arch技术研究

[1 armv8简介 2](#_Toc516904767)

[1.1 相关专业名词 2](#_Toc516904768)

[2 Execution State 3](#_Toc516904769)

[2.1 两种Execution State 3](#_Toc516904770)

[2.1.1 AArch32 3](#_Toc516904771)

[2.1.2 AArch64 3](#_Toc516904772)

[2.1.3 决定Execution State的条件 3](#_Toc516904773)

[3 dvfs功能 4](#_Toc516904774)

[4 软件架构 4](#_Toc516904775)

[5 软件模块的功能及API描述 5](#_Toc516904776)

[5.1 cpufreq core 5](#_Toc516904777)

[5.1.1 用户空间接口 5](#_Toc516904778)

[5.2 cpufreq driver 6](#_Toc516904779)

[5.3 cpufreq governors 8](#_Toc516904780)

[5.3.1 常见governors 8](#_Toc516904781)

[6 Interactive governor 8](#_Toc516904782)

[6.1 Slack Timer 8](#_Toc516904783)

[6.2 Cfinteractive线程 9](#_Toc516904784)

[6.3 update load 9](#_Toc516904785)

# armv8简介

ARMv8的架构继承以往ARMv7与之前处理器技术的基础，除了现有的16/32bit的Thumb2指令支持外，也向前兼容现有的A32(ARM 32bit)指令集，基于64bit的AArch64架构，除了新增A64(ARM 64bit)指令集外，也扩充了现有的A32(ARM 32bit)和T32(Thumb2 32bit）指令集，另外还新增加了CRYPTO(加密)模块支持。

## 相关专业名词

AArch32：描述32bit Execution State。

AArch64：描述64bit Execution State

A32、T32：AArch32 ISA （Instruction Architecture）

A64：AArch64 ISA （Instruction Architecture）

Interprocessing：描述AArch32和AArch64两种执行状态之间的切换

SIMD：Single-Instruction, Multiple-Data （单指令多数据）

# Execution State

## 两种Execution State

 ARMv8 提供AArch32 state和 AArch64 state 两种Execution State。

### AArch32

（1）提供13个32bit通用寄存器R0-R12，一个32bit PC指针 (R15)、堆栈指针SP (R13)、链接寄存器LR (R14)

（2）提供一个32bit异常链接寄存器ELR, 用于Hyp mode下的异常返回

（3）提供32个64bit SIMD向量和标量floating-point支持

（4）提供两个指令集A32（32bit）、T32（16/32bit）

（5）兼容ARMv7的异常模型

（6）协处理器只支持CP10\CP11\CP14\CP15

### AArch64

（1）提供31个64bit通用寄存器X0-X30（W0-W30），其中X30是程序链接寄存器LR

（2）提供一个64bit PC指针、堆栈指针SPx 、异常链接寄存器ELRx

（3）提供32个128bit SIMD向量和标量floating-point支持

（4）定义ARMv8异常等级ELx（x<4）,x越大等级越高，权限越大

（5）定义一组PE state寄存器PSTATE（NZCV/DAIF/CurrentEL/SPSel等），用于保存PE当前的状态信息

（6）没有协处理器概念

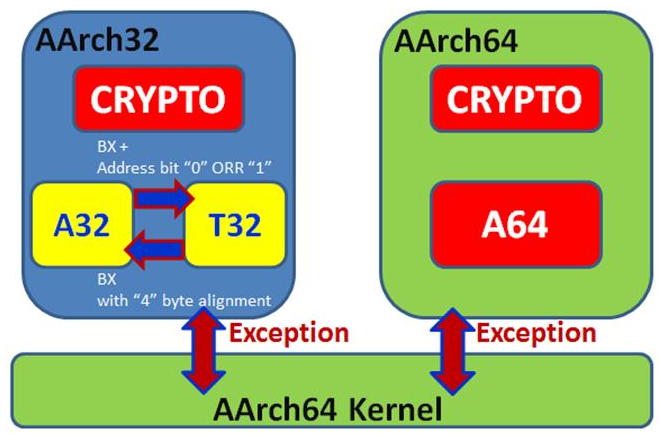
## 决定Execution State的条件

（1）SPSR\_EL1.M[4] 决定EL0的执行状态，为0 =>64bit ,否则=>32bit

（2）HCR\_EL2.RW 决定EL1的执行状态，为1 =>64bit ,否则=>32bit

（3）SCR\_EL3.RW确定EL2 or EL1的执行状态，为1 =>64bit ,否则=>32bit

（4）AArch32和AArch64之间的切换只能通过发生异常或者系统Reset来实现.（A32 -> T32之间是通过BX指令切换的）



# Exception Level

ARMv8定义EL0-EL3共 4个Exception Level来控制PE的行为.

