULL);

break;

}

}

return NOTIFY\_OK;

}

注销cpufreq\_driver实例的函数为cpufreq\_unregister\_driver(struct cpufreq\_driver \*driver)，源代码请读者自行阅读（cpufreq\_driver = NULL等）。

#### ■设置频率

策略类型cpufreq\_governor中的governor()函数在需要设置时钟频率时将调用**\_\_cpufreq\_driver\_target()**接口函数设置时钟频率，函数定义如下（/drivers/cpufreq/cpufreq.c）：

int \_\_cpufreq\_driver\_target(struct cpufreq\_policy \*policy,unsigned int target\_freq,unsigned int relation)

/\*target\_freq：目标频率，relation：实际频率与目标频率关系\*/

{

unsigned int old\_target\_freq = target\_freq;

int retval = -EINVAL;

if (cpufreq\_disabled())

return -ENODEV;

/\*确保目标频率有效\*/

if (target\_freq > policy->max)

target\_freq = policy->max;

if (target\_freq < policy->min)

target\_freq = policy->min;

...

if (target\_freq == policy->cur) /\*不需要修改频率\*/

return 0;

/\*频率需要修改，保存当前频率值\*/

policy->restore\_freq = policy->cur;

if (cpufreq\_driver->target)

retval = **cpufreq\_driver->target(policy, target\_freq, relation)**; /\*调整频率\*/

else if (cpufreq\_driver->target\_index) { /\*按频率列表值设置频率\*/

struct cpufreq\_frequency\_table \*freq\_table;

int index;

freq\_table = cpufreq\_frequency\_get\_table(policy->cpu); /\*获取频率列表\*/

...

retval = cpufreq\_frequency\_table\_target(policy, freq\_table,target\_freq, relation, **&index**);

/\*确定频率列表项索引值\*/

...

if (freq\_table[index].frequency == policy->cur) {

retval = 0;

goto out;

}

retval = **\_\_target\_index(policy, freq\_table, index)**;

/\*调用cpufreq\_driver->target\_index(policy, index)函数设置频率\*/

}

out:

return retval;

}

\_\_cpufreq\_driver\_target()函数中调用cpufreq\_driver实例中的target()或target\_index()函数设置时钟频率。对于设置了setpolicy()函数的cpufreq\_driver实例并不需要调用\_\_cpufreq\_driver\_target()函数设置具体的频率，只需要直接调用cpufreq\_driver->setpolicy()函数即可让驱动自动调整时钟频率。

### 1.4添加策略

内核在/drivers/base/cpu.c文件内定义了CPU设备总线cpu\_subsys，在初始化函数cpu\_dev\_init()内将注册cpu\_subsys总线（注册子系统），若选择了GENERIC\_CPU\_DEVICES配置选项，CPU将当作设备挂载到此总线上，由结构体cpu表示（内嵌device结构体实例）。

前面介绍的cpufreq\_register\_driver()函数中注册的子系统接口实例cpufreq\_interface定义如下：

static struct subsys\_interface cpufreq\_interface = { /\*/drivers/cpufreq/cpufreq.c\*/

.name = "cpufreq",

.subsys = &**cpu\_subsys**, /\*CPU总线\*/

.add\_dev = **cpufreq\_add\_dev**, /\*注册子系统接口时对每个CPU设备调用此函数\*/

.remove\_dev = cpufreq\_remove\_dev, /\*注销子系统接口时对每个CPU设备调用此函数\*/

};

在注册子系统接口实例时，将会遍历CPU总线下的CPU设备，对每个设备调用add\_dev()函数，如下图所示：



cpufreq\_interface实例中add\_dev()函数为**cpufreq\_add\_dev()**函数，函数执行结果如下图所示：



cpufreq\_add\_dev()函数为CPU创建cpufreq\_policy实例，调用驱动的初始化函数init()初始化实例（为其创建频率列表等），调用cpufreq\_init\_policy(policy)函数对实例执行通用的初始化设置（关联策略类型等）。

#### ■添加函数

cpufreq\_add\_dev()函数代码如下（/drivers/cpufreq/cpufreq.c）：

static int cpufreq\_add\_dev(struct device \*dev, struct subsys\_interface \*sif)

{

unsigned int j, cpu = dev->id;

int ret = -ENOMEM;

struct cpufreq\_policy \*policy; /\*策略\*/

unsigned long flags;

bool recover\_policy = !sif; /\*false\*/

...

if (cpu\_is\_offline(cpu)) { /\*不在线的CPU\*/

policy = per\_cpu(cpufreq\_cpu\_data, cpu);

return policy && !cpumask\_test\_and\_set\_cpu(cpu, policy->real\_cpus)?

add\_cpu\_dev\_symlink(policy, cpu) : 0;

}

if (!down\_read\_trylock(&cpufreq\_rwsem))

return 0;

/\*检查CPU是否有关联的策略\*/

policy = per\_cpu(cpufreq\_cpu\_data, cpu);

if (policy && !policy\_is\_inactive(policy)) { /\*如果policy存在且活跃\*/

WARN\_ON(!cpumask\_test\_cpu(cpu, policy->related\_cpus));

ret = **cpufreq\_add\_policy\_cpu(policy, cpu, dev)**; /\*设置策略CPU位图\*/

up\_read(&cpufreq\_rwsem);

return ret;

}

policy = recover\_policy ? cpufreq\_policy\_restore(cpu) : NULL;

if (!policy) { /\*policy不存在\*/

recover\_policy = false;

policy = **cpufreq\_policy\_alloc(dev)**; /\*分配策略，见前文\*/

...

}

cpumask\_copy(policy->cpus, cpumask\_of(cpu)); /\*设置位图中CPU标记位\*/

ret = **cpufreq\_driver->init(policy)**; /\*调用驱动定义的初始化函数，初始化策略\*/

...

down\_write(&policy->rwsem);

cpumask\_or(policy->related\_cpus, policy->related\_cpus, policy->cpus);

if (!recover\_policy)

cpumask\_and(policy->real\_cpus, policy->cpus, cpu\_present\_mask);

cpumask\_and(policy->cpus, policy->cpus, cpu\_online\_mask);

if (!recover\_policy) {

policy->user\_policy.min = policy->min; /\*策略限制的最小、最大频率\*/

policy->user\_policy.max = policy->max;

write\_lock\_irqsave(&cpufreq\_driver\_lock, flags);

for\_each\_cpu(j, policy->related\_cpus) /\*扫描相关的CPU\*/

**per\_cpu(cpufreq\_cpu\_data, j) = policy**; /\*CPU关联到本策略\*/

write\_unlock\_irqrestore(&cpufreq\_driver\_lock, flags);

}

if (cpufreq\_driver->get && !cpufreq\_driver->setpolicy) {

policy->cur = cpufreq\_driver->get(policy->cpu); /\*当前CPU频率\*/

...

}

/\*如果驱动要求CPU运行在频率列表中的频率，则检查当前频率有效性，必要时重置\*/

if ((cpufreq\_driver->flags & CPUFREQ\_NEED\_INITIAL\_FREQ\_CHECK)&& has\_target()) {

ret = **cpufreq\_frequency\_table\_get\_index**(policy, policy->cur);

/\*当频率对应的列表项索引值\*/

if (ret == -EINVAL) {

...

ret = \_\_cpufreq\_driver\_target(policy, policy->cur - 1,CPUFREQ\_RELATION\_L);

...

}

}

blocking\_notifier\_call\_chain(&cpufreq\_policy\_notifier\_list, CPUFREQ\_START, policy);

/\*执行cpufreq\_policy\_notifier\_list通知链中通知\*/

if (!recover\_policy) {

ret = **cpufreq\_add\_dev\_interface(policy, dev)**;

/\*在policy目录下创建属性文件（驱动中所含属性等）\*/

if (ret)

goto err\_out\_unregister;

blocking\_notifier\_call\_chain(&cpufreq\_policy\_notifier\_list,

CPUFREQ\_CREATE\_POLICY, policy);

write\_lock\_irqsave(&cpufreq\_driver\_lock, flags);

**list\_add(&policy->policy\_list, &cpufreq\_policy\_list)**; /\*将实例添加到全局双链表头部\*/

write\_unlock\_irqrestore(&cpufreq\_driver\_lock, flags);

}

**cpufreq\_init\_policy(policy)**; /\*策略实例通用初始化，关联策略类型等\*/

if (!recover\_policy) {

policy->user\_policy.policy = policy->policy;

policy->user\_policy.governor = policy->governor;

}

up\_write(&policy->rwsem);

**kobject\_uevent(&policy->kobj, KOBJ\_ADD)**;

up\_read(&cpufreq\_rwsem);

if (cpufreq\_driver->ready)

**cpufreq\_driver->ready(policy)**; /\*最后调用驱动ready()函数\*/

...

return 0; /\*成功返回0\*/

...

}

#### ■设置策略

下面主要看一下通用初始化函数**cpufreq\_init\_policy()**的实现，代码如下（/drivers/cpufreq/cpufreq.c）：

static void cpufreq\_init\_policy(struct cpufreq\_policy \*policy)

{

struct cpufreq\_governor \*gov = NULL;

struct cpufreq\_policy new\_policy; /\*cpufreq\_policy实例\*/

int ret = 0;

**memcpy(&new\_policy, policy, sizeof(\*policy))**; /\*复制原实例到新实例\*/

gov = **find\_governor**(**policy->last\_governor**); /\*/drivers/cpufreq/cpufreq.c\*/

/\*由policy->last\_governor指示的名称查找原关联的cpufreq\_governor实例\*/

if (gov)

pr\_debug("Restoring governor %s for cpu %d\n",policy->governor->name, policy->cpu);

else

gov = CPUFREQ\_DEFAULT\_GOVERNOR; /\*由配置选项确定默认的gov\*/

new\_policy.governor = gov; /\*设置策略类型实例\*/

if (**cpufreq\_driver->setpolicy**) /\*需要重新设置策略类型\*/

cpufreq\_parse\_governor(gov->name, &new\_policy.policy, NULL); /\*设置关联策略类型\*/

ret = **cpufreq\_set\_policy(policy, &new\_policy)**; /\*设置策略\*/

...

}

**cpufreq\_set\_policy()**函数用新cpufreq\_policy实例设置原实例，代码如下（/drivers/cpufreq/cpufreq.c）：

static int cpufreq\_set\_policy(struct cpufreq\_policy \*policy,struct cpufreq\_policy \*new\_policy)

{

struct cpufreq\_governor \*old\_gov;

int ret;

...

memcpy(&new\_policy->cpuinfo, &policy->cpuinfo, sizeof(policy->cpuinfo));

if (new\_policy->min > policy->max || new\_policy->max < policy->min)

return -EINVAL; /\*频率范围需有重叠\*/

ret = **cpufreq\_driver->verify(new\_policy)**; /\*检查新实例有效性，频率范围必须有效\*/

if (ret)

return ret;

/\*执行通知链通知\*/

blocking\_notifier\_call\_chain(&cpufreq\_policy\_notifier\_list,CPUFREQ\_ADJUST, new\_policy);

blocking\_notifier\_call\_chain(&cpufreq\_policy\_notifier\_list,CPUFREQ\_INCOMPATIBLE,

new\_policy);

ret = cpufreq\_driver->verify(new\_policy);

if (ret)

return ret;

/\* notification of the new policy \*/

blocking\_notifier\_call\_chain(&cpufreq\_policy\_notifier\_list,CPUFREQ\_NOTIFY, new\_policy);

policy->min = new\_policy->min; /\*频率范围\*/

policy->max = new\_policy->max;

...

if (**cpufreq\_driver->setpolicy**) { /\*自动调频，调用setpolicy()函数后即可返回\*/

policy->policy = new\_policy->policy;

pr\_debug("setting range\n");

return **cpufreq\_driver->setpolicy(new\_policy)**;

}

if (new\_policy->governor == policy->governor) /\*没有改变策略类型，函数返回\*/

goto out;

...

old\_gov = policy->governor;

if (old\_gov) {

**\_\_cpufreq\_governor**(policy, CPUFREQ\_GOV\_STOP); /\*结束与旧策略类型的关联\*/

up\_write(&policy->rwsem);

**\_\_cpufreq\_governor**(policy, CPUFREQ\_GOV\_POLICY\_EXIT);

down\_write(&policy->rwsem);

}

/\*启用新策略类型\*/

policy->governor = new\_policy->governor;

if (!**\_\_cpufreq\_governor(policy, CPUFREQ\_GOV\_POLICY\_INIT)**) { /\*初始化\*/

if (!\_\_cpufreq\_governor(policy, **CPUFREQ\_GOV\_START)**) /\*启用\*/

goto out;

up\_write(&policy->rwsem);

