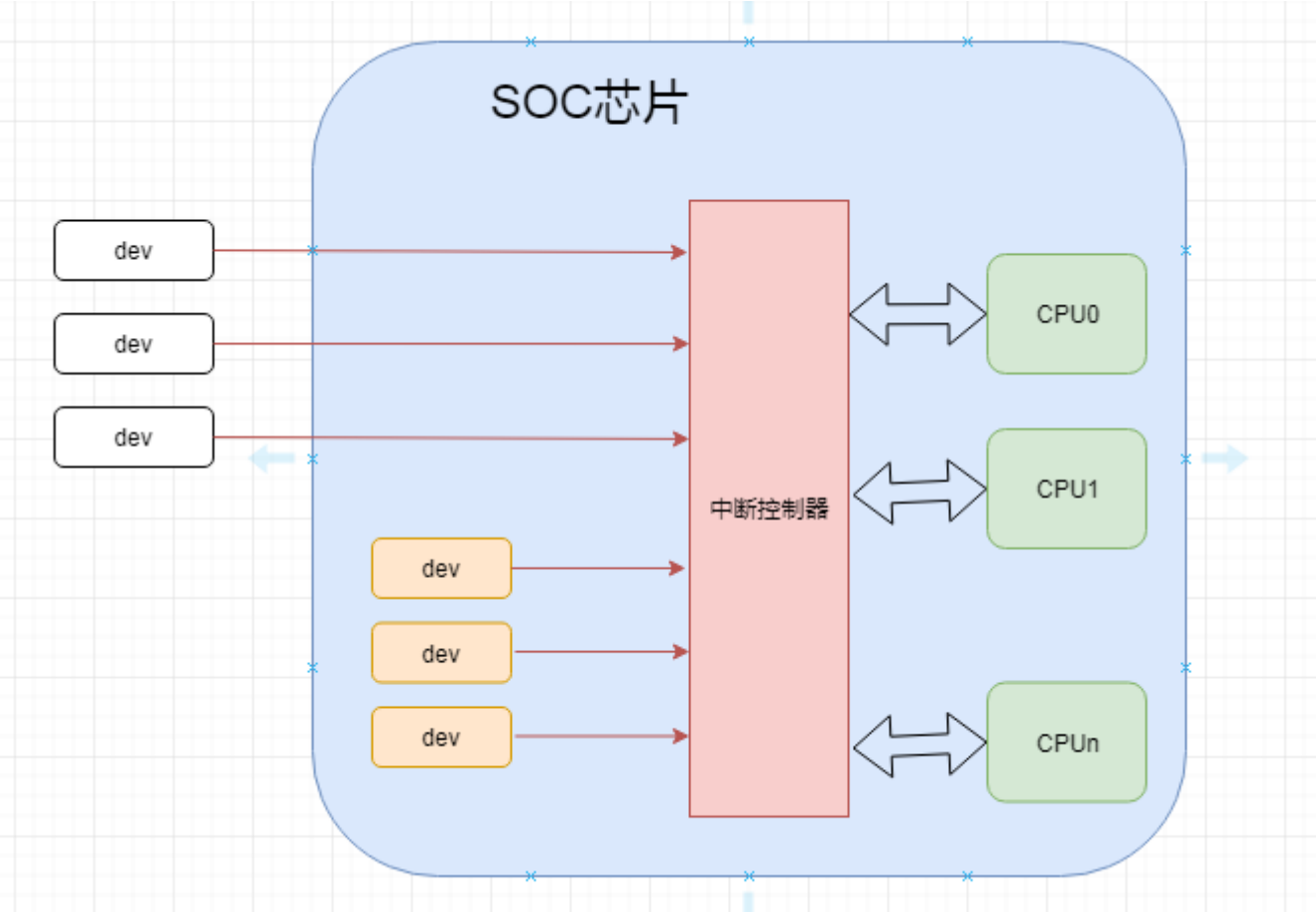
中断系统基本原理

> Author:MLK
>
> Date:MLK

主要介绍ARM架构的中断子系统，其他架构的原理实现差不多，只是内核中对硬件的抽象层会有稍微的区别

1. 设备、中断控制器和CPU

在一个SOC的芯片上，与中断相关的硬件划分为3类，他们分别是：设备、中断控制器和CPU本身，下图1.1表示了smp系统中的中断硬件结构组成:



