《openEuler内核编程》

课程讲稿

第5章 第1讲

Kunpeng架构下的中断与异常机制

软件所制

第五章 第一讲 Kunpeng架构下的中断与异常机制

**学时：**1学时

**教学目的：** 了解虚拟文件系统的特点，了解和虚拟文件系统相关的系统调用的实现。了解虚拟文件系统的数据结构

**课程时间线：**

章节简介

异常与中断基本概念

ARM体系的异常

ARM体系的中断

5min

15min

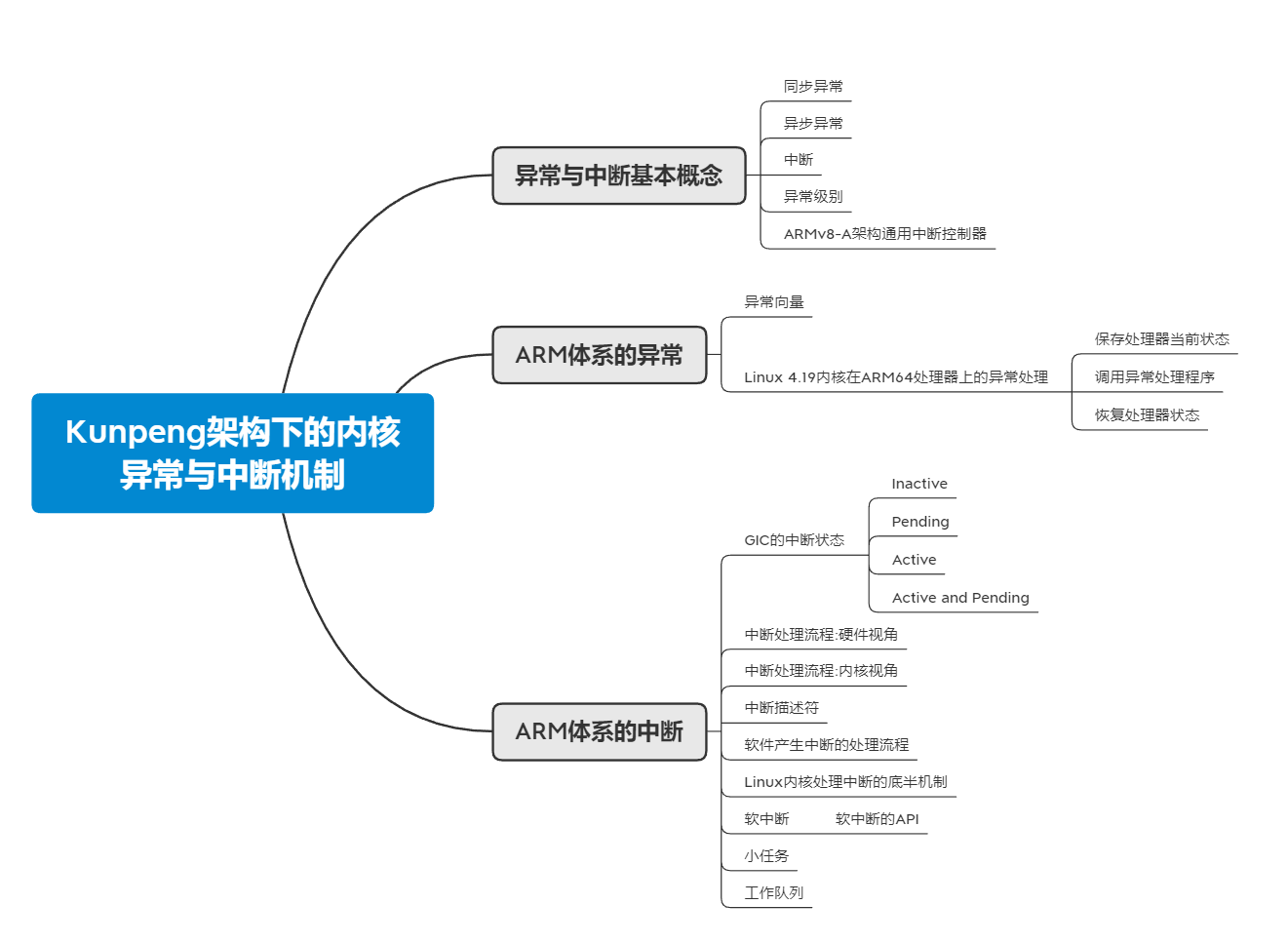
35min

45 min

**课外参考读物：**

《Linux内核设计与实现》

**知识框图：**

****

**PPT讲稿：**



今天就进入关于中断和异常的学习阶段了。



本章共分为以下6个课时的内容，是循序渐进的。



这一节一共有一课时内容，和大家讲一下，然后讲一下Kunpeng架构下的内核异常与中断机制。



在计算机术语中，异常通常指在程序运行过程中发生的异常事件，ARM64体系中的异常可以分为同步异常和异步异常。



同步异常是由正在运行的指令或指令运行的结果造成的异常，以下行为可能会导致同步异常的触发：

1）执行异常生成指令：例如 Supervisor Call （SVC）、 Hypervisor Call （HVC）、

Secure monitor Call （SMC）指令将触发同步异常

2 ）数据异常：例如，访问数据时如果没有读 写权限或虚拟地址未映射到物理地址，

将触发数据异常；

3 ）指令异常：例如，取指时如果没有执行权限或虚拟地址未映射到物理地址，将触

发指令异常；

4 ）未定义的指令：如果 C PU 遇到无法识别的指令，将触发同步异常；

5 ）调试异常：在调试过程中，单步执行过程将产生调试异常。

异步异常不由当前执行的指令触发，可分为三种异步异常：

IRQ （Interrupt ReQuest）、FIQ （Fast Interrupt reQuset）和 SError（系统错误）。SError指由硬件错误触发的异常，通常与外部异步数据异常相关。例如，若误将 ROM对应的区域的页表项设置为可读写，当某个操作触发将 CPU cache中的脏数据写回到内存时，内存系统就会返回错误（因为 ROM是只读的），从而触发一个异步数据异常 。IRQ和 FIQ是通用的中断类型。

