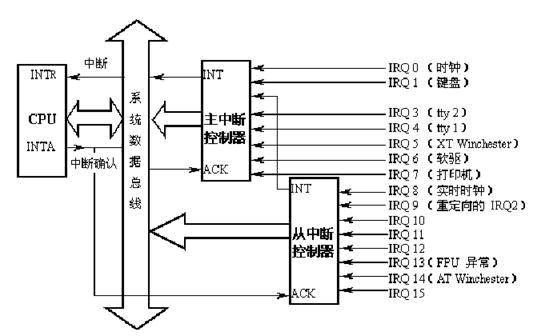
**关于中断和异常**

　　一般在书中都会把中断和异常一起说明，因为它们具有相同的特点，同时也有不同的地方。在CPU里，中断和异常都会放入到一个中断描述符表中，都需要特定的处理程序进行处理，并且它们都是异步事件，内核完全不知道何时会有一个异常或者中断发生。当异常或者中断发生时，进程都会陷入内核，在内核中执行相应的处理。异常一般都是由CPU内部或者进程产生，而中断一般都是由外部设备产生。异常处理过程实际上和系统调用没什么区别(实际上系统调用是通过一个0x80异常陷入内核当中)，而中断的处理过程和情况就相对来说比较复杂。一个中断处理分为硬中断和软中断两个部分，在中断处理的过程中系统是禁止调度和抢占的，而异常处理过程中是允许的。一个中断处理程序可以抢占其他的中断处理程序，也可以抢占异常处理程序，相反的，异常处理程序却不能够抢占中断处理程序。

**可编程中断控制器(PIC、APIC)**

　　为了方便说明，这里我们将PIC和APIC统称为中断控制器。中断控制器是作为中断(IRQ)和CPU核之间的一个桥梁而存在的，每个CPU内部都有一个自己的中断控制器，中断线并不是直接与CPU核相连，而是与CPU内部或外部的中断控制器相连。而为什么叫做可编程中断控制器，是因为其本身有一定的寄存器，CPU可以通过操作设置中断控制器屏蔽某个中断引脚的信号，实现硬件上的中断屏蔽。中断控制器也可以级联提供更多的中断线，具体如下：



　　如上图，CPU的INTR与中断控制器的INT相连，INTA与ACK相连，当一个外部中断发生时(比如键盘中断IRQ1)，中断控制器与CPU交互操作如下：

1. IRQ1发生中断，主中断控制器接收到中断信号，检查中断屏蔽寄存器IRQ1是否被屏蔽，如果屏蔽则忽略此中断信号。
2. 将中断控制器中的中断请求寄存器对应的IRQ1位置位，表示收到IRQ1中断。
3. 中断控制器拉高INT引脚电平，告知CPU有中断发生。
4. CPU每执行完一条指令时，都会检查INTR引脚是否被拉高，这里已被拉高。
5. CPU检查EFLAGS寄存器的中断运行标志位IF是否为1，若为1，表明允许中断，通过INTA向中断控制器发出应答。
6. 中断控制器接收到应答信号，将IRQ1的中断向量号发到数据总线上，此时CPU会通过数据总线读取IRQ1的中断向量号。
7. 最后，如果中断控制器需要EOI(End of Interrupt)信号，CPU则会发送，否则中断控制器自动将INT拉低，并清除IRQ1对应的中断请求寄存器位。

　　在linux内核中，用struct irq\_chip结构体描述一个可编程中断控制器，它的整个结构和调度器中的调度类类似，里面定义了中断控制器的一些操作，如下：

[复制代码](javascript:void(0);)

struct irq\_chip {

/\* 中断控制器的名字 \*/

const char \*name;

/\* 控制器初始化函数 \*/

unsigned int (\*irq\_startup)(struct irq\_data \*data);

/\* 控制器关闭函数 \*/

void (\*irq\_shutdown)(struct irq\_data \*data);