- 设备：发起中断的源，当设备需要请求某种服务的时候，它会发起一个硬件中断信号，通常该中断信号会直接连接到中断控制器（GIC），直接由中断控制器处理。目前很多的SOC芯片，发起的中断可以位于SOC芯片的外部也可以位于SOC的内部，因为目前大多数SOC集成了大量的硬件IP，I2C、SPI等控制器。
- 中断控制器：负责收集所有中断源发起的中断，目前arm中的中断控制器都是可编程的，通过对中断控制器的编程，我们可以控制每个中断源的优先级、中断的出发事件类型等，也可以打开和关闭某一个中断源，在smp系统中，甚至可以控制某个源发往哪个CPU进行处理。对于ARM架构的SOC，使用较多的是GIC（general interrupt Controller）。

- CPU ： 是最终处理中断的器件，它通过对可编程中断控制器的编程操作，控制和管理系统中的每个中断，当中断控制器最终判定一个中断可以被处理时，他会根据事先的设定，通知其中一个或者几个CPU对该中断的处理，虽然中断控制器可以同时通知数个CPU对某个中断进行处理，实际上只会有一个CPU响应这个中断请求，但具体哪个CPU进行响应可能是随机的（ ？ ），中断控制器在硬件上对这一特性进行了保证，不过也依赖与操作系统对中断系统的软件实现。在SMP系统中，CPU之间也通过IPI（inter processor interrupt）中间进行通信。

2 . IRQ编号

这个IRQ编号实在Linux系统中体现的一个中断号，并不是真正的硬件中断ID号，当向系统注册一个中断源时，系统都会分配一个唯一的编号用于识别该中断，我们称它为IRQ_NUM。IRQ_NUM贯穿在整个Linux的通用中断子系统中。在SOC中，每个中断源的IRQ_NUM编号都会在arch相关的头文件中，比如arch/arm/mach-xxx/include/irqs.h。驱动程序在请求中断服务时，它会使用相应的IRQ_NUM注册该中断，当中断发生时，CPU会从中断控制器中获取到相关的信息，然后计算出相应的IRQ_NUM,然后把该IRQ传递到相应的驱动程序中。

3 . 驱动程序申请中断

在linux的中断子系统向驱动程序提供了一系列的API接口，其中的两个可用于向系统申请中断的API如下：

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
             irq_handler_t thread_fn, unsigned long irqflags,
             const char *devname, void *dev_id)