\_\_cpufreq\_governor(policy, CPUFREQ\_GOV\_POLICY\_EXIT);

down\_write(&policy->rwsem);

}

/\* new governor failed, so re-start old one \*/

...

out:

pr\_debug("governor: change or update limits\n");

return **\_\_cpufreq\_governor**(policy, CPUFREQ\_GOV\_LIMITS);

}

cpufreq\_set\_policy()函数先检查策略的有效性，然后结束策略实例与旧策略类型之间的关联（调用旧策略类型中的governor()函数），最后启用策略实例与新策略类型之间的关联。

**\_\_cpufreq\_governor()**接口函数用于CPU策略与关联策略类型之有事件发生时调用，例如断开关联、建立关联等，函数定义如下（/drivers/cpufreq/cpufreq.c）：

static int \_\_cpufreq\_governor(struct cpufreq\_policy \*policy,unsigned int event)

{

int ret;

#ifdef CONFIG\_CPU\_FREQ\_GOV\_PERFORMANCE

struct cpufreq\_governor \*gov = &cpufreq\_gov\_performance; /\*最高频率运行，性能模式\*/

#else

struct cpufreq\_governor \*gov = NULL;

#endif

if (cpufreq\_suspended)

return 0;

if (!policy->governor)

return -EINVAL;

if (policy->governor->max\_transition\_latency &&policy->cpuinfo.transition\_latency >

policy->governor->max\_transition\_latency) {

if (!gov)

return -EINVAL;

else {

pr\_warn("%s governor failed, too long transition latency of HW, fallback to %s governor\n",

policy->governor->name, gov->name);

policy->governor = gov;

}

}

if (event == CPUFREQ\_GOV\_POLICY\_INIT)

if (!try\_module\_get(policy->governor->owner))

return -EINVAL;

...

mutex\_lock(&cpufreq\_governor\_lock);

...

if (event == CPUFREQ\_GOV\_STOP)

policy->governor\_enabled = false;

else if (event == CPUFREQ\_GOV\_START)

policy->governor\_enabled = true;

mutex\_unlock(&cpufreq\_governor\_lock);

ret = **policy->governor->governor(policy, event)**; /\*成功返回0\*/

if (!ret) {

if (event == CPUFREQ\_GOV\_POLICY\_INIT)

policy->governor->initialized++;

else if (event == CPUFREQ\_GOV\_POLICY\_EXIT)

policy->governor->initialized--;

} else {

...

}

...

return ret;

}

\_\_cpufreq\_governor()函数内主要是调用policy->governor->governor(policy, event)函数完成需要完成的工作，如果需要设置时钟频率将调用前面介绍的\_\_cpufreq\_driver\_target()接口函数设置硬件时钟频率。

### 1.5用户接口

现在我们来讨论一下cpufreq子系统向用户进程提供的接口。cpufreq子系统通过CPU策略实例下的属性文件向用户导出操作接口（sysfs文件系统）。

在分配cpufreq\_policy实例时，其kobj成员的kobj\_type类型为**ktype\_cpufreq**，定义如下：

static struct kobj\_type ktype\_cpufreq = { /\*/drivers/cpufreq/cpufreq.c\*/

.sysfs\_ops = &**sysfs\_ops**, /\*/drivers/cpufreq/cpufreq.c\*/

.default\_attrs = **default\_attrs**, /\*默认属性\*/

.release = cpufreq\_sysfs\_release,

};

#### ■策略属性

cpufreq\_policy实例中kobject成员下的属性由freq\_attr结构体表示，定义如下（/include/linux/cpufreq.h）：

struct freq\_attr {

**struct attribute attr**;

ssize\_t (\*show)(struct cpufreq\_policy \*, char \*); /\*显示属性值\*/

ssize\_t (\*store)(struct cpufreq\_policy \*, const char \*, size\_t count); /\*设置属性值\*/

};

ktype\_cpufreq实例关联的sysfs\_ops实例中的show()和store()函数调用freq\_attr结构体中的show()和store()函数。

在同一头文件内实现了定义属性的宏，例如：

#define **cpufreq\_freq\_attr\_ro(\_name)** \ /\*只读属性\*/

static struct freq\_attr \_name = \

\_\_ATTR(\_name, 0444, show\_##\_name, NULL)

#define **cpufreq\_freq\_attr\_rw(\_name)** \ /\*读写属性\*/

static struct freq\_attr \_name = \

\_\_ATTR(\_name, 0644, show\_##\_name, store\_##\_name)

ktype\_cpufreq实例中的默认属性default\_attrs定义如下（/drivers/cpufreq/cpufreq.c）：

static struct attribute \*default\_attrs[] = {

&cpuinfo\_min\_freq.attr, /\*显示最小可设置频率\*/

&cpuinfo\_max\_freq.attr, /\*显示最大可设置频率\*/

&cpuinfo\_transition\_latency.attr,

&scaling\_min\_freq.attr, /\*显示当前策略限制的最小频率\*/

&scaling\_max\_freq.attr, /\*显示当前策略限制的最大频率\*/

&affected\_cpus.attr, /\*显示在线CPU\*/

&related\_cpus.attr,

&**scaling\_governor**.attr, /\*获取/设置CPU策略类型\*/

&scaling\_driver.attr,

&scaling\_available\_governors.attr,

&**scaling\_setspeed**.attr, /\*设置频率\*/

NULL

};

以上列出的每个属性对应**/sys/devices/system/cpu/cpuX/cpufreq/**目录下的一个文件，用户进程可通过对文件的读写获取/设置策略属性文件对CPU频率进行控制。

下面以设置CPU频率属性的读写函数为例说明用户操作接口的实现，其它属性的定义及读写函数的实现请读者自行阅读源代码。

#### ■策略类型属性

scaling\_governor属性用于获取/设置CPU策略类型，属性读写函数如下（/drivers/cpufreq/cpufreq.c）：

static ssize\_t show\_scaling\_governor(struct cpufreq\_policy \*policy, char \*buf) /\*获取策略类型\*/

{

if (policy->policy == CPUFREQ\_POLICY\_POWERSAVE)

return sprintf(buf, "powersave\n");

else if (policy->policy == CPUFREQ\_POLICY\_PERFORMANCE)

return sprintf(buf, "performance\n");

else if (policy->governor)

return **scnprintf(buf, CPUFREQ\_NAME\_PLEN, "%s\n",policy->governor->name)**;

return -EINVAL;

}

static ssize\_t store\_scaling\_governor(struct cpufreq\_policy \*policy,const char \*buf, size\_t count)

{ /\*设置策略类型，输入为策略类型名称字符串\*/

int ret;

char str\_governor[16];

struct cpufreq\_policy new\_policy;

ret = **cpufreq\_get\_policy(&new\_policy, policy->cpu)**;

if (ret)

return ret;

ret = **sscanf(buf, "%15s", str\_governor)**;

if (ret != 1)

return -EINVAL;

if (**cpufreq\_parse\_governor**(str\_governor, &new\_policy.policy,&new\_policy.governor))

return -EINVAL;

ret = **cpufreq\_set\_policy(policy, &new\_policy)**; /\*设置策略\*/

policy->user\_policy.policy = policy->policy;

policy->user\_policy.governor = policy->governor;

if (ret)

return ret;

else

return count;

}

例如对CPU0采用userspace策略类型，可执行下以下命令：

#echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/**scaling\_governor** /\*写属性文件\*/

#### ■频率属性

如果用户对CPU采用userspace策略类型，还可以通过写**scaling\_setspeed**属性，对CPU设置具体的频率。**scaling\_setspeed**属性读写函数定义如下（/drivers/cpufreq/cpufreq.c）：

static ssize\_t show\_scaling\_setspeed(struct cpufreq\_policy \*policy, char \*buf) /\*读取频率值\*/

{

if (!policy->governor || !policy->governor->show\_setspeed)

return sprintf(buf, "<unsupported>\n");

return **policy->governor->show\_setspeed(policy, buf)**;

}

static ssize\_t store\_scaling\_setspeed(struct cpufreq\_policy \*policy,const char \*buf, size\_t count)

{

unsigned int freq = 0;

unsigned int ret;

if (!policy->governor || !policy->governor->store\_setspeed)

return -EINVAL;

ret = sscanf(buf, "%u", &freq);

if (ret != 1)

return -EINVAL;

**policy->governor->store\_setspeed(policy, freq)**;

return count;

}

userspace策略类型show\_setspeed()函数通过policy->cur获取当前频率值，store\_setspeed()函数调用接口函数\_\_cpufreq\_driver\_target()设置频率值。

假设CPU0策略已经设置了采用userspace策略类型，可通过以下命令设置CPU0频率：

#echo 700000 > /sys/devices/system/cpu/cpu0/cpufreq/**scaling\_setspeed** /\*写属性文件，设为700MHz\*/

### 1.6驱动示例

下面以龙芯1B处理器cpufreq驱动为例，说明驱动程序的实现。

在平台相关的/arch/mips/loongson32/common/platform.c文件内实现了plat\_ls1x\_cpufreq结构体实例，它用于向驱动传递处理器的时钟信息（作为platform\_device实例的平台相关数据结构），文件内还实现了名称为"ls1x-cpufreq"的platfrom\_device实例，并在初始化时注册到内核。

static struct plat\_ls1x\_cpufreq ls1x\_cpufreq\_pdata = {

.clk\_name = "cpu\_clk",

.osc\_clk\_name = "osc\_33m\_clk",

.max\_freq = 266 \* 1000,

.min\_freq = 33 \* 1000,

}; /\*结构体定义位于/arch/mips/include/asm/mach-loongson32/cpufreq.h\*/

struct platform\_device ls1x\_cpufreq\_pdev = {

.name = **"ls1x-cpufreq"**,

.dev = {

.platform\_data = **&ls1x\_cpufreq\_pdata**,

},

};

龙芯1B处理器cpufreq驱动代码位于/drivers/cpufreq/ls1x-cpufreq.c，文件内实现了platform\_driver实例，并向内核注册。platform\_driver的探测函数probe()将调用cpufreq\_register\_driver()函数注册文件内实现的cpufreq\_driver实例，如下图所示。



cpufreq\_driver实例定义如下：

static struct cpufreq\_driver ls1x\_cpufreq\_driver = {

.name = "cpufreq-ls1x",

.flags = CPUFREQ\_STICKY | CPUFREQ\_NEED\_INITIAL\_FREQ\_CHECK,

.verify = cpufreq\_generic\_frequency\_table\_verify,

.target\_index = **ls1x\_cpufreq\_target**, /\*按频率表设置CPU频率\*/

.get = cpufreq\_generic\_get,

.init = **ls1x\_cpufreq\_init**, /\*创建频率表，初始化cpufreq\_policy实例\*/

.exit = ls1x\_cpufreq\_exit,

.attr = **cpufreq\_generic\_attr**, /\*默认属性，添加到策略目录下，/drivers/cpufreq/freq\_table.c\*/

};

platform\_driver实例定义如下：

static struct platform\_driver ls1x\_cpufreq\_platdrv = {

.driver = {

.name = **"ls1x-cpufreq"**,

},

.probe = **ls1x\_cpufreq\_probe**,

.remove = ls1x\_cpufreq\_remove,

};

在探测函数ls1x\_cpufreq\_probe()中将调用cpufreq\_register\_driver()函数注册定义的ls1x\_cpufreq\_driver实例，在实例的init()函数ls1x\_cpufreq\_init()中将创建频率列表，并初始化cpufreq\_policy实例。处理器相关的函数中通过前一小节介绍的CCF设置硬件时钟，源代码请读者自行阅读。

MIPS体系结构需要选择配置选项MIPS\_EXTERNAL\_TIMER才会编译cpufreq子系统，详见配置文件/arch/mips/Kconfig。

## 2 cpuidle子系统

目前，SoC大多支持几个不同的idle级别，SoC在不同的idle级别具有不同的状态，不同的状态具有不同的功耗状态。这里的idle级别是硬件级别，不是软件级别。cpuidle子系统的目的就在于管理SoC的idle状态，以实现节能的目的。