/\* 使能irq操作，通常是直接调用irq\_unmask()，通过data参数指明irq \*/

void (\*irq\_enable)(struct irq\_data \*data);

/\* 禁止irq操作，通常是直接调用irq\_mask，严格意义上，他俩其实代表不同的意义，disable表示中断控制器根本就不响应该irq，而mask时，中断控制器可能响应该irq，只是不通知CPU \*/

void (\*irq\_disable)(struct irq\_data \*data);

/\* 用于CPU对该irq的回应，通常表示cpu希望要清除该irq的pending状态，准备接受下一个irq请求 \*/

void (\*irq\_ack)(struct irq\_data \*data);

/\* 屏蔽irq操作，通过data参数表明指定irq \*/

void (\*irq\_mask)(struct irq\_data \*data);

/\* 相当于irq\_mask() + irq\_ack() \*/

void (\*irq\_mask\_ack)(struct irq\_data \*data);

/\* 取消屏蔽指定irq操作 \*/

void (\*irq\_unmask)(struct irq\_data \*data);

/\* 某些中断控制器需要在cpu处理完该irq后发出eoi信号 \*/

void (\*irq\_eoi)(struct irq\_data \*data);

/\* 用于设置该irq和cpu之间的亲和力，就是通知中断控制器，该irq发生时，那些cpu有权响应该irq \*/

int (\*irq\_set\_affinity)(struct irq\_data \*data, const struct cpumask \*dest, bool force);

int (\*irq\_retrigger)(struct irq\_data \*data);

/\* 设置irq的电气触发条件，例如 IRQ\_TYPE\_LEVEL\_HIGH(电平触发) 或 IRQ\_TYPE\_EDGE\_RISING(边缘触发) \*/

int (\*irq\_set\_type)(struct irq\_data \*data, unsigned int flow\_type);

/\* 通知电源管理子系统，该irq是否可以用作系统的唤醒源 \*/

int (\*irq\_set\_wake)(struct irq\_data \*data, unsigned int on);

void (\*irq\_bus\_lock)(struct irq\_data \*data);

void (\*irq\_bus\_sync\_unlock)(struct irq\_data \*data);

void (\*irq\_cpu\_online)(struct irq\_data \*data);

void (\*irq\_cpu\_offline)(struct irq\_data \*data);

void (\*irq\_suspend)(struct irq\_data \*data);

void (\*irq\_resume)(struct irq\_data \*data);

void (\*irq\_pm\_shutdown)(struct irq\_data \*data);

void (\*irq\_calc\_mask)(struct irq\_data \*data);

void (\*irq\_print\_chip)(struct irq\_data \*data, struct seq\_file \*p);

int (\*irq\_request\_resources)(struct irq\_data \*data);

void (\*irq\_release\_resources)(struct irq\_data \*data);

unsigned long flags;

};

[复制代码](javascript:void(0);)

**中断描述符表(IDT)**

　　在中断系统中有两个名字很相像的结构，就是**中断描述符表**和**中断描述符数组**。这里我们先说说中断描述符表。  ​一个系统中的中断和异常加起来一共是256个，它们以向量的形式保存在中断描述符表中，每一个向量是8字节(整个表大小就是8 x 256=2048字节)，**其主要保存着权限位和向量对应的中断或异常处理程序的入口地址**。而一般的，linux会将中断描述符表中的0~31用于非屏蔽中断和异常，其他的中断用于32~255之间。CPU把中断描述符表的向量类型分为三种类型：**任务门、中断门、陷阱门**。CPU为了防止恶意程序访问中断，限制了中断门的权限，而在某些时候，用户程序又必须使用中断，所以Linux把中断描述符的中断向量类型改为了5种：**中断门，系统门，系统中断门，陷阱门，任务门**。这个中断描述符表的基地址保存在idtr寄存器中。