int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
        const char *name, void *dev)
```

其实最终request_irq也会调用request_thread_irq函数，只是irq_handler_t参数默认传入为NULL。

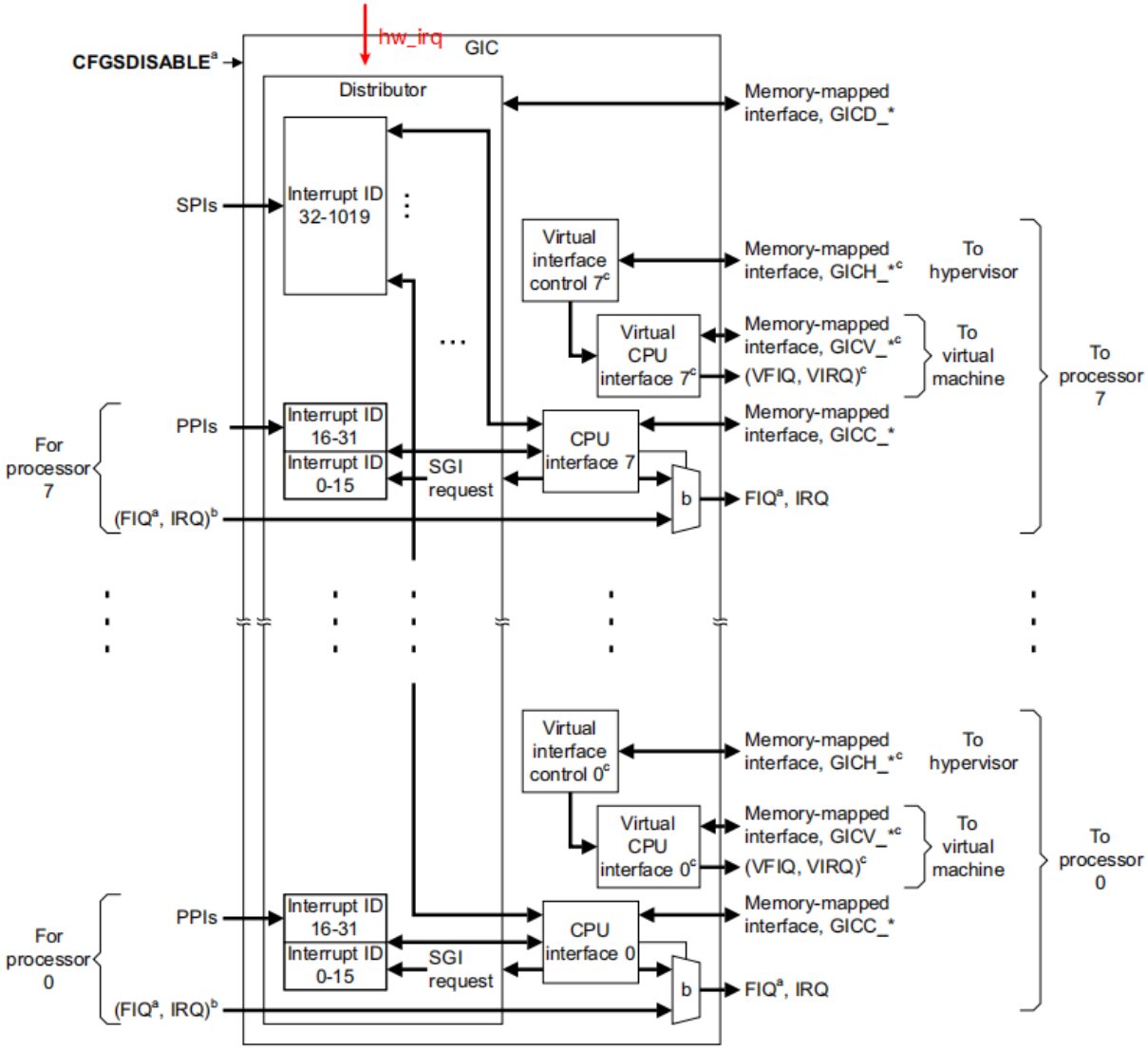
参数含义：

- irq要申请的IRQ_NUM
- handler是中断处理服务函数，在函数工作在中断上下文，如果不需要可以设置为NULL，但是不能和thread_fn同时为NULL
- thread_fn是中断线程的回调函数，该函数工作在进程上下文中，如果不需要，也可以传入NULL，但是不能和handler同时为NULL
- irqflags是该中断的标志位，指定该中断触发的类型，比如电平触发还是边沿触发或者是共享中断等。
- devname用来设定该中断的名字
- dev_id用于共享中断时的cookie data，主要用于区分共享中断具体由哪个设备发起。

4 . 通用中断子系统（Generic irq）的软件抽象

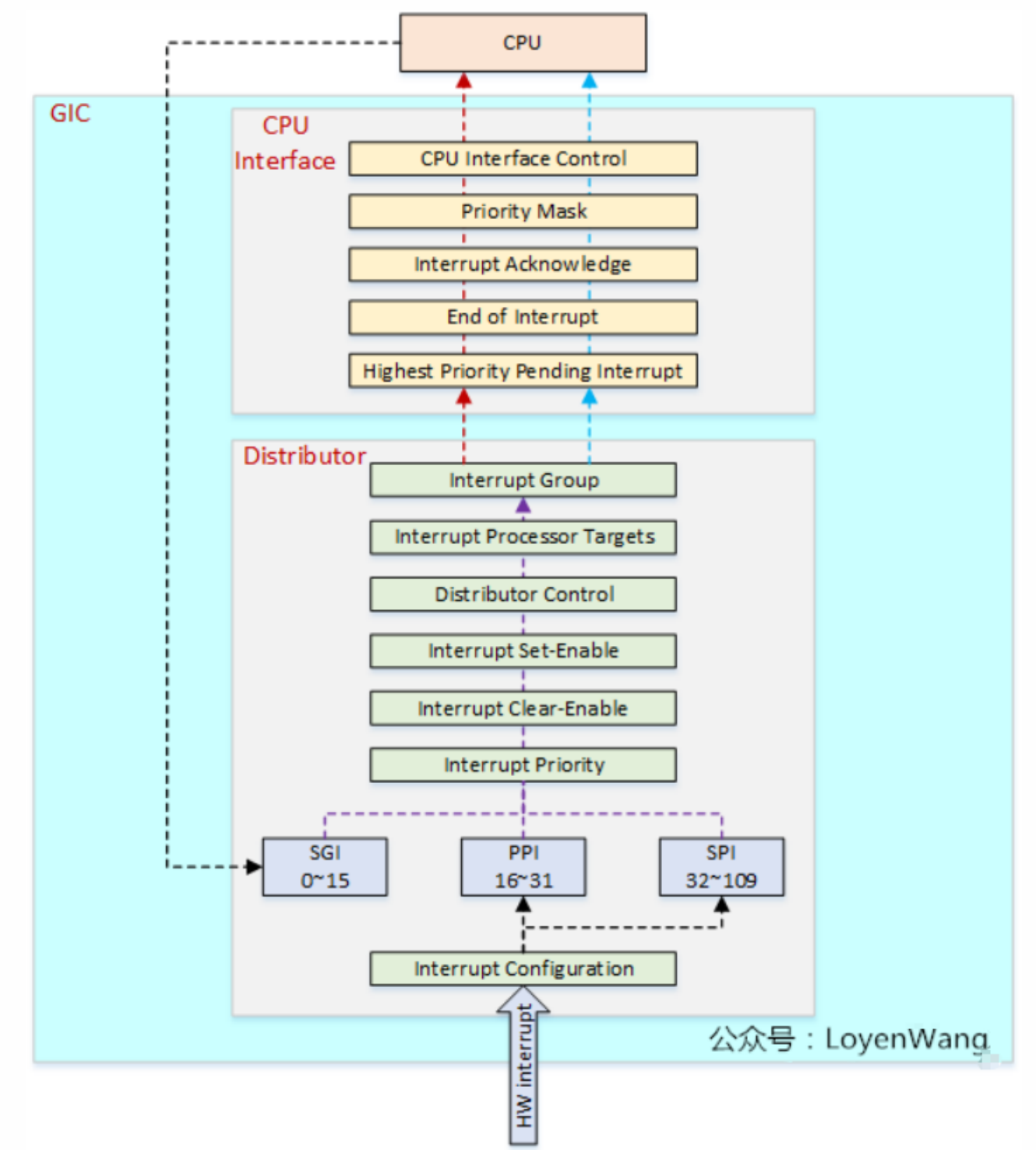
4.1 GIC硬件原理

本文仅针对ARM-GIC2.0控制器分析，其他架构体系或者控制器版本不在本文探讨范围内，不过有一定的参考性。在讲通用中断子系统之前，先介绍一下GIC中断控制器的原理，ARM公司提供了一个通用的中断控制器GIC（Generic Interrupt Controller），GIC的版本包括V1~V4,ARMV7使用的是中断控制器的V2版本，TI-AM5827和AM335X都使用此版本，原理框架如下：



- GIC-V2从功能上来说，除了常用的中断使能，中断屏蔽、优先级管理等功能外，还支持安全扩展、虚拟化等。