### 2.1概述

cpuidle驱动与cpufreq驱动类似，底层驱动需要提供一个idle级别列表，列表项中要实现进入和退出各状态的操作函数。cpuidle策略主要提供选择idle级别的策略。每个CPU核对应一个cpuidle\_device实例，用于管理CPU核的idle级别信息，如下图所示。



cpuidle子系统对外主要提供两个接口：

●内核接口：提供选择CPU核idle级别和进入退出idle级别的函数等，主要用于内核idle线程。

●用户接口：通过sysfs文件系统中的属性文件导出cpuidle子系统信息，主要用于显示信息。

### 2.2 cpuidle策略

cpuidle子系统定义了策略类型，用于表示按何种策略（步骤）改变CPU的idle级别。策略类型由结构体cpuidle\_governor表示，cpuidle子系统默认实现并注册了"ladder"策略和"menu"策略类型。在首次注册策略类型，或注册的策略类型等级高于当前使用的策略类型时，将会把注册的策略类型设置为当前使用的策略类型，并使能相关CPU采用新的策略类型。策略类型相关代码在/drivers/cpuidle/governor.c文件内实现。

#### ■数据结构

cpuidle策略类型由cpuidle\_governor结构体表示，定义如下（/include/linux/cpuidle.h）：

struct cpuidle\_governor {

char name[CPUIDLE\_NAME\_LEN]; /\*名称，用于标识策略类型\*/

struct list\_head governor\_list; /\*双链表成员，将实例添加到全局双链表cpuidle\_governors\*/

unsigned int **rating**; /\*策略类型等级\*/

int (\***enable**) (struct cpuidle\_driver \*drv,struct cpuidle\_device \*dev);

void (\***disable**) (struct cpuidle\_driver \*drv,struct cpuidle\_device \*dev);

int (\***select**) (struct cpuidle\_driver \*drv,struct cpuidle\_device \*dev); /\*选择idle级别\*/

void (\***reflect**) (struct cpuidle\_device \*dev, int index); /\*用于内核接口函数\*/

struct module \*owner;

};

cpuidle\_governor结构体主要成员简介如下：

●**name[]**：策略类型名称，用于标识实例。

●**governor\_list**：双链表成员，将实例添加到全局双链表cpuidle\_governors，内核定义了全局双链表cpuidle\_governors用于管理注册的cpuidle\_governor实例。

●**rating：**策略类型等级。

●**enable()：**CPU使能该策略类型。

●**disable()：**CPU禁用该策略类型。

●**select()：**确定要进入的idle级别，用于内核接口函数。

cpuidle子系统默认实现了"ladder"策略和"menu"策略（并注册），相关代码位于/drivers/cpuidle/governors/目录下的laddr.c和menu.c文件内，源代码请读者自行阅读。

"ladder"策略在进入和退出idle级别时是步进的，它以过去的idle时间作为参考，而"menu"策略总是根据预期的空闲时间直接进入目标idle级别。前者适用于没有采用动态时钟的系统（没有选择NO\_HZ选项），不依赖NO\_HZ选项，后者适用于动态时钟系统。

#### ■注册策略

注册cpuidle\_governor实例的接口函数**cpuidle\_register\_governor()**定义在/drivers/cpuidle/governor.c文件内：

int cpuidle\_register\_governor(struct cpuidle\_governor \*gov)

{

int ret = -EEXIST;

if (!gov || !gov->select)

return -EINVAL;

if (cpuidle\_disabled())

return -ENODEV;

mutex\_lock(&cpuidle\_lock);

if (**\_\_cpuidle\_find\_governor**(gov->name) == NULL) { /\*在全局双链表cpuidle\_governors中查找\*/

ret = 0;

**list\_add\_tail**(&gov->governor\_list, &cpuidle\_governors); /\*将实例插入全局链表末尾\*/

if (!cpuidle\_curr\_governor ||cpuidle\_curr\_governor->rating < gov->rating)

**cpuidle\_switch\_governor(gov)**; /\*切换cpuidle\_governor实例\*/

}

mutex\_unlock(&cpuidle\_lock);

return ret;

}

全局变量cpuidle\_curr\_governor指向当前使用的策略类型，注册cpuidle\_governor实例的函数中，先在全局双链表cpuidle\_governors中查找是否存在同名的实例，存在则返回。不存在则将实例添加到全局双链表，并判断cpuidle\_curr\_governor是否为空，如果为空或者注册实例等级大于当前使用实例，则执行切换当前使用策略类型的操作。

切换当前使用cpuidle\_governor实例的函数定义如下（/drivers/cpuidle/governor.c）：

int **cpuidle\_switch\_governor**(struct cpuidle\_governor \*gov)

{

struct cpuidle\_device \*dev;

if (gov == cpuidle\_curr\_governor)

return 0;

cpuidle\_uninstall\_idle\_handler();

if (cpuidle\_curr\_governor) {

list\_for\_each\_entry(dev, &cpuidle\_detected\_devices, device\_list) /\*遍历cpuidle\_device实例\*/

**cpuidle\_disable\_device(dev)**; /\*CPU禁用当前策略类型\*/

/\*调用cpuidle\_curr\_governor->disable(drv, dev)函数，将设备属性移出sysfs文件系统等\*/

module\_put(cpuidle\_curr\_governor->owner);

}

**cpuidle\_curr\_governor = gov**; /\*设置当前使用cpuidle策略类型\*/

if (gov) {

if (!try\_module\_get(cpuidle\_curr\_governor->owner))

return -EINVAL;

list\_for\_each\_entry(dev, &cpuidle\_detected\_devices, device\_list)

**cpuidle\_enable\_device(dev)**; /\*使能CPU使用新策略类型\*/

/\*调用cpuidle\_curr\_governor->enable(drv, dev)函数，创建目录和属性文件等\*/

cpuidle\_install\_idle\_handler();

printk(KERN\_INFO "cpuidle: using governor %s\n", gov->name);

}

return 0;

}

切换函数中将对CPU核禁用当前策略类型，设置新实例为当前使用策略类型，最后对每个CPU核使能新的策略类型，使能函数定义如下（/drivers/cpuidle/cpuidle.c）：

int cpuidle\_enable\_device(struct cpuidle\_device \*dev)

{

int ret;

struct cpuidle\_driver \*drv;

...

if (dev->enabled)

return 0;

**drv** = **cpuidle\_get\_cpu\_driver(dev)**; /\*获取当前cpuidle\_driver实例\*/

...

ret = **cpuidle\_add\_device\_sysfs(dev)**; /\*向CPU添加属性，见下文，/drivers/cpuidle/sysfs.c\*/

...

if (cpuidle\_curr\_governor->enable &&(ret = **cpuidle\_curr\_governor->enable(drv, dev)**))

goto fail\_sysfs; /\*调用策略类型的enable()函数\*/

smp\_wmb();

dev->enabled = 1;

enabled\_devices++;

return 0;

...

}

### 2.3 cpuidle驱动

SoC平台相关代码需要定义cpuidle驱动cpuidle\_driver实例，并向cpuidle子系统注册。cpuidle\_driver实例最主要的工作就是要实现idle级别列表，列表项中包含进入和退出该级别的函数指针。

cpuidle驱动相关代码在/drivers/cpuidle/driver.c文件内实现。

#### ■数据结构

cpuidle驱动由cpuidle\_driver结构体表示，定义如下（/include/linux/cpuidle.h）：

struct cpuidle\_driver {

const char \*name; /\*名称\*/

struct module \*owner;

int refcnt; /\*引用计数\*/

unsigned int bctimer:1;

**struct cpuidle\_state states[CPUIDLE\_STATE\_MAX]**; /\*idle级别列表，按功耗递减排列\*/

int **state\_count**; /\*支持idle级别数量，前一成员有效数组项数\*/

int safe\_state\_index;

struct cpumask \***cpumask**; /\*驱动适用的CPU核位图\*/

};

cpuidle\_driver结构体中最主要的成员是cpuidle\_state结构体数组，cpuidle\_state结构体表示idle级别，主要包含进入和退出该级别的函数指针。cpuidle\_state结构体数组中各等级需按功耗递减的顺序排列。

cpuidle\_state结构体定义如下（/include/linux/cpuidle.h）：

struct cpuidle\_state {

char name[CPUIDLE\_NAME\_LEN];

char desc[CPUIDLE\_DESC\_LEN];

unsigned int flags;

unsigned int exit\_latency; /\* in US \*/

int power\_usage; /\* in mW \*/

unsigned int target\_residency; /\* in US \*/

bool disabled; /\* disabled on all CPUs \*/

int (\***enter**) (struct cpuidle\_device \*dev,struct cpuidle\_driver \*drv,int index);

int (\*enter\_dead) (struct cpuidle\_device \*dev, int index);

void (\*enter\_freeze) (struct cpuidle\_device \*dev,struct cpuidle\_driver \*drv,int index);

};

#### ■注册驱动

SoC相关代码需要实现cpuidle\_driver实例，如果选择了CPU\_IDLE\_MULTIPLE\_DRIVERS配置选项，则表示注册了多个cpuidle\_driver实例，由实例cpumask成员表示驱动适用于哪些CPU核，如果没有选择此选项则内核中只有一个cpuidle\_driver实例，适用于所有CPU核。本书暂不考虑多个cpuidle\_driver实例的情况。

注册cpuidle\_driver实例的接口函数**cpuidle\_register\_driver()**定义如下（/drivers/cpuidle/driver.c）：

int cpuidle\_register\_driver(struct cpuidle\_driver \*drv)

{

int ret;

spin\_lock(&cpuidle\_driver\_lock);

ret = **\_\_cpuidle\_register\_driver(drv)**; /\*/drivers/cpuidle/driver.c\*/

spin\_unlock(&cpuidle\_driver\_lock);

return ret;

}

注册函数内调用\_\_cpuidle\_register\_driver()函数完成实例的注册，函数定义如下：

static int \_\_cpuidle\_register\_driver(struct cpuidle\_driver \*drv)

{

int ret;

if (!drv || !drv->state\_count)

return -EINVAL;

if (cpuidle\_disabled())

return -ENODEV;

**\_\_cpuidle\_driver\_init(drv)**; /\*初始化cpuidle\_driver实例，/drivers/cpuidle/driver.c\*/

ret = **\_\_cpuidle\_set\_driver(drv)**; /\*设置cpuidle\_curr\_driver = drv\*/

...

if (drv->bctimer)

on\_each\_cpu\_mask(drv->cpumask, cpuidle\_setup\_broadcast\_timer, (void \*)1, 1);

poll\_idle\_init(drv); /\*没有设置ARCH\_HAS\_CPU\_RELAX配置选项为空操作\*/

return 0; /\*注册成功返回0\*/

}

\_\_cpuidle\_driver\_init(drv)函数用于初始化cpuidle\_driver实例，主要是设置drv->cpumask位图成员等简单操作，\_\_cpuidle\_set\_driver(drv)函数将全局指针cpuidle\_curr\_driver指向注册的cpuidle\_driver实例。

SoC平台相关代码中需要实现cpuidle\_driver实例，并调用cpuidle\_register\_driver()函数向cpuidle子系统注册。

注销驱动的函数为cpuidle\_unregister\_driver(struct cpuidle\_driver \*drv)。

### 2.4 cpuidle设备

cpuidle设备介于策略与驱动之间，cpuidle子系统中的设备是CPU核。cpuidle设备由cpuidle\_device结构体，内核为每个CPU核静态定义了此结构体实例（percpu变量）。在注册了驱动cpuidle\_driver实例后需要注册cpuidle\_device实例（初始化实例）。cpuidle\_device实例可通过属性在sysfs文件系统向用户空间导出idle级别信息。

#### ■数据结构

cpuidle子系统中的设备依然是CPU核，设备由cpuidle\_device结构体表示（/include/linux/cpuidle.h）：

struct cpuidle\_device {

unsigned int registered:1; /\*实例是否已经注册\*/

unsigned int enabled:1;

unsigned int **cpu**; /\*CPU核编号\*/

int last\_residency;

struct cpuidle\_state\_usage **states\_usage[CPUIDLE\_STATE\_MAX]**;

struct cpuidle\_state\_kobj \*kobjs[CPUIDLE\_STATE\_MAX];

struct cpuidle\_driver\_kobj \*kobj\_driver;

**struct cpuidle\_device\_kobj \*kobj\_dev**; /\*导出设备属性\*/

struct list\_head **device\_list**; /\*双链表成员，将实例链接到 链表\*/

...