在通常情况下，中断指某些类型的异步异常，如 IRQ和 FIQ。 FIQ的优先级高于 IRQ。

ARMv8架构 中 ，异常 exception级别 又 称特权为级别，ARMv8架构的 异常模型包含 4个异常级别，分别是 EL0~EL3。在这些异常级别中， EL0的异常 级别最低， EL3的异常级别最 高 。运行在更高异常级别的程序 对硬件的 控制 权限 、寄存器的访问权限以及指令的执行权限也越高。应用程序运行在异常级别EL0，因此，EL0 也被称为用户模式。。操作系统内核则运行在异常级别EL1，因此，EL1 也被称为内核模式。虚拟机监视器（Hypervisor）程序通常运行在异常级别EL2；而底层控件和安全管理程序则运行在异常级别EL3。

在异常产生时 以及异常处理 结束 时这两种情况下，将发生异常级别的切换。CPU当前的异常级别由时。 ARMv8架构中的 CurrentEL寄存器保存，它的值 在异常产生或 异常处理结束时由硬件自动改变。对于软件而言 CurrentEL寄存器是只读的。异常可能由缺页错误、除零错误等原因被动触发，也可能由程序 执行专用 的 指令 而 主动 触发。在 ARMv8架构中， SVC、 HVC和 SMC三条 指令用于主动触发异常。这三条指令通 常封装在 高异常级别程序 提供给低异常级别程序的调用接口中 使得处理器可以切换到高异 常级别，从而 安全地执行低异常级别程序所不能执行的敏感操作。SVC指令将触发异常级别从 EL0到 EL1的切换。该指令 通常封装在 操作系统内核 向 应用程序 提供的 API接口中。 HVC指令将触发异常级别从 EL1到 EL2的切换。该指 令通常封装在 虚拟机监视器程序 向 操作系统程序 提供的接口中 。 SMC指令将触发异常级别从 EL2到 EL3的切换。 该指令通常封装在 安全监视器程序（ EL3） 向 操作系统或虚拟机监视器 提供的接口中 。

中断是外设给处理器传递信息的一种方式，外设中断相对于处理器通常是异步的。ARM64体系中外设中断是通过通用中断控制器转发给处理器的。通用中断控制器是支持和管理中断的硬件部件，它提供了用于管理中断源、中断行为和将中断路由至一个或多个处理器的寄存器。GIC定义了四种外设中断：