- 由上图看，GIC-V2主要由Distributor和CPU Interface两个模块组成，Distributor主要负责中断源的管理，暴包括优先级的处理、屏蔽、抢占等，并将最高优先级的中断分发给CPU Interface,CPU Interface主要用于连接处理器，与处理器进行交互，主要把中断分发给哪个CPU进行处理。
- 而Virtual Interface都是与虚拟化有关，本文不涉及此分析。

在此引用网上的一章细节图来说明Distributor和CPU Interface的功能：



- GIC-V2支持三种类型的中断；
  - `SGI （software-generate interrupts）`：软件中断,主要用于核间通信，内核中的IPI ： inter-processor interrupts就是基于SGI，中断号为ID0~ID15;
  - `PPI(Private Peripheral Interrupt)` ：私有外设中断，没个CPU都有自己的私有中断，典型的应用有Local time，中断号ID16~ID31；
  - `SPI（Shaerd Perpheral Interrupt）`:共享外设中断，中断产生后，可以分发到某个CPU上，中断号为ID32-ID1019，ID1020-ID1023保留用于特殊用途；
- `Distributor`功能：
  - 全局开关控制`Distributor`分发到`CPU Interface`；

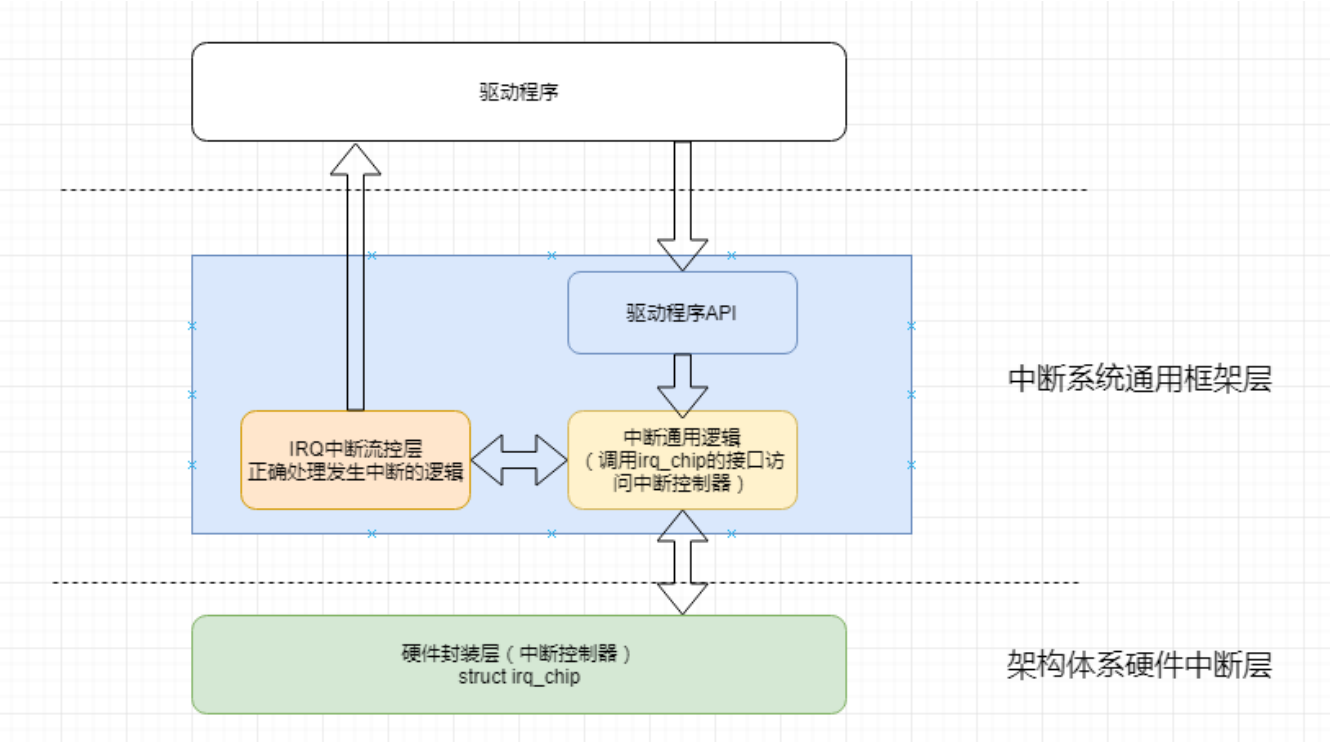
    - 打开或关闭每个中断；
    - 设置每个中断的优先级；
    - 设置每个中断将路由的CPU列表；
    - 设置每个外设中断的触发方式：电平触发、边缘触发；
    - 设置每个中断的Group：Group0或Group1，其中Group0用于安全中断，支持FIQ和IRQ，Group1用于非安全中断，只支持IRQ；
    - 将`SGI`中断分发到目标CPU上；
    - 每个中断的状态可见；
    - 提供软件机制来设置和清除外设中断的pending状态；