};

cpuidle\_device结构体主要成员简介如下：

●**cpu：**CPU核编号。

●**states\_usage[]：**cpuidle\_state\_usage结构体数组，结构体定义如下：

struct cpuidle\_state\_usage {

unsigned long long disable;

unsigned long long usage;

unsigned long long time; /\* in US \*/

};

●**kobjs[]**：cpuidle\_state\_kobj结构体指针数组，结构体定义如下（/drivers/cpuidle/sysfs.c）：

struct cpuidle\_state\_kobj {

struct cpuidle\_state \*state;

struct cpuidle\_state\_usage \*state\_usage;

struct completion kobj\_unregister;

**struct kobject kobj**; /\*导出到sysfs\*/

};

cpuidle\_state\_kobj结构体主要用于导出idle级别属性。

●**kobj\_driver**：指向cpuidle\_driver\_kobj结构体，定义如下（/drivers/cpuidle/sysfs.c）：

struct cpuidle\_driver\_kobj {

struct cpuidle\_driver \*drv;

struct completion kobj\_unregister;

**struct kobject kobj;**

};

cpuidle\_driver\_kobj结构体主要用于导出驱动属性。

●**kobj\_dev**：指向cpuidle\_device\_kobj结构体，定义如下（/drivers/cpuidle/sysfs.c）：

struct cpuidle\_device\_kobj {

struct cpuidle\_device \*dev;

struct completion kobj\_unregister;

**struct kobject kobj**;

};

cpuidle\_device\_kobj结构体主要用于导出设备属性。

内核静态为每个CPU核创建了cpuidle\_device实例（/drivers/cpuidle/cpuidle.c）：

DEFINE\_PER\_CPU(**struct cpuidle\_device \*, cpuidle\_devices**); /\*全局指针数组\*/

DEFINE\_PER\_CPU(**struct cpuidle\_device, cpuidle\_dev**); /\*cpuidle\_device实例\*/

#### ■注册设备

cpuidle子系统中CPU设备由cpuidle\_device结构体表示，内核为每个CPU核静态定义了此结构体实例，函数**cpuidle\_register\_device()**用于设置结构体实例，注册CPU的电源管理属性。

cpuidle\_register\_device()函数定义如下（/drivers/cpuidle/cpuidle.c）：

int cpuidle\_register\_device(struct cpuidle\_device \*dev)

{

int ret = -EBUSY;

if (!dev)

return -EINVAL;

mutex\_lock(&cpuidle\_lock);

if (dev->registered)

goto out\_unlock;

**\_\_cpuidle\_device\_init(dev)**; /\*清零dev->states\_usage成员等，/drivers/cpuidle/cpuidle.c\*/

ret = **\_\_cpuidle\_register\_device(dev)**;

/\*实例关联到cpuidle\_devices指针项，添加到cpuidle\_detected\_devices双链表等\*/

...

ret = **cpuidle\_add\_sysfs(dev)**; /\*创建/sys/devices/system/cpu/cpuX/cpuidle/目录\*/

...

ret = **cpuidle\_enable\_device(dev)**; /\*使能设备，/drivers/cpuidle/sysfs.c\*/

/\*调用cpuidle\_curr\_governor->**enable**(drv, dev)函数，创建目录和属性文件等\*/

...

cpuidle\_install\_idle\_handler();

out\_unlock:

mutex\_unlock(&cpuidle\_lock);

return ret;

...

}

cpuidle\_register\_device()函数首先清零清零dev->states\_usage成员，然后将实例关联到cpuidle\_devices指针项，将实例添加到cpuidle\_detected\_devices双链表，为实例创建/sys/devices/system/cpu/cpuX/cpuidle/目录以容纳设备属性文件，最后使能设备，创建属性目录/文件等。

注销cpuidle\_device实例的函数为cpuidle\_unregister\_device(struct cpuidle\_device \*dev)。

SoC平台相关代码中需要实现cpuidle\_driver实例，并调用cpuidle\_register\_driver()函数向cpuidle子系统注册，然后调用cpuidle\_register\_device()函数注册驱动适用CPU核的cpuidle\_devices实例。

另外，接口函数**cpuidle\_register**(struct cpuidle\_driver \*drv,const struct cpumask \*const coupled\_cpus)用于同时注册cpuidle\_driver实例和cpuidle\_devices实例，drv指向cpuidle\_driver实例，coupled\_cpus指向驱动适用的CPU位图（/drivers/cpuidle/cpuidle.c）。注销函数为cpuidle\_unregister(struct cpuidle\_driver \*drv)。

### 2.5内核接口

idle子系统最重要的功能是向内核其它部分提供接口函数，实现对CPU核idle级别的操作，主要接口函数如下（/drivers/cpuidle/cpuidle.c）：

●int **cpuidle\_select**(struct cpuidle\_driver \*drv,struct cpuidle\_device \*dev)：询问cpuidle子系统CPU核选择一个idle级别。

●int **cpuidle\_enter**(struct cpuidle\_driver \*drv, struct cpuidle\_device \*dev,int index)：使CPU核进入指定的idle级别。

内核接口函数主要在内核idle线程中调用，详见**/kernel/sched/idle.c**文件代码。

### 2.6用户接口

cpuidle子系统还可以通过sysfs文件系统向用户提供操作接口，相关代码位于/drivers/cpuidle/sysfs.c文件内。

#### ■全局属性

cpuidle子系统全局属性导出到/sys/devices/system/cpu/cpuidle/目录下，在/drivers/cpuidle/sysfs.c文件内创建了添加到此目录下的默认属性：

static struct attribute\_group cpuidle\_attr\_group = {

.attrs = **cpuidle\_default\_attrs**,

.name = "**cpuidle**", /\*目录名称\*/

};

static struct attribute \*cpuidle\_default\_attrs[] = {

&dev\_attr\_**current\_driver**.attr, /\*只读属性，显示当前驱动\*/

&dev\_attr\_**current\_governor\_ro**.attr, /\*只读属性，显示策略类型\*/

NULL

};

如果需要从用户空间改变cpuidle策略类型，需要在命令行参数中设置"cpuidle\_sysfs\_switch"参数，此时默认属性定义如下：

static struct attribute \*cpuidle\_switch\_attrs[] = {

&dev\_attr\_available\_governors.attr, /\*只读属性，显示可用的策略类型\*/

&dev\_attr\_current\_driver.attr,

&dev\_attr\_**current\_governor**.attr, /\*读写属性，显示/设置当前策略类型（适用所有CPU核）\*/

NULL

}; /\*属性类型为通用的device\_attribute属性\*/

初始化函数cpuidle\_init()将会创建/sys/devices/system/cpu/cpuidle/目录，并将默认属性导出到此目录下：

static int \_\_init cpuidle\_init(void) /\*/drivers/cpuidle/cpuidle.c\*/

{

int ret;

if (cpuidle\_disabled())

return -ENODEV;

ret = **cpuidle\_add\_interface(cpu\_subsys.dev\_root)**;

/\*创建目录和属性文件，/drivers/cpuidle/sysfs.c\*/

...

latency\_notifier\_init(&cpuidle\_latency\_notifier);

return 0;

}

module\_param(off, int, 0444);

core\_initcall(cpuidle\_init);

cpuidle\_add\_interface()函数定义如下，函数内在表示CPU总线设备的目录下创建cpuidle目录，并在其下创建属性文件：

int cpuidle\_add\_interface(struct device \*dev)

{

if (sysfs\_switch)

cpuidle\_attr\_group.attrs = **cpuidle\_switch\_attrs**; /\*默认属性\*/

return sysfs\_create\_group(&dev->kobj, &**cpuidle\_attr\_group**); /\*创建目录和属性文件\*/

}

sysfs\_create\_group()函数定义在/fs/sysfs/group.c文件内，函数内先创建目录，再在目录下创建属性文件。

#### ■设备属性

cpuidle子系统中除了导出全局属性外，每个CPU核还有自己的属性文件。在注册CPU设备时将为CPU核在sysfs文件系统中创建目录/sys/devices/system/cpu/cpuX/cpuidle/以容纳设备属性文件，在使能设备时会将idle级别和驱动定义的属性导出到设备目录下，函数调用关系如下图所示。



cpuidle\_add\_sysfs()函数用于创建设备目录，cpuidle\_add\_device\_sysfs()函数用于创建级别/驱动属性目录和文件，先看一下前一个函数的实现：

int cpuidle\_add\_sysfs(struct cpuidle\_device \*dev)

{

struct cpuidle\_device\_kobj \*kdev;

struct device \*cpu\_dev = get\_cpu\_device((unsigned long)dev->cpu);

int error;

**kdev = kzalloc(sizeof(\*kdev), GFP\_KERNEL)**; /\*创建cpuidle\_device\_kobj实例\*/

...

kdev->dev = dev;

**dev->kobj\_dev = kdev**;

init\_completion(&kdev->kobj\_unregister);

error = kobject\_init\_and\_add(&**kdev->kobj**, &**ktype\_cpuidle**, &cpu\_dev->kobj,**"cpuidle"**);

/\*创建目录/sys/devices/system/cpu/cpuX/cpuidle/\*/

...

kobject\_uevent(&kdev->kobj, KOBJ\_ADD);

return 0;

}

ktype\_cpuidle实例定义如下：

static struct kobj\_type ktype\_cpuidle = {

.sysfs\_ops = &**cpuidle\_sysfs\_ops**, /\*读写属性操作，调用cpuidle\_attr结构体中读写函数\*/

.release = cpuidle\_sysfs\_release,

}; /\*默认属性为空\*/

CPU设备属性由cpuidle\_attr结构体表示，定义如下：

struct cpuidle\_attr {

struct attribute attr;

ssize\_t (\*show)(struct cpuidle\_device \*, char \*); /\*读函数\*/

ssize\_t (\*store)(struct cpuidle\_device \*, const char \*, size\_t count); /\*写函数\*/

};

设备默认属性为空。

cpuidle\_add\_device\_sysfs()函数用于导出级别和驱动中定义的属性，函数定义如下：

int cpuidle\_add\_device\_sysfs(struct cpuidle\_device \*device)

{

int ret;

ret = **cpuidle\_add\_state\_sysfs(device)**; /\*创建级别定义的属性文件\*/

...

ret = **cpuidle\_add\_driver\_sysfs(device)**; /\*创建驱动定义的属性文件\*/

..

return ret;

}

cpuidle\_add\_state\_sysfs(device)函数内为每个级别创建/sys/devices/system/cpu/cpuX/cpuidle/stateX/目录并导出属性文件。

cpuidle\_add\_driver\_sysfs(device)函数内创建/sys/devices/system/cpu/cpuX/cpuidle/driver/目录并导出属性文件。

#### ■级别属性

cpuidle\_add\_state\_sysfs()函数用于导出级别属性，函数定义如下：

static int cpuidle\_add\_state\_sysfs(struct cpuidle\_device \*device)

{

int i, ret = -ENOMEM;

struct cpuidle\_state\_kobj \*kobj;

struct cpuidle\_device\_kobj \***kdev = device->kobj\_dev**; /\*父节点\*/

struct cpuidle\_driver \*drv = cpuidle\_get\_cpu\_driver(device); /\*当前注册的驱动\*/

for (i = 0; i < drv->state\_count; i++) { /\*遍历驱动中的级别数组\*/

kobj = kzalloc(sizeof(struct cpuidle\_state\_kobj), GFP\_KERNEL); /\*创建结构体实例\*/

if (!kobj)

goto error\_state;

kobj->state = **&drv->states[i]**;

kobj->state\_usage = &**device->states\_usage[i]**;

init\_completion(&kobj->kobj\_unregister);

ret = kobject\_init\_and\_add(&kobj->kobj, &**ktype\_state\_cpuidle**,&kdev->kobj, "**state%d**", i);

/\*创建目录/sys/devices/system/cpu/cpuX/cpuidle/stateX/\*/

...

kobject\_uevent(&kobj->kobj, KOBJ\_ADD);

device->kobjs[i] = kobj;

}

return 0;

...