（1）ELx（x<4），x越大等级越高，执行特权越高

（2）执行在EL0称为非特权执行

（3）EL2 没有Secure state，只有Non-secure state

（4）EL3 只有Secure state，实现EL0/EL1的Secure 和Non-secure之间的切换

（5）EL0 & EL1 必须要实现，EL2/EL3则是可选实现

## Exception Level 与Security

|  |  |
| --- | --- |
| Exception Level | |
| EL0 | Application |
| EL1 | Linux，optee |
| EL2 | Hypervisor (可以理解为上面跑多个虚拟OS) |
| EL3 | Secure Monitor(ARM Trusted Firmware) （atf） |
| Security | |
| Non-secure | EL0/EL1/EL2, 只能访问Non-secure memory |
| Secure | EL0/EL1/EL3, 可以访问Non-secure memory & Secure memory,可起到物理屏障安全隔离作用 |

### EL3使用AArch64、AArch32的对比

# dvfs功能

实现原理：通过动态调整cpu的电压和频率达到性能和功耗的平衡点。

实现dvfs，两个关键点：

其一：如何控制cpu core的电压和频率

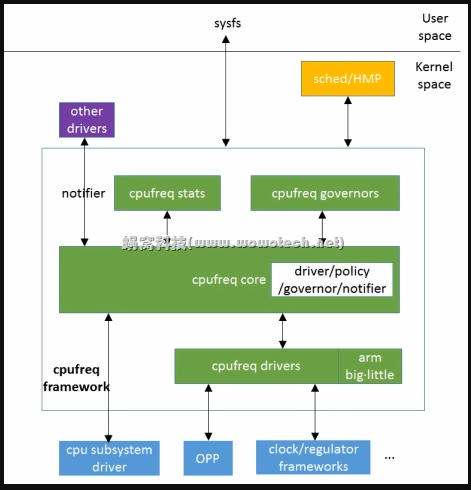
其二：何时改变cpu core的电压和频率

内核针对上述两关键点，实现两种方式：

（1）cpu core根据自身的负载，自动调整电压和频率，不需要os级别的软件参与。

（2）cpu core不参与任何的逻辑动作，由os软件根据系统运行情况，调整电压和频率、为实现上述功能要求，cpufreq framework抽象出cpufreq driver、cpufreq policy、cpufreq governor等软件实体。

# 软件架构



对下，cpufreq framework基于cpu subsystem driver、OPP、clock framework、regulator framework等模块，提供对CPU core频率和电压的控制。这一部分主要由cpufreq driver实现。

对上，cpufreq framework会通过cpufreq core、cpufreq governors、cpufreq stats等模块，以sysfs的形式，向用户空间提供cpu frequency的查询、控制等接口。同时，在频率改变的时候，通过notifier通知关心的driver内部，cpufreq framework包括cpufreq core、cpufreq driver、cpufreq governors、cpufreq stats等模块。

# 软件模块的功能及API描述

## cpufreq core

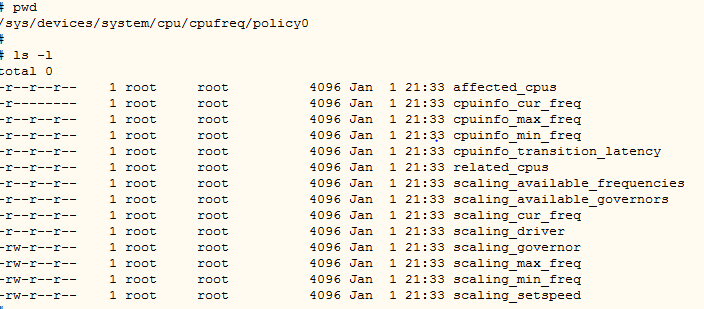
cpufreq core是cpufreq framework的核心模块，和kernel其它framework类似，它主要实现三类功能：