**中断门**

        ​用户程序不能访问的CPU中断门(权限字段为0)，所有的中断处理程序都是这个，被限定在内核态执行。**会清除IF标志，屏蔽可屏蔽中断**。

**系统门**

        用户程序可以访问的CPU陷阱门(权限字段为3)。我们的系统调用就是通过向量128系统门进入的。

**系统中断门**

        能够被用户进程访问的CPU陷阱门(权限字段为3)，作为一个特别的异常处理所用。

**陷阱门**

        用户进程不能访问的CPU陷阱门(权限字段为0)，大部分异常处理程序入口都为陷阱门。

**任务门**

        用户进程不能访问的CPU任务门(权限字段为0)，''Double fault"异常处理程序入口。

　　当我们发生异常或中断时，系统首先会判断权限字段(安全处理)，权限通过则进入指定的处理函数，而所有的中断门的中断处理函数都是同一个，它首先是一段汇编代码，汇编代码操作如下：

* 执行SAVE\_ALL宏，保存**中断向量号**和**寄存器上下文**至当前运行进程的内核栈或者硬中断请求栈(当内核栈大小为8K时保存在内核栈，若为4K，则保存在硬中断请求栈)。
* 调用**do\_IRQ()**函数。
* 跳转到ret\_from\_intr，这是一段汇编代码，主要用于判断是否需要进行调度。

**中断处理**

　　每个能够产生中断的设备或者模块都会在内核中注册一个**中断服务例程(ISR)**，当产生中断时，中断处理程序会被执行，在中断处理程序中，首先会保存中断向量号和上下文，之后执行中断线对应的中断服务例程。对于CPU来说，中断线是非常宝贵的资源，而由于计算机的发展，外部设备数量和种类越来越多，导致了中断线资源不足的情况，linux为了应对这种情况，实现了两种中断线分配方式，分别是：**共享中断线，中断线动态分配**。