}

ktype\_state\_cpuidle实例定义如下：

static struct kobj\_type ktype\_state\_cpuidle = {

.sysfs\_ops = &cpuidle\_state\_sysfs\_ops, /\*调用cpuidle\_state\_attr结构体中的读写函数\*/

.default\_attrs = **cpuidle\_state\_default\_attrs**, /\*默认属性\*/

.release = cpuidle\_state\_sysfs\_release,

};

idle级别属性由cpuidle\_state\_attr结构体表示，定义如下（/drivers/cpuidle/sysfs.c）：

struct cpuidle\_state\_attr {

struct attribute attr;

ssize\_t (\*show)(struct cpuidle\_state \*, struct cpuidle\_state\_usage \*, char \*);

ssize\_t (\*store)(struct cpuidle\_state \*, struct cpuidle\_state\_usage \*, const char \*, size\_t);

};

cpuidle子系统默认的级别属性定义如下：

static struct attribute \***cpuidle\_state\_default\_attrs[]** = {

&attr\_name.attr,

&attr\_desc.attr,

&attr\_latency.attr,

&attr\_residency.attr,

&attr\_power.attr,

&attr\_usage.attr,

&attr\_time.attr,

&attr\_disable.attr,

NULL

};

每个idle级别属性对应/sys/devices/system/cpu/cpuX/cpuidle/stateX/目录下的一个属性文件，以上属性除disable外，其它都是只读属性，用于显示信息。具体属性的定义和读写操作函数请读者自行阅读源代码。

#### ■驱动属性

cpuidle\_add\_driver\_sysfs(device)函数内创建/sys/devices/system/cpu/cpuX/cpuidle/driver/目录并导出驱动属性文件，函数定义如下：

static int cpuidle\_add\_driver\_sysfs(struct cpuidle\_device \*dev)

{

struct cpuidle\_driver\_kobj \*kdrv;

**struct cpuidle\_device\_kobj \*kdev = dev->kobj\_dev**; /\*父节点\*/

struct cpuidle\_driver \*drv = cpuidle\_get\_cpu\_driver(dev); /\*驱动\*/

int ret;

**kdrv = kzalloc(sizeof(\*kdrv), GFP\_KERNEL)**; /\*创建cpuidle\_driver\_kobj结构体实例\*/

...

kdrv->drv = drv;

init\_completion(&kdrv->kobj\_unregister);

ret = kobject\_init\_and\_add(&kdrv->kobj, &**ktype\_driver\_cpuidle**,&kdev->kobj, **"driver"**);

/\*创建/sys/devices/system/cpu/cpuX/cpuidle/driver/目录\*/

...

kobject\_uevent(&kdrv->kobj, KOBJ\_ADD);

dev->kobj\_driver = kdrv;

return ret;

}

ktype\_driver\_cpuidle实例定义如下：

static struct kobj\_type ktype\_driver\_cpuidle = {

.sysfs\_ops = &cpuidle\_driver\_sysfs\_ops, /\*调用cpuidle\_driver\_attr结构体中的读写函数\*/

.default\_attrs = **cpuidle\_driver\_default\_attrs**, /\*默认属性\*/

.release = cpuidle\_driver\_sysfs\_release,

};

驱动属性由cpuidle\_driver\_attr结构体表示，定义如下：

struct cpuidle\_driver\_attr {

struct attribute attr;

ssize\_t (\*show)(struct cpuidle\_driver \*, char \*);

ssize\_t (\*store)(struct cpuidle\_driver \*, const char \*, size\_t);

};

默认属性数组定义如下：

static struct attribute \***cpuidle\_driver\_default\_attrs[]** = {

&**attr\_driver\_name**.attr, /\*只读属性，显示驱动名称\*/

NULL

};

驱动默认属性只有一个，那就是显示驱动名称（只读属性）。

总结，cpuidle子系统提供的接口为sysfs文件系统属性文件，例如：

●/sys/devices/system/cpu/cpuidle/**current\_governor：**显示/设置idle策略。

●/sys/devices/system/cpu/cpuX/cpuidle/stateX/：idle级别属性文件。

●/sys/devices/system/cpu/cpuX/cpuidle/driver/：驱动属性文件。

### 2.7驱动示例

SoC（平台）驱动主要是实现并注册cpuidle\_driver实例，下面以MIPS体系结构为例简要说明cpuidle驱动的实现，驱动代码位于/drivers/cpuidle/cpuidle-cps.c文件内（需选择MIPS\_CPS、SMP和CPS\_CPUIDLE配置选项）。

MIPS处理器idle级别名称定义如下：

enum cps\_idle\_state {

STATE\_WAIT = 0, /\* MIPS wait instruction, coherent \*/

STATE\_NC\_WAIT, /\* MIPS wait instruction, non-coherent \*/

STATE\_CLOCK\_GATED, /\* Core clock gated \*/

STATE\_POWER\_GATED, /\* Core power gated \*/

STATE\_COUNT

};

驱动cpuidle\_driver实例定义如下：

static struct cpuidle\_driver cps\_driver = {

.name = "cpc\_cpuidle",

.owner = THIS\_MODULE,

.states = {

[STATE\_WAIT] = MIPS\_CPUIDLE\_WAIT\_STATE,

[STATE\_NC\_WAIT] = {

.enter = **cps\_nc\_enter**,

.exit\_latency = 200,

.target\_residency = 450,

.name = "nc-wait",

.desc = "non-coherent MIPS wait",

},

[STATE\_CLOCK\_GATED] = {

.enter = cps\_nc\_enter,

.exit\_latency = 300,

.target\_residency = 700,

.flags = CPUIDLE\_FLAG\_TIMER\_STOP,

.name = "clock-gated",

.desc = "core clock gated",

},

[STATE\_POWER\_GATED] = {

.enter = cps\_nc\_enter,

.exit\_latency = 600,

.target\_residency = 1000,

.flags = CPUIDLE\_FLAG\_TIMER\_STOP,

.name = "power-gated",

.desc = "core power gated",

},

},

.state\_count = STATE\_COUNT,

.safe\_state\_index = 0,

};

初始化函数**cps\_cpuidle\_init(void)**用于注册驱动和设备，函数代码简列如下：

static int \_\_init cps\_cpuidle\_init(void)

{

int err, cpu, core, i;

struct cpuidle\_device \*device;

...

err = **cpuidle\_register\_driver(&cps\_driver)**; /\*注册驱动cpuidle\_driver实例\*/

...

for\_each\_possible\_cpu(cpu) { /\*遍历CPU核\*/

core = cpu\_data[cpu].core;

device = &per\_cpu(cpuidle\_dev, cpu);

device->cpu = cpu;

...

err = **cpuidle\_register\_device(device)**; /\*注册设备\*/

...

}

return 0;

...

}

device\_initcall(cps\_cpuidle\_init);

## 3 PM QoS

前面介绍的功耗管理主要是从CPU核的角度来控制功耗，从本小节开始将介绍如何从外部设备的角度来实现功耗的管理。

PM QoS是PM Quality Of Service的缩写，意思是功耗管理服务质量。内核PM QoS框架向其它部分或用户进程提供了控制功耗管理质量（期望）的接口，本小节主要介绍PM QoS框架的实现。

内核定义了两个不同PM QoS 框架：

■**PM QoS**：系统级的控制，用于设置cpu\_dma\_latency, network\_latency, network\_throughput,

memory\_bandwidth限制值，相关代码位于/kernel/power/qos.c文件内。

■**per-device PM QoS**：用于设备设置功耗管理限制值和标记，相关代码位于/drivers/base/power/qos.c文件内。

### 3.1 PM QoS

PM QoS框架如下图所示，框架代码默认编译入内核，不受配置选项的控制。PM QoS框架核心层由4个类型约束实例（pm\_qos\_constraints结构体）组成。约束相当于向功耗管理子系统（如cpuidle等）提出的限制要求，以控制其行为，内核其它部分或设备驱动可向约束发送请求，提出限制条件，限制条件在约束中按优先级从高到低排序。功耗管理子系统在执行某项操作前会读取约束当前的限制值，以控制自身的行为。

框架中内核接口用于内核代码通过内核API向PM QoS框架约束限制提交请求。用户接口用于进程通过设备文件向PM QoS框架提交请求，设备文件操作函数中调用内核API完成操作。

用户接口创建的设备文件有/dev/cpu\_dma\_latency，/dev/network\_latency，/dev/network\_throughput，/dev/memory\_bandwidth。



#### ■核心层

PM QoS框架核心层由pm\_qos\_constraints结构体表示的约束组成，结构体定义如下：

struct pm\_qos\_constraints { /\*/include/linux/pm\_qos.h\*/

struct plist\_head **list**; /\*请求链表头\*/

s32 **target\_value**; /\* Do not change to 64 bit \*/

s32 default\_value;

s32 no\_constraint\_value;

enum pm\_qos\_type **type**; /\*类型\*/

struct blocking\_notifier\_head **\*notifiers**;

};

pm\_qos\_constraints结构体主要成员简介如下：

●**list：**优先级链表头，链表中节点为plist\_node结构体（内嵌在pm\_qos\_request结构体），链表中成员按优先级递减排序。

●**type：**类型，由pm\_qos\_type枚举类型表示，定义如下：

enum pm\_qos\_type {

PM\_QOS\_UNITIALIZED,

PM\_QOS\_MAX, /\* return the largest value \*/

PM\_QOS\_MIN, /\* return the smallest value \*/

PM\_QOS\_SUM /\* return the sum \*/

};

●**notifiers：**指向阻塞通知链表头。

内核在/kernel/power/qos.c文件内静态定义了4个pm\_qos\_constraints实例：

static BLOCKING\_NOTIFIER\_HEAD(cpu\_dma\_lat\_notifier);

static struct pm\_qos\_constraints cpu\_dma\_constraints = {

.list = PLIST\_HEAD\_INIT(cpu\_dma\_constraints.list),

.target\_value = PM\_QOS\_CPU\_DMA\_LAT\_DEFAULT\_VALUE,

.default\_value = PM\_QOS\_CPU\_DMA\_LAT\_DEFAULT\_VALUE,

.no\_constraint\_value = PM\_QOS\_CPU\_DMA\_LAT\_DEFAULT\_VALUE,

.type = PM\_QOS\_MIN,

.notifiers = &cpu\_dma\_lat\_notifier,

};

static BLOCKING\_NOTIFIER\_HEAD(network\_lat\_notifier);

static struct pm\_qos\_constraints network\_lat\_constraints = {

.list = PLIST\_HEAD\_INIT(network\_lat\_constraints.list),

.target\_value = PM\_QOS\_NETWORK\_LAT\_DEFAULT\_VALUE,

.default\_value = PM\_QOS\_NETWORK\_LAT\_DEFAULT\_VALUE,

.no\_constraint\_value = PM\_QOS\_NETWORK\_LAT\_DEFAULT\_VALUE,

.type = PM\_QOS\_MIN,

.notifiers = &network\_lat\_notifier,

};

static BLOCKING\_NOTIFIER\_HEAD(network\_throughput\_notifier);

static struct pm\_qos\_constraints network\_tput\_constraints = {

.list = PLIST\_HEAD\_INIT(network\_tput\_constraints.list),

.target\_value = PM\_QOS\_NETWORK\_THROUGHPUT\_DEFAULT\_VALUE,

.default\_value = PM\_QOS\_NETWORK\_THROUGHPUT\_DEFAULT\_VALUE,

.no\_constraint\_value = PM\_QOS\_NETWORK\_THROUGHPUT\_DEFAULT\_VALUE,

.type = PM\_QOS\_MAX,

.notifiers = &network\_throughput\_notifier,

};

static BLOCKING\_NOTIFIER\_HEAD(memory\_bandwidth\_notifier);

static struct pm\_qos\_constraints memory\_bw\_constraints = {

.list = PLIST\_HEAD\_INIT(memory\_bw\_constraints.list),

.target\_value = PM\_QOS\_MEMORY\_BANDWIDTH\_DEFAULT\_VALUE,

.default\_value = PM\_QOS\_MEMORY\_BANDWIDTH\_DEFAULT\_VALUE,

.no\_constraint\_value = PM\_QOS\_MEMORY\_BANDWIDTH\_DEFAULT\_VALUE,

.type = **PM\_QOS\_SUM**,

.notifiers = &memory\_bandwidth\_notifier,

};

pm\_qos\_constraints结构体中list成员链接的是请求**pm\_qos\_request**实例，结构体定义如下：

struct pm\_qos\_request { /\*/include/linux/pm\_qos.h\*/

struct plist\_node **node**; /\*链接到pm\_qos\_constraints.list链表\*/

int **pm\_qos\_class**; /\*初始值必须为0\*/

struct delayed\_work work ; /\*延时工作\*/

};

pm\_qos\_request结构体成员简介如下：

●**node**：plist\_node结构体成员，链接到pm\_qos\_constraints.list链表。plist\_node结构体定义如下：

struct plist\_node {

int **prio**; /\*优先级，确定pm\_qos\_request实例在链表中的顺序\*/

struct list\_head prio\_list; /\*优先级链表\*/

struct list\_head node\_list; /\*顺序链表\*/

};

●**pm\_qos\_class：**约束类别，定义如下：

enum {

PM\_QOS\_RESERVED = 0, /\*对应null\_pm\_qos实例\*/

PM\_QOS\_CPU\_DMA\_LATENCY,

PM\_QOS\_NETWORK\_LATENCY,

PM\_QOS\_NETWORK\_THROUGHPUT,

PM\_QOS\_MEMORY\_BANDWIDTH,

/\* insert new class ID \*/

PM\_QOS\_NUM\_CLASSES,

};