Locality-specific Peripheral Interrupt （LPI）：一种基于消息的边沿中断。该中断不需要通过中断线发送中断信号，而通过总线写GIC中的寄存器来发送中断信息。

Private Peripheral Interrupt （PPI）：发送给特定处理器的外设中断，不同的处理器对于不同事件可以用相同的中断号（一个中断可以用一个ID号表示，称为中断ID号）。

Shared Peripheral Interrupt （SPI）：GIC将该种类型的中断路由给能处理该中断的处理器，不限于特定的处理器。

Software Generated Interrupt （SGI）：该种类型的中断用于处理器间通信，通过写GIC中的SGI寄存器产生。

从图中，我们可以看到，GICv3由Distributor、Redistributor、CPU Interface和ITS四个部分组成，它们的功能是：Distributor：为SPI提供路由配置并保存相关的路由信息和优先级信息；Redistributor：为PPI提供配置，将处于pending状态的最高优先级的中断在有限时间内发送给CPU接口；CPU Interface：CPU Interface提供一个连接处理器（Processing Element， PE）与GIC的接口；ITS（Interrupt Translation Service）：将事件转化为LPI（可选的）。

我们接着看GIC是怎么处理物理中断的，第一步外设或软件产生中断；第二步设定中断的分组和优先级，并路由至CPU接口（CPU接口提供了处理器和GIC连接的接口）；第三步CPU接口将中断传给其对应的处理器；第四步当处理器得知中断的存在时，GIC将被激活的中断设置为最高优先级，从而对SPI、PPI和SGI来说中断变成激活态；第五步处理器上运行的软件向GIC发送信号表明最高优先级的中断已经可以降级为被处理前的优先级状态了；第六步清除中断的激活状态，可以被配置为和Priority Drop同时发生。

接下来我们看一下ARM体系的异常。

存储异常处理程序的内存位置称为异常向量，通常把所有异常向量放在一张表中，该表称为异常向量表。ARM64处理器的异常级别EL1、EL2和EL3分别有自己的异常向量表，异常向量表的起始虚拟地址则放在向量基准地址寄存器VBAR\_ELn中。ARM64异常级别n的异常向量表如右图所示。

当异常发生时，处理器需要调用异常处理程序来处理异常，该调用过程可以粗略地分为保存处理器当前状态、调用异常处理程序和恢复异常发生前的处理器状态三步



第一步保存处理器当前状态，将当前处理器状态PSTATE保存在SPSR\_EL1寄存器中，将返回地址保存在ELR\_EL1寄存器中，这两个寄存器中的变量会在返回时被eret指令用于恢复处理器状态。通过设置处理器状态中的调试掩码位D、系统错误掩码位A、中断掩码位I和快速中断掩码位F禁止调试异常、系统错误异常、外部中断和快速中断。将发生错误的原因保存在ESR\_EL1寄存器中，将同步异常的错误地址保存在FAR\_EL1寄存器中。如果处理器处于异常级别EL0则将异常级别提升到EL1。根据异常向量表基地址、生成异常的异常级别和异常类型计算出异常向量的位置，通过异常向量跳转到异常处理程序的入口。异常向量表的基地址是保存在VBAR\_EL1寄存器中的。

第二步调用异常处理程序。以异常级别EL0下64位应用程序发生的同步异常为例，其异常处理程序入口为sync，异常向量表通过kernel\_ventry宏跳转到了该入口。kernel\_ventry在跳转的时候会为将异常级别加到跳转入口之前，所以这时实际跳转到的入口是el0\_sync，其汇编代码在openeuler/kernel/blob/kernel-4.19/arch/arm64/kernel/entry.S文件中可以找到。el0\_sync首先通过kernel\_entry宏保存了异常处理前的通用寄存器状态，然后从ESR\_EL1寄存器中读取了错误的原因，移位之后和各种异常的标志值相比较，若与某一异常的标志值相同则跳转到该异常的处理程序。kernel\_entry宏会将通用寄存器的值保存在当前进程的内核栈中。跳转到处理程序后将会进行异常处理，以异常级别EL0下的数据中止异常为例，其处理程序的汇编代码在同一个文件中可以找到。