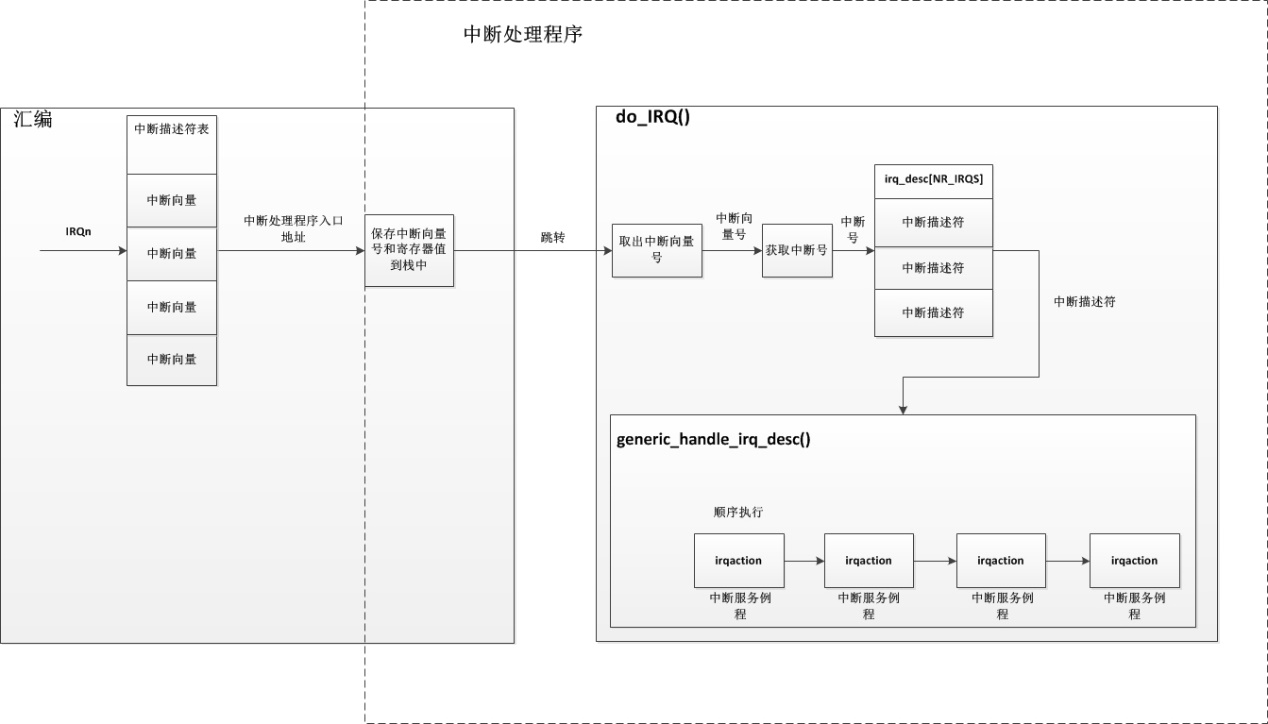
**共享中断线**

　　多个设备共用一条中断线，当此条中断线发生中断时，因为不可能预先知道哪个特定的设备产生了中断，因此，这条中断线上的每个中断服务例程都会被执行，以验证是哪个设备产生的中断(一般的，设备产生中断时，会标记自己的状态寄存器，中断服务例程通过检查每个设备的状态寄存器来查找产生中断的设备)。

**中断线动态分配**

　　一条中断线在可能使用的时刻才与一个设备驱动程序关联起来，这样一来，即使几个硬件设备并不共享中断线，同一个中断向量也可以由这几个设备在不同时刻运行。

　　共享中断线的分配方式是比较常见的，一次典型的基于共享中断线的中断处理流程如下：

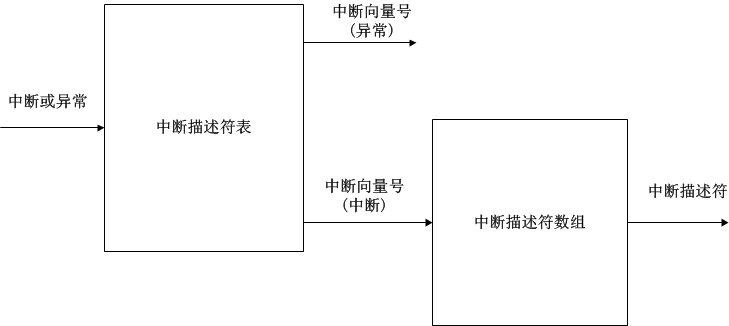


　　由于中断处于中断上下文中，所以在中断处理过程中，会有以下几个特性：

* 中断处理程序正在运行时，CPU会通知中断控制器屏蔽产生此中断的中断线。此中断线发出的信号被暂时忽略，当中断处理程序结束时恢复此中断线。
* 在中断服务例程的设计中，原则上是立即处理紧急的操作，将非紧急的操作延后处理(交给软中断进行处理)。
* 中断处理程序是运行在中断上下文，但是其是代表进程运行的，因此它所代表的进行必须处于TASK\_RUNNING状态，否则可能出现僵死情况，因此在中断处理程序中不能执行任何阻塞过程。

**中断描述符**

　　中断描述符用于描述IRQ线的属性与状态，每个IRQ都有它自己的中断描述符，这些中断描述符用一个数组保存， 这个数组就是**中断描述符数组**，整个中断描述符数组长度为NR\_IRQS（通常为224）项。当产生一个中断或者异常时，首先会从中断描述符表中获取到一个中断向量号时(此中断向量号有可能表示中断，也可能表示的是一个异常)，如果是一个中断导致的，会执行do\_IRQ()函数，而在do\_IRQ()函数中，会根据中断向量号，从中断描述符数组中获取对应的中断描述符，如下图：



　　整个中断描述符结构如下：

[复制代码](javascript:void(0);)

struct irq\_desc {

struct irq\_data irq\_data;