#### ■内核接口

内核接口函数包括向PM QoS框架添加请求、更新请求、获取约束值等。

**●添加请求**

添加请求的接口函数为**pm\_qos\_add\_request**()，函数代码如下：

void **pm\_qos\_add\_request**(struct pm\_qos\_request \*req,int **pm\_qos\_class**, s32 **value**)

/\*req：请求，pm\_qos\_class：约束类别，value：数值\*/

{

if (!req) /\*guard against callers passing in null \*/

return;

if (pm\_qos\_request\_active(req)) { /\*req->pm\_qos\_class成员值须为0\*/

WARN(1, KERN\_ERR "pm\_qos\_add\_request() called for already added request\n");

return;

}

**req->pm\_qos\_class = pm\_qos\_class**;

INIT\_DELAYED\_WORK(&req->work, pm\_qos\_work\_fn);

trace\_pm\_qos\_add\_request(pm\_qos\_class, value);

**pm\_qos\_update\_target**(**pm\_qos\_array[pm\_qos\_class]->constraints**,

&req->node, PM\_QOS\_ADD\_REQ, value);

}

**pm\_qos\_update\_target()**函数用于将请求添加到约束请求链表的合适位置，代码如下：

int pm\_qos\_update\_target(struct pm\_qos\_constraints \*c, struct plist\_node \*node,

enum pm\_qos\_req\_action **action**, int **value**)

{

unsigned long flags;

int prev\_value, curr\_value, new\_value;

int ret;

spin\_lock\_irqsave(&pm\_qos\_lock, flags);

prev\_value = **pm\_qos\_get\_value(c)**; /\*约束限制值，通常为请求链表第一个节点的优先级值\*/

if (value == PM\_QOS\_DEFAULT\_VALUE)

new\_value = c->default\_value;

else

new\_value = **value**;

switch (action) {

case PM\_QOS\_REMOVE\_REQ: /\*移出请求\*/

plist\_del(node, &c->list); /\*移出请求\*/

break;

case PM\_QOS\_UPDATE\_REQ: /\*更新请求\*/

plist\_del(node, &c->list);

case PM\_QOS\_ADD\_REQ: /\*添加请求\*/

plist\_node\_init(node, new\_value);

**plist\_add(node, &c->list)**; /\*请求按优先级插入到请求链表\*/

break;

default:

/\* no action \*/

;

}

curr\_value = **pm\_qos\_get\_value(c)**;

**pm\_qos\_set\_value(c, curr\_value);**  /\*重新获取当前限制值\*/

spin\_unlock\_irqrestore(&pm\_qos\_lock, flags);

trace\_pm\_qos\_update\_target(action, prev\_value, curr\_value);

if (prev\_value != curr\_value) {

ret = 1;

if (c->notifiers)

blocking\_notifier\_call\_chain(c->notifiers,(unsigned long)curr\_value,NULL);

} else {

ret = 0;

}

return ret;

}

**●更新请求**

更新现有请求的接口函数**pm\_qos\_update\_request()**定义如下：

void pm\_qos\_update\_request(struct pm\_qos\_request \*req,s32 new\_value)

{

if (!req)

return;

if (!pm\_qos\_request\_active(req)) {

...

}

cancel\_delayed\_work\_sync(&req->work);

**\_\_pm\_qos\_update\_request**(req, new\_value);

/\*更新请求限制值，**执行通知链**等，源代码请读者自行阅读\*/

}

●**获取约束限制值**

获取约束当前限制值的接口函数**pm\_qos\_request()**定义如下：

int pm\_qos\_request(int pm\_qos\_class)

{

return pm\_qos\_read\_value(pm\_qos\_array[pm\_qos\_class]->constraints); /\*返回target\_value值\*/

}

功耗管理子系统通过此接口函数获取当前约束的限制值来控制自己的行为。

#### ■用户接口

PM QoS框架定义了pm\_qos\_object结构体，用于将通过设备文件将约束类型导出用户空间，结构体定义如下（/kernel/power/qos.c）：

struct pm\_qos\_object {

struct pm\_qos\_constraints \***constraints**; /\*指向pm\_qos\_constraints实例\*/

struct miscdevice **pm\_qos\_power\_miscdev**; /\*导出到字符设备数据库\*/

char \***name;**  /\*设备文件名\*/

};

PM QoS框架为每个约束类型定义了pm\_qos\_object实例，全局指针数组指向各实例：

static struct pm\_qos\_object \*pm\_qos\_array[] = { /\*关联各pm\_qos\_constraints实例\*/

&null\_pm\_qos,

&cpu\_dma\_pm\_qos,

&network\_lat\_pm\_qos,

&network\_throughput\_pm\_qos,

&memory\_bandwidth\_pm\_qos,

};

用户接口采用了misc驱动程序框架，初始化函数**pm\_qos\_power\_init()**定义如下：

static int \_\_init pm\_qos\_power\_init(void)

{

int ret = 0;

int i;

struct dentry \*d;

...

d = debugfs\_create\_dir("pm\_qos", NULL);

...

for (i = PM\_QOS\_CPU\_DMA\_LATENCY; i < PM\_QOS\_NUM\_CLASSES; i++) {

ret = **register\_pm\_qos\_misc(pm\_qos\_array[i], d)**;

...

}

return ret;

}

late\_initcall(pm\_qos\_power\_init);

register\_pm\_qos\_misc()函数注册pm\_qos\_object实例中的miscdevice结构体成员，为各约束创建设备文件，例如：/dev/cpu\_dma\_latency，设备文件操作结构实例如下：

static const struct file\_operations pm\_qos\_power\_fops = {

.write = **pm\_qos\_power\_write**,

.read = **pm\_qos\_power\_read**,

.open = **pm\_qos\_power\_open**,

.release = pm\_qos\_power\_release,

.llseek = noop\_llseek,

};

pm\_qos\_power\_open()函数内将创建、添加pm\_qos\_request实例（默认的优先级值）并赋予文件file实例，pm\_qos\_power\_read()函数读取约束的当前限制值，pm\_qos\_power\_write()函数用于设置打开请求实例的限制值。

### 3.2 per-device PM QoS

per-device PM QoS框架如下图所示（需选择PM配置选项，/kernel/power/Kconfig），通用驱动模型device结构体中power成员指向dev\_pm\_info结构体成员，dev\_pm\_info结构体qos成员指向dev\_pm\_qos结构体。dev\_pm\_qos结构体中包含pm\_qos\_constraints结构体约束成员。请求由dev\_pm\_qos\_request结构体表示，内核接口函数可以向约束成员添加限制请求，读取设备的约束值等。



#### ■数据结构

dev\_pm\_qos结构体属性设备device实例，结构体定义如下（/include/linux/pm\_qos.h）：

struct dev\_pm\_qos {

**struct pm\_qos\_constraints resume\_latency;**

**struct pm\_qos\_constraints latency\_tolerance**;

**struct pm\_qos\_flags flags;**

struct dev\_pm\_qos\_request \*resume\_latency\_req;

struct dev\_pm\_qos\_request \*latency\_tolerance\_req;

struct dev\_pm\_qos\_request \*flags\_req;

};

请求由dev\_pm\_qos\_request结构体表示，定义如下（/include/linux/pm\_qos.h）：

struct dev\_pm\_qos\_request {

enum dev\_pm\_qos\_req\_type **type**;

union {

struct plist\_node **pnode**;

struct pm\_qos\_flags\_request **flr**; /\*标记类型节点\*/

} data;

struct device \*dev;

};

dev\_pm\_qos\_request实例有两个类型，一是普通的限制节点，一是表示标记值的节点，结构体主要成员简介如下：

●**type：**请求类型，枚举类型定义如下：

enum dev\_pm\_qos\_req\_type {

DEV\_PM\_QOS\_RESUME\_LATENCY = 1,

DEV\_PM\_QOS\_LATENCY\_TOLERANCE,

DEV\_PM\_QOS\_FLAGS, /\*标记类型请求\*/

};

●**flr：**pm\_qos\_flags\_request结构体成员，定义如下：

struct pm\_qos\_flags\_request {

struct list\_head node;

s32 **flags**; /\* Do not change to 64 bit \*/

};

标记flags成员取值目前只有两个：

#define PM\_QOS\_FLAG\_NO\_POWER\_OFF (1 << 0)

#define PM\_QOS\_FLAG\_REMOTE\_WAKEUP (1 << 1)

#### ■接口函数

per-device PM QoS框架向内核提供了接口函数，下面简要介绍其中的几个。

●**添加请求**

添加请求的接口函数为**dev\_pm\_qos\_add\_request()**，函数定义如下（/drivers/base/power/qos.c）：

int dev\_pm\_qos\_add\_request(struct device \*dev, struct dev\_pm\_qos\_request \*req,

enum dev\_pm\_qos\_req\_type type, s32 value)

{

int ret;

mutex\_lock(&dev\_pm\_qos\_mtx);

ret = **\_\_dev\_pm\_qos\_add\_request(dev, req, type, value)**;

mutex\_unlock(&dev\_pm\_qos\_mtx);

return ret;

}

\_\_dev\_pm\_qos\_add\_request()函数定义如下（/drivers/base/power/qos.c）：

static int \_\_dev\_pm\_qos\_add\_request(struct device \*dev,struct dev\_pm\_qos\_request \*req,

enum dev\_pm\_qos\_req\_type type, s32 value)

{

int ret = 0;

...

if (IS\_ERR(dev->power.qos))

ret = -ENODEV;

else if (!**dev->power.qos**) /\*尚未关联dev\_pm\_qos实例\*/

ret = **dev\_pm\_qos\_constraints\_allocate**(dev); /\*创建并初始化**dev\_pm\_qos**实例\*/

trace\_dev\_pm\_qos\_add\_request(dev\_name(dev), type, value);

if (!ret) {

req->dev = dev;

req->type = type;

ret = **apply\_constraint(req, PM\_QOS\_ADD\_REQ, value)**;

/\*添加请求到dev\_pm\_qos实例的pm\_qos\_constraints实例成员中\*/

}

return ret;

}

●**s32** **dev\_pm\_qos\_read\_value**(struct device \*dev)：读取设备的PM QoS限制值。

●**int** **dev\_pm\_qos\_add\_notifier**(struct device \*dev, struct notifier\_block \*notifier)：添加阻塞通知。

## 4 PM core

PM core（内核功耗管理核心框架）提供了使用系统进入睡眠状态的机制，以达到节能的目的，本小节主要介绍PM core框架的实现。

内核支持4种系统睡眠状态，若需要支持系统睡眠和唤醒需选择SUSPEND和PM\_SLEEP（自动选择PM选项），各配置选项定义在/kernel/power/Kconfig文件内，PM core框架代码在/kernel/power/目录下实现。

### 4.1概述

内核支持4种类型的系统睡眠状态，按睡眠等级从低到高依次是："freeze"（冻结）、"standby"（旁路）、"mem"（挂起到RAM）和"disk"（挂起到硬盘、冬眠）。

●**"freeze"**：普通的纯软件的，轻量级的系统睡眠状态。通过冻结用户进程（可能包含内核线程）、使外设进入低功耗状态以达到节能的目的，此状态总是被支持的。

●**"standby"**：旁路状态，提供适度的节能状态，在"freeze"基础上使非启动CPU核离线，所有执行底层系统（外设）挂起函数（CPU仍供电）。

●**"mem"**：挂起到RAM（STR），系统中除内存外都进入低功耗状态，系统和设备状态保存在内存中，所有设备挂起，外设总线断电。

●**"disk"**：挂起到硬盘（STD），又称hibernation（冬眠），最深度的睡眠状态，系统和设备状态保存到硬盘，整个系统下电，恢复时从硬盘中恢复。

各睡眠状态的描述可参考内核文档/Documentation/power/states.txt。

#### ■PM core框架

PM core框架如下图所示，用户进程可通过向/sys/power/state属性文件写入睡眠状态描述字符串使系统进入相应的睡眠状态。PM core框架实现各睡眠状态需要相应数据结构实例的支持，以实现相应的操作，例如："standby"和"mem"状态需要platform\_suspend\_ops数据结构的支持。各数据结构实例由平台或驱动相关代码实现并注册。

属性文件/sys/power/state的写操作函数根据指定的睡眠状态执行相应的操作，并调用相应数据结构中的执行函数进入相应的睡眠状态。



#### ■初始化

PM初始化函数pm\_init()主要是创建PM core框架在sysfs文件系统中的节点，并创建属性文件，函数定义如下（/kernel/power/main.c）：

static int \_\_init pm\_init(void)

{

int error = **pm\_start\_workqueue()**; /\*创建pm\_wq工作队列，/kernel/power/main.c\*/

...

hibernate\_image\_size\_init();