跳转到处理程序后将会进行异常处理，以异常级别EL0下的数据中止异常为例，其处理程序的汇编代码在同一个文件中可以找到。在这段代码中，处理器从FAR\_EL1寄存器中读取了数据中止发生的虚拟地址，然后调用了C程序处理函数do\_mem\_abort并通过寄存器X0和X1传递了两个参数，其中x0中保存了错误发生的虚拟地址，X1中保存了错误发生的原因（el0\_sync宏中从ELR\_EL1寄存器读到了X25中）。



第三步恢复处理器状态，继续执行程序。异常处理程序执行完之后跳转到了ret\_to\_user。这段代码在最后调用了kernel\_exit宏，其作用是将之前kernel\_entry中保存的通用寄存器恢复，并使用eret指令返回异常发生前的程序执行位置，该位置的地址是被事先保存在ELR\_EL1寄存器中的，其取值情况为：对于系统调用，返回地址为系统调用指令后面第一条指令的地址；对于系统调用外的同步异常，返回地址为生成异常的指令，因为该指令需要被重新执行；对于异步异常，返回地址为没执行的第一条指令。

接下来我们讲ARM体系的中断。

在前面我们介绍GICv3。ARMv8-A架构的处理器在接收外设中断时，外设中断会先通过通用中断控制器（General Interrupt Controller， GIC）进行转发。GIC会根据中断的类型将中断通过CPU接口转发至合适的CPU进行处理。GIC为每一个所支持的中断都维护了状态，这些状态共有四种：

Inactive：中断既不在active状态也不在pending状态，中断源没有发送中断；

Pending：处理器确认硬件或软件产生了中断，中断等待相关处理器的处理；

Active：处理器确认并正在处理一个中断，因此另一个相同中断源发送的中断只有在该中断不再处于Active状态后才会被处理器接受；

Active and Pending:处理器正在处理中断，相同的中断源又发送了一个中断。



中断的四种状态之间可以互相转换，其状态转换关系图如右图所示，各个状态间的转移条件为：状态转移A1和A2：当外设或软件产生中断时发生该状态转移，中断增加了pending状态；状态转移B1和B2: 当外设取消一个电平敏感的中断或软件改变中断的pending状态时发生此状态转移，中断的pending状态被移除了；状态转移C: 当处理器确认边沿敏感的SPI、SGI和PPI时发生此状态转移，中断的pending状态变为了active状态；

状态转移D: 当处理器确认电平敏感的SPI、SGI和PPI时发生此状态转移，中断的pending状态变为了active and pending状态；状态转移E1和E2: 当软件使SPI、SGI和PPI失活时发生此状态转移，active状态被移除了；



接下来我们首先从ARM处理器和GIC的视角来介绍中断处理的过程。当异常在程序执行过程中发生时，首先将PC寄存器和处理器状态PSTATE分别保存在寄存器ELR\_EL1和SPSR\_EL1中，然后调用汇编语言IRQ处理程序。汇编语言IRQ处理程序首先保存可能被污染的寄存器，将PC保存在LR中便于子程序返回，然后调用C语言子程序并执行中断处理程序。调用完C语言子程序后返回汇编语言处理程序并恢复之前保存的可能被污染的寄存器寄存器，然后恢复PC寄存器和之前保存的处理器状态并继续程序的执行。

如果发生嵌套中断，需要将SPSR\_EL1和ELR\_EL1的内容保存在内存栈上，还需要保存被中断的中断处理程序的相关信息（可能被污染的寄存器）。在处理器处理中断的时候，它首先会通过中断向量跳转到顶层中断处理程序并执行。顶层中断处理程序会先读取CPU接口中的中断确认寄存器（Interrupt Acknowledge Register）并得到中断号。当返回中断号时，分配器（Distributor）中的中断状态会变成active。当中断号确定之后，顶层中断处理程序会调用和该外设相关的中断处理程序来处理该中断。当外设相关的中断处理程序执行完之后，顶层中断处理程序将会把中断号写入CPU接口中的中断结束寄存器（End of Interrupt （EoI） register）来表示中断处理流程的结束。中断状态将会从active变为inactive或从active and pending变为pending从而完成了单个中断的处理过程。