（1）对上，以sysfs的形式向用户空间提供统一的接口，以notifier的形式向其它driver提供频率变化的通知。

（2）对下，提供CPU core频率和电压控制的驱动框架，方便底层driver的开发；同时，提供governor框架，用于实现不同的频率调整机制。

（3）内部，封装各种逻辑，实现所需功能。这些逻辑主要围绕struct cpufreq\_driver、struct cpufreq\_policy和struct cpufreq\_governor三个数据结构进行。

### 用户空间接口



cpufreq driver初始化时，会根据frequency table等信息，填充struct cpufreq\_policy变量中的struct cpufreq\_cpuinfo变量，该变量保存了CPU调频有关的固定信息，不可以在运行过程中修改，主要包括：最大频率（cpuinfo\_max\_freq）、最小频率（cpuinfo\_min\_freq）、频率转换延迟（cpuinfo\_transition\_latency ）。

cpuinfo\_cur\_freq：获取cpu core的当前频率（真实的、cpu的当前运行频率，会通过cpufreq\_driver->get回调读取）。

scaling\_available\_frequencies：获取当前可以配置的频率列表，从frequency table直接读取。readonly。

scaling\_driver：当前加载的cpufreq driver名称，readonly。

scaling\_available\_governors和scaling\_governor：系统中可用的governor列表，以及当前使用的governor。readonly。

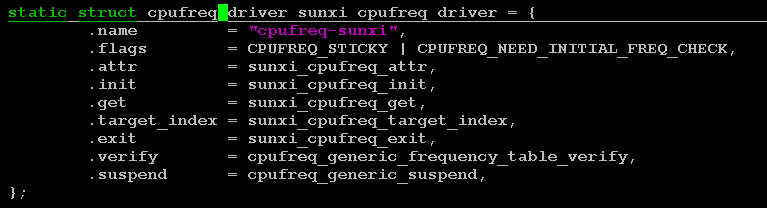
scaling\_cur\_freq：从cpufreq core或者governor的角度，看到的当前频率，和cpuinfo\_cur\_freq的意义不相同。readonly。

scaling\_max\_freq、scaling\_min\_freq和scaling\_setspeed：scaling\_max\_freq和scaling\_min\_freq表示调频策略所允许的最大和最小频率，对于可以自动调整频率的cpu，修改它们，就是最终的频率调整。

scaling\_setspeed：如果使用的governor是“userspace” governor，则可以通过scaling\_setspeed节点，直接修改cpu频率。

## cpufreq driver

1. 定义一个struct cpufreq\_driver变量，填充必要的字段，并根据平台的特性，实现其中的回调函数。



（1）init回调函数是cpufreq driver的入口，由cpufreq core在CPU device添加之后调用，其主要功能就是初始化policy变量

（2）verify回调函数，当上层软件需要设定一个新的policy的时候，会调用driver的verify回调函数，检查该policy是否合法。

（3）target\_index回调函数，对于不可以自动调频的CPU，该接口用于指定CPU的运行频率。index表示frequency table中的index。driver需要通过index，将频率值取出，通过clock framework提供的API，将CPU的频率设置为对应的值。同时，driver可以调用OPP interface，获取该频率对应的电压值，通过regulator framework提供的API，将CPU的电压设置为对应的值。

（4）get回调函数，用于获取指定cpu的频率值，如果可以的话，driver应尽可能提供。如果在init接口中给policy->clk赋值的话，则可以使用cpufreq framework提供的通用接口。

（5）attr，如果cpufreq driver需要提供一些额外的sysfs attribute，可以通过如下的attribute宏设置，然后保存在cpufreq\_driver的attr数组中。

（6）exit，和init对应，在CPU device被remove时调用。

（7）suspend、resume回调函数。系统给suspend的时候，clock、regulator等driver有可能被suspend，因此需要在这之前将CPU设置为一个确定的频率值。driver可以通过suspend回调设置，也可以通过policy中的suspend\_freq字段设置（cpufreq core会自动切换）。同理，系统resume后，CPU的运行频率是什么，可以通过resume回调设置，也可以通过policy中的restore\_freq字段设置。