/\* irq的统计信息，在proc中可查到 \*/

unsigned int \_\_percpu \*kstat\_irqs;

/\* 回调函数，当此中断产生中断时，会调用handle\_irq，在handle\_irq中进行遍历irqaction链表

\* handle\_simple\_irq 用于简单处理；

\* handle\_level\_irq 用于电平触发中断的流控处理；

\* handle\_edge\_irq 用于边沿触发中断的流控处理；

\* handle\_fasteoi\_irq 用于需要响应eoi的中断控制器；

\* handle\_percpu\_irq 用于只在单一cpu响应的中断；

\* handle\_nested\_irq 用于处理使用线程的嵌套中断；

\*/

irq\_flow\_handler\_t handle\_irq;

#ifdef CONFIG\_IRQ\_PREFLOW\_FASTEOI

irq\_preflow\_handler\_t preflow\_handler;

#endif

/\* 中断服务例程链表 \*/

struct irqaction \*action; /\* IRQ action list \*/

/\* 状态 \*/

unsigned int status\_use\_accessors;

/\* 函数调用中使用，另一个名称为istate \*/

unsigned int core\_internal\_state\_\_do\_not\_mess\_with\_it;

/\* 嵌套深度，中断线被激活显示0，如果为正数，表示被禁止次数 \*/

unsigned int depth; /\* nested irq disables \*/

unsigned int wake\_depth; /\* nested wake enables \*/

/\* 此中断线上发生的中断次数 \*/

unsigned int irq\_count; /\* For detecting broken IRQs \*/

/\* 上次发生未处理中断时的jiffies值 \*/

unsigned long last\_unhandled; /\* Aging timer for unhandled count \*/

/\* 中断线上无法处理的中断次数，如果当第100000次中断发生时，有超过99900次是意外中断，系统会禁止这条中断线 \*/

unsigned int irqs\_unhandled;

atomic\_t threads\_handled;

int threads\_handled\_last;

/\* 锁 \*/

raw\_spinlock\_t lock;

struct cpumask \*percpu\_enabled;

#ifdef CONFIG\_SMP

/\* CPU亲和力关系，其实就是每个CPU是占一个bit长度，某CPU上置为1表明该CPU可以进行这个中断的处理 \*/

const struct cpumask \*affinity\_hint;

struct irq\_affinity\_notify \*affinity\_notify;

#ifdef CONFIG\_GENERIC\_PENDING\_IRQ

/\* 用于调整irq在各个cpu之间的平衡 \*/

cpumask\_var\_t pending\_mask;

#endif

#endif

unsigned long threads\_oneshot;

atomic\_t threads\_active;

/\* 用于synchronize\_irq()，等待该irq所有线程完成 \*/

wait\_queue\_head\_t wait\_for\_threads;

#ifdef CONFIG\_PM\_SLEEP

/\* irqaction数量 \*/

unsigned int nr\_actions;

unsigned int no\_suspend\_depth;

unsigned int force\_resume\_depth;

#endif

#ifdef CONFIG\_PROC\_FS

/\* 指向与IRQn相关的/proc/irq/n目录的描述符 \*/

struct proc\_dir\_entry \*dir;

#endif

int parent\_irq;

struct module \*owner;

/\* 在/proc/interrupts所显示名称 \*/

const char \*name;

} \_\_\_\_cacheline\_internodealigned\_in\_smp;

[复制代码](javascript:void(0);)

　　core\_internal\_state\_\_do\_not\_mes\_with\_it成员是用于记录此中断线状态的，中断线状态有如下几种形式：

[复制代码](javascript:void(0);)