从上面讲的异常向量表中，我们可以看到，当发生IRQ中断时，内核会通过异常向量表中的IRQ相关项进入中断处理流程，我们以64位用户程序在异常级别EL0运行时的中断处理流程为例，程序通过异常向量表跳转到entry.S文件的el0\_irq处执行汇编代码，这段代码使用kernel\_entry宏将进程的寄存器值保存到内核栈中，然后调用了irq\_handler宏，最后跳转到ret\_to\_user处使用内核栈保存的寄存器值恢复进程的寄存器并返回用户模式。



在entry.S 文件中我们可以找到irq\_handler宏的汇编代码，这段代码通过irq\_stack\_entry将进程从内核栈切换到中断栈，并调用函数指针handle\_arch\_irq所指向的函数，然后通过irq\_stack\_exit从中断栈切换到内核栈。handle\_arch\_irq会在GIC初始化时被设置为gic\_handle\_irq，设置handle\_arch\_irq的函数调用过程为gic\_of\_init（）->gic\_init\_bases（）->set\_handle\_irq（）。



GICv3版本的gic\_handle\_irq可以在irq-gic-v3.c文件中找到，该函数首先读取处理器接口中的中断确认寄存器得到硬件中断号。当硬件中断号大于15且小于1020或硬件中断号大于或等于8192时，中断来自外设，则调用函数handle\_domain\_irq（）。handle\_domain\_irq（）函数实际上是调用了\_\_handle\_domain\_irq（）函数，首先调用函数irq\_enter（）进入中断上下文，接着调用函数irq\_find\_mapping（）根据硬件中断号查找Linux中断号[1]，然后调用generic\_handle\_irq（）函数处理中断，最后调用irq\_exit（）函数退出中断上下文。generic\_handle\_irq（）函数在irqdesc.c文件中也可以找到，它通过Linux中断号找到了对应的中断描述符，并通过generic\_handle\_irq\_desc（）函数调用中断描述符对应的handle\_irq（）函数。



中断描述符在ARM Linux中由结构体irq\_desc来表示，其代码在openEuler源码仓库的/openeuler/kernel/blob/kernel- 4.19/include/linux/irqdesc.h文件中可以找到。该结构体中定义了两层中断处理函数[1]：第一层中断处理函数是handle\_irq，irq\_flow\_handler\_t是一个函数指针；第二层中断处理函数保存在结构体irqaction组成的链表里，每一个链表元素都保存了一个设备驱动注册的中断处理函数。因为多个硬件设备可能共享同一个硬件中断号，所以irqaction组成的链表里可能有多个元素。



irq\_flow\_handler\_t的定义代码可以在openEuler源码仓库的/openeuler/kernel/blob/kernel-4.19/include/linux/irqhandler.h文件中可以找到。该函数指针指向的函数输入为指向结构体irq\_desc的指针类型，输出为void类型。

我们在openEuler源码仓库的/openeuler/kernel/blob/kernel-4.19/include/linux/irq.h文件中可以找到多个符合irq\_flow\_handler\_t类型的替换函数：当硬件中断号小于16时中断为软件产生的中断（SGI），中断处理程序调用handle\_IPI（）函数处理；当硬件中断号大于或等于16而小于32时中断为私有外设中断（PPI），handle\_irq（）函数被设置为handle\_percpu\_devid\_irq（）；

当硬件中断号大于等于32但小于gic\_data.irq\_nr（该值在irq-gic-v3.c文件中的gic\_init\_bases（）函数中初始化，其最大值为1020，表示GICv3最多支持1020个中断源（SGI+PPI+SPI））时中断为共享外设中断，handle\_irq（）被设置为handle\_fasteoi\_irq（）；当硬件中断号大于等于8192但小于GIC\_ID\_NR时中断为局部特定外设中断（LPI），handle\_irq（）被设置为handle\_fasteoi\_irq（）。

irqaction结构体的代码在openEuler源码仓库的/openeuler/kernel/blob/kernel-4.19/include/linux/interrupt.h文件中可以找到。这个结构体通过next指针构成了链表，从而可以保存多个设备驱动注册的处理函数并使多个设备共享同一个硬件中断号。

接下来我们介绍一下ARM Linux内核中软件产生的中断。软件产生的中断（SGI）是ARM GICv3支持的一种中断类型，SGI通常用于处理器间的通行，通过写GIC的SGI寄存器能产生SGI。ARM Linux内核将硬件中断号小于16的中断识别为SGI，并调用handle\_IPI（）函数处理