（8）cpufreq\_driver flags

CPUFREQ\_STICKY，表示就算所有的init调用都失败了，driver也不被remove。具体应用场景不明。

CPUFREQ\_CONST\_LOOPS，表示频率的调整，不影响loops\_per\_jiffy等kernel常来的计算。

CPUFREQ\_PM\_NO\_WARN，suspend/resume过程相关的flag。

CPUFREQ\_HAVE\_GOVERNOR\_PER\_POLICY，表示不同的CPU，有不同的频率控制方式，因此cpufreq core会为每一个CPU创建一个cpufreq调频接口。否则（也是正常情况下），一个调频接口可以调整所有CPU的频率。

2. 调用cpufreq\_register\_driver将driver注册到cpufreq framework中

## cpufreq governors

cpufreq policy负责设定cpu调频的一个大致范围，而cpu的具体运行频率，则需要由相应的cufreq governor决定。

### 常见governors

1）Performance

性能优先的governor，直接将cpu频率设置为policy->{min,max}中的最大值。

2）Powersave

功耗优先的governor，直接将cpu频率设置为policy->{min,max}中的最小值。

3）Userspace

由用户空间程序通过scaling\_setspeed文件修改频率。

4）Ondemand

根据CPU的当前使用率，动态的调节CPU频率。

5）Conservative

类似Ondemand，不过频率调节的会平滑一下，不会忽然调整为最大值，又忽然调整为最小值。

# Interactive governor

## Slack Timer

1. per cpu变量，governor初始化时申请，callback函数为空。作用：定时中断系统，从idle唤醒。
2. 启动timer时机：

* cpufreq\_ineractive\_start：当启动一个新的governor时，通过cpufreq\_ineractive\_start—>slack\_timer\_resched->gov\_slack\_timer\_start函数调用，per cpu启动timer，timer的timer expires为DEFAULT\_TIMER\_SLACK（80s）+DEFAULT\_SAMPLING\_RATE（20s）。
* Cpufreq\_interactive\_update：该入口又分为两种情况
  + 系统退出idle的时候，通过idle notifier，当状态为IDLE\_END时调用cpufreq\_interactive\_idle\_end->cpufreq\_interactive\_update启动timer，同时调用eval\_target\_freq计算系统负载。
  + 启动governor时，percpu通过cpufreq\_add\_update\_util\_hook注册一个hook函数update\_util\_handler，该函数调用irq\_work\_queue将irq\_work任务添加到队列，在irq\_work函数中会调用Cpufreq\_interactive\_update函数。

update\_util\_handler调用时机：当 cfs, rt, deadline 3 个调度类的 capacity 出现变化时，调用 cpufreq\_update\_util() 来触发 hook，实现类似 notifier 的效果。

1. cpufreq\_interactive\_update

cpufreq\_interactive\_update

eval\_target\_freq

update\_load

slack\_timer\_resched

## Cfinteractive线程

（1）注册时机：cpufreq\_interactive\_gov\_init初始化不但为每个cpu注册了单独的slack timer，同时还建立了一条线程cfinteractive，线程处理函数为cpufreq\_interactive\_speedchange\_task。

（2）线程作用：更新freq；更新负载

（3）线程唤醒时机：其一，eval\_target\_freq评估完需要唤醒线程；其二，store\_boost->cpufreq\_interactive\_boost来唤醒。

## update load

（1）当系统cfs，rt，deadline调度类的capacity出现变化，调用cpufreq\_update\_util来触发hook函数update\_util\_handler，通过调用eval\_target\_freq->update\_load来更新负载。

（2）通过线程Cfinteractive调用notifier (cpufreq\_notifier\_block)callback函数cpufreq\_interactive\_notifier->update\_load更新负载。流程如下

cpufreq\_interactive\_speedchange\_task

cpufreq\_interactive\_adjust\_cpu

\_\_cpufreq\_driver\_target

\_\_target\_index

cpufreq\_freq\_transition\_end

cpufreq\_notify\_post\_transition

cpufreq\_notify\_transition

srcu\_notifier\_call\_chain

\_\_srcu\_notifier\_call\_chain

notifier\_call\_chain

cpufreq\_notifier\_block这个notifier\_block是在cpufreq\_interactive\_init初始化的时候注册的，同时注册的还有cpufreq\_interactive\_idle\_nb，这个是退出idle状态启动timer并且更新负载（eval\_target\_freq）用的（5.1有提到）。