/\*分配内存空间（需选择HIBERNATION选项），/kernel/power/snapshot.c\*/

hibernate\_reserved\_size\_init();

/\*分配内存空间（需选择HIBERNATION选项），/kernel/power/snapshot.c\*/

power\_kobj = **kobject\_create\_and\_add("power", NULL)**; /\*创建/sys/power/目录\*/

...

error = **sysfs\_create\_group**(power\_kobj, &**attr\_group**);

/\*在/sys/power/目录下创建属性文件（组）\*/

...

pm\_print\_times\_init(); /\*没有选择PM\_SLEEP\_DEBUG配置选项时操作为空\*/

return **pm\_autosleep\_init()**; /\*自动睡眠初始化，/kernel/power/autosleep.c\*/

}

**core\_initcall(pm\_init)**;

在/sys/power/目录下创建的属性组如下（/kernel/power/main.c）：

static struct attribute \* g[] = {

&**state\_attr**.attr, /\*state属性\*/

#ifdef CONFIG\_PM\_TRACE

&pm\_trace\_attr.attr,

&pm\_trace\_dev\_match\_attr.attr,

#endif

#ifdef CONFIG\_PM\_SLEEP

&**pm\_async\_attr**.attr,

&**wakeup\_count\_attr**.attr,

#ifdef CONFIG\_PM\_AUTOSLEEP

&**autosleep\_attr**.attr,

#endif

#ifdef CONFIG\_PM\_WAKELOCKS

&wake\_lock\_attr.attr,

&wake\_unlock\_attr.attr,

#endif

#ifdef CONFIG\_PM\_DEBUG

&pm\_test\_attr.attr,

#endif

#ifdef CONFIG\_PM\_SLEEP\_DEBUG

&pm\_print\_times\_attr.attr,

#endif

#endif

#ifdef CONFIG\_FREEZER

&pm\_freeze\_timeout\_attr.attr,

#endif

NULL,

};

static struct attribute\_group **attr\_group** = { /\*属性组\*/

.**attrs = g**,

};

内核在/kernel/power/power.h头文件内定义了声明PM core属性的宏：

#define power\_attr(\_name) \

static struct kobj\_attribute \_name##\_attr = { \

.attr = { \

.name = \_\_stringify(\_name), \

.mode = **0644**, \ /\*读写权限\*/

}, \

.show = \_name##\_show, \ /\*读函数名称\*/

.store = \_name##\_store, \ /\*写函数名称\*/

}

### 4.2数据结构

内核内部用suspend\_state\_t数据类型表示系统睡眠状态，数据类型定义如下（/include/linux/suspend.h）：

typedef int \_\_bitwise suspend\_state\_t; /\*位图\*/

#define PM\_SUSPEND\_ON ((\_\_force suspend\_state\_t) 0) /\*不睡眠\*/

#define PM\_SUSPEND\_FREEZE ((\_\_force suspend\_state\_t) 1) /\*"freeze"\*/

#define PM\_SUSPEND\_STANDBY ((\_\_force suspend\_state\_t) 2) /\*"standby"\*/

#define PM\_SUSPEND\_MEM ((\_\_force suspend\_state\_t) 3) /\*"mem"\*/

#define PM\_SUSPEND\_MIN PM\_SUSPEND\_FREEZE

#define PM\_SUSPEND\_MAX ((\_\_force suspend\_state\_t) 4) /\*"disk"\*/

内核在/kernel/power/suspend.c定义了睡眠状态的描述字符串指针数组：

const char \*pm\_labels[] = { **"mem", "standby", "freeze", NULL** };

const char \*pm\_states[PM\_SUSPEND\_MAX];

#### ■平台睡眠结构

"standby"和"mem"睡眠状态需要平台底层代码的支持，platform\_suspend\_ops结构体中包含睡眠状态的操作函数，平台（处理器）相关代码需要实现并注册此结构体实现，结构体定义如下：

struct platform\_suspend\_ops { /\*/include/linux/suspend.h\*/

int (\*valid)(suspend\_state\_t state); /\*检查平台是否支持参数指定的睡眠状态\*/

int (\*begin)(suspend\_state\_t state); /\*转入指定睡眠状态前的初始化工作\*/

int (\*prepare)(void); /\*准备进入指定的睡眠状态，在begin()之后调用\*/

int (\*prepare\_late)(void); /\*后期的准备工作\*/

int (\***enter**)(suspend\_state\_t state); /\*进入指定的睡眠状态\*/

void (\***wake**)(void); /\*从睡眠状态唤醒\*/

void (\*finish)(void); /\*结束唤醒操作\*/

bool (\*suspend\_again)(void); /\*是否要再次进入睡眠状态\*/

void (\*end)(void); /\*设备唤醒后调用\*/

void (\*recover)(void); /\*挂起失败后调用\*/

};

内核在/kernel/power/suspend.c文件内定义了全局指针指向注册的platform\_suspend\_ops实例：

static const struct platform\_suspend\_ops **\*suspend\_ops**;

接口函数**suspend\_set\_ops()**用于注册platform\_suspend\_ops实例，定义如下（/kernel/power/suspend.c）：

void suspend\_set\_ops(const struct platform\_suspend\_ops \*ops)

{

suspend\_state\_t i;

int j = 0;

lock\_system\_sleep();

**suspend\_ops = ops**;

for (i = **PM\_SUSPEND\_MEM; i >= PM\_SUSPEND\_STANDBY**; i--)

if (valid\_state(i)) {

**pm\_states[i] = pm\_labels[j++]**; /\*指向睡眠状态描述字符串\*/

} else if (!relative\_states) {

pm\_states[i] = NULL;

j++;

}

**pm\_states[PM\_SUSPEND\_FREEZE] = pm\_labels[j]**; /\*指向"freeze"字符串\*/

unlock\_system\_sleep();

}

平台（处理器）相关代码需要实例并注册platform\_suspend\_ops实例，例如，loongson64平台代码中实现并注册了此实例（/arch/mips/loongson64/common/pm.c）：

static const struct platform\_suspend\_ops loongson\_pm\_ops = {

.valid = loongson\_pm\_valid\_state,

.enter = loongson\_pm\_enter,

};

static int \_\_init loongson\_pm\_init(void)

{

**suspend\_set\_ops(&loongson\_pm\_ops)***;*

return 0;

}

arch\_initcall(loongson\_pm\_init);

#### ■平台冻结操作结构

系统进入"freeze"状态需要platform\_freeze\_ops数据结构的支持，定义如下（/include/linux/suspend.h）：

struct platform\_freeze\_ops {

int (\*begin)(void);

int (\*prepare)(void);

void (\*restore)(void);

void (\*end)(void);

};

内核定义了全局指针freeze\_ops用于指向注册的platform\_freeze\_ops实例：

static const struct platform\_freeze\_ops \***freeze\_ops**;

注册platform\_freeze\_ops实例的接口函数为**freeze\_set\_ops**(const struct platform\_freeze\_ops \*ops)，函数内只是简单地将实例赋予freeze\_ops指针成员。

platform\_freeze\_ops实例在驱动代码中实现并注册，例如：/drivers/acpi/sleep.c。

#### ■冬眠操作结构

系统进入"disk"（冬眠）睡眠状态需要platform\_hibernation\_ops数据结构的支持，定义如下：

struct platform\_hibernation\_ops { /\*/include/linux/suspend.h\*/

int (\*begin)(void);

void (\*end)(void);

int (\*pre\_snapshot)(void);

void (\*finish)(void);

int (\*prepare)(void);

int (\*enter)(void);

void (\*leave)(void);

int (\*pre\_restore)(void);

void (\*restore\_cleanup)(void);

void (\*recover)(void);

};

platform\_hibernation\_ops结构体中的操作函数与platform\_suspend\_ops结构体相似，这里就不再介绍了，读者可参考头文件中的说明。

内核定义了全局指针hibernation\_ops指向注册的platform\_suspend\_ops实例：

static const struct platform\_hibernation\_ops \***hibernation\_ops**;

注册实例的**hibernation\_set\_ops**(const struct platform\_hibernation\_ops \*ops)函数主要工作就是将实例赋予全局指针hibernation\_ops，请读者自行阅读源代码（/kernel/power/hibernate.c）。

hibernation\_ops实例在驱动代码中实现并注册，例如：/drivers/acpi/sleep.c。

### 4.3用户接口

用户进程可通过对**/sys/power/state**属性文件写入"freeze"、"standby"、"mem"或"disk"使系统进入相应的睡眠状态。

state属性定义如下（/kernel/power/main.c）：

power\_attr(state);

state属性读函数state\_show()用于显示系统支持的睡眠状态，源代码请读者自行阅读。

state属性写函数state\_store()用于设置系统睡眠状态，定义如下（/kernel/power/main.c）：

static ssize\_t state\_store(struct kobject \*kobj, struct kobj\_attribute \*attr,const char \*buf, size\_t n)

/\*buf：指向睡眠状态描述字符串，n：字符串长度\*/

{

suspend\_state\_t **state**; /\*睡眠状态类型\*/

int error;

error = pm\_autosleep\_lock();

...

if (pm\_autosleep\_state() > PM\_SUSPEND\_ON) {

...

}

state = **decode\_state(buf, n)**; /\*由输入字符串查找pm\_states[]指针数组，返回睡眠状态类型\*/

if (state < PM\_SUSPEND\_MAX)

error = **pm\_suspend(state)**; /\*进入睡眠状态，/kernel/power/suspend.c\*/

else if (state == PM\_SUSPEND\_MAX)

error = **hibernate()**; /\*进入冬眠状态（本书暂不介绍），/kernel/power/hibernate.c\*/

else

error = -EINVAL;

out:

pm\_autosleep\_unlock();

return error ? error : n;

}

系统进入"freeze"、"standby"和"mem"睡眠状态的接口函数为pm\_suspend()，函数代码简列如下：

int pm\_suspend(suspend\_state\_t state)

{

int error;

... /\*睡眠状态有效性检查\*/

error = **enter\_state(state)**; /\*进入睡眠状态，/kernel/power/suspend.c\*/

... /\*更新统计量\*/

return error;

}

enter\_state(state)函数调用关系如下图所示，函数内首先冻结用户进程和内核线程（可以冻结的线程），然后使外设进和睡眠状态（详见下一小节），最后将系统切换到"freeze"、"standby"或"mem"睡眠状态。



## 5设备功耗管理

在前面介绍的PM core框架中在进入睡眠状态时会使外设也进入睡眠状态，外设进入睡眠状态需要设备功耗管理数据结构实例的支持。另外，设备驱动程序中也可以单独将自身设为睡眠状态和唤醒。

设备功耗管理通用层代码位于/drivers/base/power/目录下。

### 5.1概述

设备功耗管理的核心数据结构是dev\_pm\_ops结构体，此结构体中定义了设备执行功耗管理的相关操作函数。在通用驱动模型中，设备、设备类型、设备类、总线、设备驱动数据结构中都包含dev\_pm\_ops结构体指针成员，如下图所示。



内核在执行设备功耗管理的操作中将会按照优先级顺序调用实例中的相关函数，优先级顺序为：设备、设备类型、设备类、总线、设备驱动。如果高优先级dev\_pm\_ops实例中的相关函数被调用了，后面低优先级实例中的函数将不再调用。

设备device结构体中内嵌入dev\_pm\_info结构体成员表示了设备功耗管理相关的信息，dev\_pm\_info实例由全局双链表管理。在系统睡眠的操作中将扫描全局链表dev\_pm\_info实例调用，调用设备关联的结构体dev\_pm\_ops实例中的相关函数使设备进入睡眠状态。

设备驱动程序也可以根据自身运行的情况，使自身进入睡眠状态，而不需要使系统进入睡眠状态，这称为运行时功耗管理。

### 5.2数据结构

设备功耗管理中的核心数据结构是dev\_pm\_ops结构体，下面简要介绍一下此结构体的定义以及device结构体中与功耗管理相关的成员。

#### ■dev\_pm\_ops

dev\_pm\_ops结构体定义如下（/include/linux/pm.h）：

struct dev\_pm\_ops {

int (\*prepare)(struct device \*dev);

void (\*complete)(struct device \*dev);

int (\*suspend)(struct device \*dev); /\*系统睡眠时调用\*/

int (\*resume)(struct device \*dev); /\*系统唤醒时调用\*/

int (\*freeze)(struct device \*dev);

int (\*thaw)(struct device \*dev);

int (\*poweroff)(struct device \*dev);

int (\*restore)(struct device \*dev);

int (\*suspend\_late)(struct device \*dev);

int (\*resume\_early)(struct device \*dev);

int (\*freeze\_late)(struct device \*dev);

int (\*thaw\_early)(struct device \*dev);

int (\*poweroff\_late)(struct device \*dev);

int (\*restore\_early)(struct device \*dev);

int (\*suspend\_noirq)(struct device \*dev); /\*完成由suspend()开始的工作\*/

int (\*resume\_noirq)(struct device \*dev); /\*执行resume()函数前的准备工作\*/

int (\*freeze\_noirq)(struct device \*dev);

int (\*thaw\_noirq)(struct device \*dev);

int (\*poweroff\_noirq)(struct device \*dev);

int (\*restore\_noirq)(struct device \*dev);