ARM Linux内核支持的SGI有如下几种类型：IPI\_RESCHEDULE：函数smp\_send\_reschedule()生成的中断，该中断重新调度进程，scheduler\_ipi()函数调用sched\_ttwu\_pending()函数唤醒pending的任务然后调用raise\_softirq\_irqoff（）函数发起一个软中断。



IPI\_CALL\_FUNC：函数smp\_call\_function()生成的中断，通过调用函数generic\_smp\_call\_function\_interrupt()最终调用函数flush\_smp\_call\_function\_queue（），该函数调用所有在队列中pending的回调函数。



IPI\_CPU\_STOP：函数smp\_send\_stop()生成的中断，调用函数ipi\_cpu\_stop()。函数ipi\_cpu\_stop()将CPU标记为offline状态，将DAIF标志位置一，然后循环调用cpu\_relax()。 cpu\_relax()包含一个内存屏障和一个yield指令，可以降低处理器功耗并且将系统资源让渡给其它线程。



IPI\_CPU\_CRASH\_STOP:该中断使处理器停止；

IPI\_TIMER:该中断广播时钟事件；

IPI\_IRQ\_WORK:在中断上下文中执行回调函数，irq\_work\_run()函数的源码可以在openEuler源码仓库的openeuler/kernel/blob/kernel-4.19/kernel/irq\_work.c文件中可以找到；

IPI\_WAKEUP：唤醒处理器；

IPI\_CPU\_BACKTRACE：打印函数调用栈



接下来我们讲一下Linux内核处理中断的底半机制，Linux系统内核在处理中断时，为避免复杂的中断嵌套，通常采用关闭中断的策略，但是系统关闭中断的时间不能过长，这是因为用户敲击鼠标和键盘等外设中断的响应时间关系到用户体验。为了减少关闭中断的时间，Linux系统内核将中断的处理按重要性分为两个部分：上半部分和下半部分。中断的上半部分又称为硬中断，Linux内核在关闭中断的情况下处理上半部分的工作，它包括中断中对响应时间要求较高或与硬件相关的部分；中断的下半部分则包括软中断（softirq）、小任务（=小任务）和工作队列（work queue）三种类型，Linux内核在开中断的情况下处理中断的下半部分。



三种类型的中断下半部分存在区别，软中断和小任务的处理函数中不允许睡眠，而工作队列的处理函数运行在内核线程上，可以睡眠。可重入函数指该函数的不同副本可以由多个线程同时运行，该函数要么没有使用共享资源从而有相互分离的函数栈，要么使用锁保护了临界区中的共享资源。同一种软中断的处理函数可能会在多个处理器上执行，从而要求处理函数是可重入的。同一个小任务只能在一个处理器上执行，从而不要求函数是可重入的。在后文我们将看到，工作队列添加工作项时，将会把正在运行的工作留在原工作池，以保证任务的不可重入性。软中断的种类是在静态编译时确定的，因此不能动态添加和删除。而小任务和工作队列中的工作项都是可以动态添加和删除的。



软中断在中断的下半部分执行，可以被硬中断抢占。我们要明确一个概念，软中断并不等于软件产生的中断，前者是中断处理过程中下半部分处理非紧急任务的一种方式，后者又被称为处理器间中断，通常用于处理器间的通信。此外，软中断产生后并不是马上可以执行，必须要等待内核的调度才能执行。软中断不能被自己打断(即单个cpu上软中断不能嵌套执行)，只能被硬件中断打断（上半部）。



openEuler 内核中定义了10种软中断，定义代码可以在 openEuler 源码仓库的 openeuler/kernel/blob/kernel-4.19/include/linux/interrupt.h 文件中找到。枚举类型中软中断的编号也表现了它们的优先级，编号较小的软中断优先级较高。软中断首先定义了软中断的注册表，用来记录软中断号和软中断处理函数的对应关系，它是一个 softirq\_action 结构体的数组。void(\*action)是软中断触发时的执行函数。该函数指针的注册在open\_softirq（）函数中完成。



软中断的注册涉及注册对应类型的处理函数到全局数组softirq\_vec中。例如网络发包对应类型为NET\_TX\_SOFTIRQ的处理函数net\_tx\_action.，注册过程在open\_softirq（）函数中完成。软中断的触发是通过之前介绍的 raise\_softirq() 函数完成的， pending 变量中的第n位被置为1表示软中断号为n的软中断被触发了。