IRQS\_AUTODETECT 　　　　/\* 该IRQ线用来进行硬件设备探测 \*/

IRQS\_SPURIOUS\_DISABLED /\* 该IRQ线被禁止，是由于产生了欺骗性中断 \*/

IRQS\_POLL\_INPROGRESS /\* 该IRQ进行轮询检查是否发生中断 \*/

IRQS\_ONESHOT　　　　　　　　/\* 此IRQ没有在主处理函数中进行unmasked处理 \*/

IRQS\_REPLAY　　　　　　　　 /\* IRQ线已被禁止，但前一个出现的中断还没有被应答 \*/

IRQS\_WAITING　　　　　　　　/\* 进行硬件设备探测时，会将所有没有挂载中断服务程序的IRQ线状态设置为IRQS\_WAITING，如果该IRQ上有中断产生，就清除这个状态，可以推断哪些引脚产生过中断 \*/

IRQS\_PENDING　　　　　　　　/\* IRQ已经被应答(挂起)，但是内核还没有进行处理 \*/

IRQS\_SUSPENDED　　　　　　 /\* 此IRQ被延迟 \*/

[复制代码](javascript:void(0);)

**中断服务例程(ISR)**

　　中断服务例程用于描述一个设备的中断处理(区别与中断处理函数)，每个申请了中断的外部设备都会有一个中断服务例程，其作用就是执行对应设备的中断处理。当多个设备共享IRQ线时，内核会将此IRQ线上所有设备的中断服务例程组织成一个链表并保存在中断描述符中，当此IRQ线产生中断时，中断处理函数会依次执行此IRQ线上的中断服务例程。内核使用struct irqaction描述一个中断服务例程：

[复制代码](javascript:void(0);)

struct irqaction {

/\* 此中断服务例程的中断处理函数 \*/

irq\_handler\_t handler;

/\* 设备ID，一般用于指向中断处理时需要的数据结构传入handler \*/

void \*dev\_id;

/\* 此中断服务例程在CPU上所对应的设备ID \*/

void \_\_percpu \*percpu\_dev\_id;

/\* 链表中下一个中断服务例程 \*/

struct irqaction \*next;

/\* 进行中断处理的内核线程执行函数 \*/

irq\_handler\_t thread\_fn;

/\* 一个内核线程，用于执行中断处理 \*/

struct task\_struct \*thread;

/\* IRQ线，IRQ号 \*/

unsigned int irq;

unsigned int flags;

unsigned long thread\_flags;

unsigned long thread\_mask;

const char \*name;

/\* 指向/proc/irq/n目录的描述符 \*/

struct proc\_dir\_entry \*dir;

} \_\_\_\_cacheline\_internodealigned\_in\_smp;

[复制代码](javascript:void(0);)

**irq\_stat数组**

　　此数组包含NR\_CPUS个元素，系统中每个CPU对应数组中的一个元素。每个元素的类型为irq\_cpustat\_t，其包含几个计数器和内核记录CPU正在做什么的标志。

[复制代码](javascript:void(0);)

typedef struct {

unsigned int \_\_softirq\_pending; /\* 表示挂起的软中断，每一位表示一个软中断，为1表示挂起 \*/

long idle\_timestamp; /\* CPU变为空闲的时间点 \*/

/\* 硬中断统计. \*/

unsigned int irq\_timer\_count; /\* 定时器中断统计 \*/

unsigned int irq\_syscall\_count; /\* 系统调用中断统计 \*/

unsigned int irq\_resched\_count;

unsigned int irq\_hv\_flush\_count;

unsigned int irq\_call\_count;

unsigned int irq\_hv\_msg\_count;

unsigned int irq\_dev\_intr\_count;

} \_\_\_\_cacheline\_aligned irq\_cpustat\_t;

[复制代码](javascript:void(0);)

**数据结构总结**

　　到此，在中断处理中所涉及的几个重要的数据结构已经说明，其最主要的数据结构为：**中断描述符(struct irq\_desc)，中断控制器描述符(struct irq\_chip)，中断服务例程(struct irqaction)**。它们的组织形式如下：