/\*以下是运行时功耗管理函数，由设备驱动程自身调用\*/

int (\***runtime\_suspend**)(struct device \*dev); /\*设备准备挂起\*/

int (\***runtime\_resume**)(struct device \*dev); /\*设备唤醒\*/

int (**\*runtime\_idle**)(struct device \*dev); /\*设备进入低功耗状态\*/

};

dev\_pm\_ops结构体中主要包含在系统进出睡眠状态对设备的回调函数，以及设备运行时功耗管理的回调函数。

内核在/include/linux/pm.h头文件定义了初始化dev\_pm\_ops实例的宏，例如：

#define SET\_SYSTEM\_SLEEP\_PM\_OPS(suspend\_fn, resume\_fn) \ /\*系统睡眠相关的函数\*/

.suspend = suspend\_fn, \

.resume = resume\_fn, \

.freeze = suspend\_fn, \

.thaw = resume\_fn, \

.poweroff = suspend\_fn, \

.restore = resume\_fn,

#define SET\_RUNTIME\_PM\_OPS(suspend\_fn, resume\_fn, idle\_fn) \ /\*运行时功耗管理相关函数\*/

.runtime\_suspend = suspend\_fn, \

.runtime\_resume = resume\_fn, \

.runtime\_idle = idle\_fn,

在设备device、设备类型device\_type、设备类class、总线bus\_type、设备驱动device\_driver结构体中都包含dev\_pm\_ops结构体指针，但是它们有优先级顺序，只有最高优先级的实例中的函数才会被调用。

#### ■device相关成员

在通用驱动模型中，设备device结构体中与功耗管理相关的成员定义如下：

struct device {

...

**struct dev\_pm\_info power**; /\*dev\_pm\_info结构体成员，/include/linux/pm.h\*/

**struct dev\_power\_domain \*pm\_domain**; /\*提供关机、停机等时候回调函数\*/

...

}

device结构体中与功耗管理相关成员简介如下：

●**power：**dev\_pm\_info结构体成员，定义如下（/include/linux/pm.h）：

struct dev\_pm\_info {

pm\_message\_t power\_state;

unsigned int can\_wakeup:1;

unsigned int async\_suspend:1;

bool is\_prepared:1; /\* Owned by the PM core \*/

bool is\_suspended:1; /\* Ditto \*/

bool is\_noirq\_suspended:1;

bool is\_late\_suspended:1;

bool ignore\_children:1;

bool early\_init:1; /\* Owned by the PM core \*/

bool direct\_complete:1; /\* Owned by the PM core \*/

spinlock\_t lock;

#ifdef CONFIG\_PM\_SLEEP

struct list\_head **entry**; /\*将实例添加到dpm\_list全局双链表\*/

struct completion completion;

struct wakeup\_source \*wakeup;

bool wakeup\_path:1;

bool syscore:1;

#else

unsigned int should\_wakeup:1;

#endif

#ifdef CONFIG\_PM

struct timer\_list suspend\_timer;

unsigned long timer\_expires;

struct work\_struct work;

wait\_queue\_head\_t wait\_queue;

struct wake\_irq \*wakeirq;

atomic\_t usage\_count;

atomic\_t child\_count;

unsigned int disable\_depth:3;

unsigned int idle\_notification:1;

unsigned int request\_pending:1;

unsigned int deferred\_resume:1;

unsigned int run\_wake:1;

unsigned int runtime\_auto:1;

unsigned int no\_callbacks:1;

unsigned int irq\_safe:1;

unsigned int use\_autosuspend:1;

unsigned int timer\_autosuspends:1;

unsigned int memalloc\_noio:1;

enum rpm\_request request;

enum rpm\_status runtime\_status;

int runtime\_error;

int autosuspend\_delay;

unsigned long last\_busy;

unsigned long active\_jiffies;

unsigned long suspended\_jiffies;

unsigned long accounting\_timestamp;

#endif

struct pm\_subsys\_data \*subsys\_data; /\* Owned by the subsystem. \*/

void (\*set\_latency\_tolerance)(struct device \*, s32);

**struct dev\_pm\_qos \*qos**; /\*见9.11.3小节\*/

};

**●pm\_domain：**dev\_power\_domain结构体指针，结构体定义如下（/include/linux/pm.h）：

struct dev\_pm\_domain {

**struct dev\_pm\_ops ops**; /\*dev\_pm\_ops结构体实例\*/

void (\*detach)(struct device \*dev, bool power\_off); /\*设备从pm\_domain移出时调用\*/

int (\*activate)(struct device \*dev); /\*执行总线或驱动的探测函数前调用（匹配设备与驱动时）\*/

void (\*sync)(struct device \*dev); /\*探测函数之后调用\*/

void (\*dismiss)(struct device \*dev); /\*探测函数失败后调用\*/

};

内核在/drivers/base/power/main.c文件内定义了以下全局双链表用于管理**dev\_pm\_info**结构体成员：

LIST\_HEAD(dpm\_list);

static LIST\_HEAD(dpm\_prepared\_list); /\*管理已经准备如挂起的设备\*/

static LIST\_HEAD(dpm\_suspended\_list); /\*管理已经挂起的设备\*/

static LIST\_HEAD(dpm\_late\_early\_list);

static LIST\_HEAD(dpm\_noirq\_list);

在添加设备的**device\_add()**函数中将调用**device\_pm\_add**(dev)函数，函数内将dev\_pm\_info结构体成员（dev->power）添加到全局双链表**dpm\_list**末尾，函数定义如下（/drivers/base/power/main.c）：

void device\_pm\_add(struct device \*dev)

{

...

mutex\_lock(&dpm\_list\_mtx);

...

**list\_add\_tail(&dev->power.entry, &dpm\_list)**;

mutex\_unlock(&dpm\_list\_mtx);

}

### 5.3挂起设备

pm\_message\_t结构体用于PM core等高层向设备功耗管理层传递必要的信息，结构体定义如下：

typedef struct pm\_message {

int event;

} pm\_message\_t;

pm\_message\_t取值定义简列如下：

#define PM\_EVENT\_INVALID (-1)

#define PM\_EVENT\_ON 0x0000

#define PM\_EVENT\_FREEZE 0x0001

#define PM\_EVENT\_SUSPEND 0x0002 /\*系统将挂起\*/

#define PM\_EVENT\_HIBERNATE 0x0004

#define PM\_EVENT\_QUIESCE 0x0008

#define PM\_EVENT\_RESUME 0x0010 /\*系统正在恢复\*/

#define PM\_EVENT\_THAW 0x0020

#define PM\_EVENT\_RESTORE 0x0040

#define PM\_EVENT\_RECOVER 0x0080

#define PM\_EVENT\_USER 0x0100

#define PM\_EVENT\_REMOTE 0x0200

#define PM\_EVENT\_AUTO 0x0400 /\*设备自动功耗管理\*/

在前一小节介绍的系统睡眠操作中将挂起设备，函数调用关系简列如下：



**dpm\_suspend\_start()**函数用于挂起外设，函数定义如下（/drivers/base/power/main.c）：

int dpm\_suspend\_start(pm\_message\_t state)

/\*state：.event = PM\_EVENT\_SUSPEND\*/

{

int error;

error = **dpm\_prepare(state)**; /\*准备挂起，/drivers/base/power/main.c\*/

if (error) {

suspend\_stats.failed\_prepare++;

dpm\_save\_failed\_step(SUSPEND\_PREPARE);

} else

error = **dpm\_suspend(state)**; /\*挂起设备，/drivers/base/power/main.c\*/

return error;

}

dpm\_suspend\_start()函数主要分两步，一是调用dpm\_prepare(state)函数执行准备工作，二是调用函数dpm\_suspend(state)执行挂起操作，下面简要介绍这两个函数的实现。

#### ■准备挂起

dpm\_prepare()准备挂起操作函数定义如下：

int dpm\_prepare(pm\_message\_t state)

{

int error = 0;

...

might\_sleep();

mutex\_lock(&dpm\_list\_mtx);

while (!list\_empty(&**dpm\_list**)) { /\*dpm\_list双链表不为空，执行循环\*/

struct device \*dev = to\_device(dpm\_list.next);

get\_device(dev);

...

error = **device\_prepare(dev, state)**; /\*调用最高优先级dev\_pm\_ops实例中prepare()函数等\*/

...

dev->power.is\_prepared = true;

if (!list\_empty(&dev->power.entry)) /\*移到dpm\_prepared\_list全局双链表\*/

**list\_move\_tail(&dev->power.entry, &dpm\_prepared\_list)**;

put\_device(dev);

}

...

return error;

}

dpm\_prepare()函数扫描dpm\_list双链表中实例，对每个实例调用device\_prepare(dev, state)函数，函数内调用所代表device实例最高优先级dev\_pm\_ops实例中prepare()函数等，然后将dpm\_list双链表中实例移入到dpm\_prepared\_list全局双链表。

#### ■挂起操作

dpm\_suspend()函数执行挂起设备操作，函数定义如下：

int dpm\_suspend(pm\_message\_t state)

{

ktime\_t starttime = ktime\_get();

int error = 0;

...

**cpufreq\_suspend()**; /\*调用cpufreq\_driver->suspend(policy)函数，/drivers/cpufreq/cpufreq.c\*/

mutex\_lock(&dpm\_list\_mtx);

pm\_transition = state;

async\_error = 0;

while (!list\_empty(&**dpm\_prepared\_list**)) { /\*扫描dpm\_prepared\_list全局链表中实例\*/

struct device \*dev = to\_device(dpm\_prepared\_list.prev);

get\_device(dev);

...

error = **device\_suspend(dev)**; /\*执行挂起设备操作，调用dev\_pm\_ops实例中suspend()函数\*/

...

if (!list\_empty(&dev->power.entry))

**list\_move(&dev->power.entry, &dpm\_suspended\_list)**; /\*移动到全局双链表\*/

put\_device(dev);

if (async\_error)

break;

}

mutex\_unlock(&dpm\_list\_mtx);

async\_synchronize\_full();

...

return error;

}

dpm\_suspend()函数扫描dpm\_prepared\_list全局链表中实例，对每个实例调用device\_suspend(dev)函数，函数内调用最高优先级dev\_pm\_ops实例中suspend()函数等，最后将实例移动到dpm\_suspended\_list全局双链表。

### 5.4运行时功耗管理

运行时功耗管理是指单个设备可以在设备空闲时进入挂起状态，而不是要等到系统挂起时才挂起设备，设备的挂起由驱动程序自身控制。

dev\_pm\_ops结构体中与运行时功耗管理相关的成员如下：

struct dev\_pm\_ops {

...

int (\***runtime\_suspend**)(struct device \*dev); /\*设备准备挂起\*/

int (\***runtime\_resume**)(struct device \*dev); /\*设备唤醒\*/

int (**\*runtime\_idle**)(struct device \*dev); /\*设备进入低功耗状态\*/

};

设备运行时功耗管理相关代码位/time.c文件内。

设备运行时功耗管理请求类型定义如下（/include/linux/pm.h）：

enum rpm\_request {

RPM\_REQ\_NONE = 0,

RPM\_REQ\_IDLE, /\*调用runtime\_idle()函数，进入空闲状态\*/

RPM\_REQ\_SUSPEND, /\*调用runtime\_suspend()函数，进入挂起状态\*/

RPM\_REQ\_AUTOSUSPEND,

RPM\_REQ\_RESUME, /\*调用runtime\_resume()函数\*/

};

运行时功耗管理接口函数声明（定义）在/include/linux/pm\_runtime.h头文件，例如：

●void pm\_runtime\_enable(struct device \*dev)：使能运行时功耗管理。

●void \_\_pm\_runtime\_disable(struct device \*dev, bool check\_resume)：关闭运行时功耗管理。

●int pm\_runtime\_idle(struct device \*dev)：调用runtime\_idle()函数。

●int pm\_runtime\_suspend(struct device \*dev)：调用runtime\_suspend()函数。

●int pm\_runtime\_resume(struct device \*dev)：唤醒（恢复）设备。