执行软中断处理函数\_\_do\_softirq前首先要满足两个条件:

(1)不在中断中(硬中断、软中断和NMI) 。1

(2)有软中断处于pending状态。

系统这么设计是为了避免软件中断在中断嵌套中被调用，并且达到在单个CPU上软件中断不能被重入的目的。对于ARM架构的CPU不存在中断嵌套中调用软件中断的问题，因为ARM架构的CPU在处理硬件中断的过程中是关闭掉中断的。只有在进入了软中断处理过程中之后才会开启硬件中断，如果在软件中断处理过程中有硬件中断嵌套，也不会再次调用软中断，because硬件中断是软件中断处理过程中再次进入的，此时preempt\_count已经记录了软件中断！对于其它架构的CPU，有可能在触发调用软件中断前，也就是还在处理硬件中断的时候，就已经开启了硬件中断，可能会发生中断嵌套，在中断嵌套中是不允许调用软件中断处理的。Why？我的理解是，在发生中断嵌套的时候，表明这个时候是系统突发繁忙的时候，内核第一要务就是赶紧把中断中的事情处理完成，退出中断嵌套。避免多次嵌套，哪里有时间处理软件中断，所以把软件中断推迟到了所有中断处理完成的时候才能触发软件中断。

由于软中断必须使用可重入函数，这就导致设计上的复杂度变高，作为设备驱动程序的开发者来说，增加了负担。而如果某种应用并不需要在多个CPU上并行执行，那么软中断其实是没有必要的。因此诞生了弥补以上两个要求的小任务。它具有以下特性：

a）一种特定类型的小任务只能运行在一个CPU上，不能并行，只能串行执行。

b）多个不同类型的小任务可以并行在多个CPU上。

c）软中断是静态分配的，在内核编译好之后，就不能改变。但小任务就灵活许多，可以在运行时改变（比如添加模块时）。

小任务是在两种软中断类型的基础上实现的，因此如果不需要软中断的并行特性，小任务就是最好的选择。也就是说小任务是软中断的一种特殊用法，即延迟情况下的串行执行。



我们看一下小任务相关的数据结构，小任务描述符和小任务链表



我们看一下小任务相关的操作，如何创建和操作小任务



从上面的介绍看以看出，软中断运行在中断上下文中，因此不能阻塞和睡眠，而tasklet使用软中断实现，当然也不能阻塞和睡眠。但如果某延迟处理函数需要睡眠或者阻塞呢？没关系工作队列就可以如您所愿了。 把推后执行的任务叫做工作（work），描述它的数据结构为work\_struct ，这些工作以队列结构组织成工作队列（workqueue），其数据结构为workqueue\_struct ，而工作线程就是负责执行工作队列中的工作。系统默认的工作者线程为events。 工作队列(work queue)是另外一种将工作推后执行的形式。工作队列可以把工作推后，交由一个内核线程去执行—这个下半部分总是会在进程上下文执行，但由于是内核线程，其不能访问用户空间。最重要特点的就是工作队列允许重新调度甚至是睡眠。 实际上，工作队列的本质就是将工作交给内核线程处理，因此其可以用内核线程替换。但是内核线程的创建和销毁对编程者的要求较高，而工作队列实现了内核线程的封装，不易出错，所以我们也推荐使用工作队列。



通常，在工作队列和软中断/tasklet中作出选择非常容易。可使用以下规则： - 如果推后执行的任务需要睡眠，那么只能选择工作队列。 - 如果推后执行的任务需要延时指定的时间再触发，那么使用工作队列，因为其可以利用timer延时(内核定时器实现)。 - 如果推后执行的任务需要在一个tick之内处理，则使用软中断或tasklet，因为其可以抢占普通进程和内核线程，同时不可睡眠。 - 如果推后执行的任务对延迟的时间没有任何要求，则使用工作队列，此时通常为无关紧要的任务。 实际上，工作队列的本质就是将工作交给内核线程处理，因此其可以用内核线程替换。但是内核线程的创建和销毁对编程者的要求较高，而工作队列实现了内核线程的封装，不易出错，所以我们也推荐使用工作队列。



我们看一下工作队列相关的数据结构，工作结构体和工作队列结构体



我们看一下小任务相关的操作，如何创建和操作小任务