- `CPU Interface`功能：
    - 使能中断请求信号到CPU上；
    - 中断的确认；
    - 标识中断处理的完成；
    - 为处理器设置中断优先级掩码；
    - 设置处理器的中断抢占策略；
    - 确定处理器的最高优先级pending中断；

4.2 通用中断子系统软件框架

在通用中断子系统（generic irq）出现之前，内核使用__do_IRQ处理所用的中断，这也就是说__do_IRQ需要处理各种类型的中断，这就导致了软件实现的复杂性。这对这种情况，内核的开发者在2.6以后的某个版本就彻底丢弃掉了此种实现方式。然后设计了通用中断子系统，把不同的中断类型区分开来，比如：

- SGI
- PPI
- SPI

但是ARM的GIC2.0的中断控制器在CPU处理完中断后，需要给控制器发送结束控制器信号，被称为需要回应的中断控制器eoi（end of interrupt），加入了fast eoi type类型，然后又针对smp架构加入了per cpu type。把这些不同的中断类型抽象出来，称为了中断子系统的流控层。要使所有的CPU架构体系都可以使用中断流控层的代码，需要把中断控制器进一步进行封装，形成中断子系统的硬件封装层，下图表示了通用中断子系统的一个层次结构：



- 硬件封装层 ： 主要包含了体系架构相关的所有代码，包括中断控制器的抽象封装，中断控制器的初始化，以及各个IRQ的相关数据的初始化，CPU的中断入口函数初始化。然后中断通用逻辑层通过标准的封装接口（struct irq_chip结构中定义的接口）访问并控制中断控制器的行为，当外设中断发生时，CPU的中断入口函数通过读取寄存器获取到实际的硬件中断号（hwirq），然后再通过之前已经映射对应好的hwirq到virq（irq_num对Linux系统而言），获取到irq_num，通过中断通用逻辑层提供的标准函数，把中断调用传递到中断流控层。对中断控制器的抽象用struct irq_chip结构体来表示：

```
struct irq_chip {
    const char  *name;
    unsigned int    (*irq_startup)(struct irq_data *data);
    void        (*irq_shutdown)(struct irq_data *data);
    void        (*irq_enable)(struct irq_data *data);
    void        (*irq_disable)(struct irq_data *data);

    void        (*irq_ack)(struct irq_data *data);
    void        (*irq_mask)(struct irq_data *data);
    void        (*irq_mask_ack)(struct irq_data *data);
    void        (*irq_unmask)(struct irq_data *data);
    void        (*irq_eoi)(struct irq_data *data);

    int     (*irq_set_affinity)(struct irq_data *data, const struct cpumask *dest,
bool force);
    int     (*irq_retrigger)(struct irq_data *data);
    int     (*irq_set_type)(struct irq_data *data, unsigned int flow_type);
    int     (*irq_set_wake)(struct irq_data *data, unsigned int on);
        ......
};
```

- 中断流控层 ： 可以合理并正确的处理连续发生的中断，比如一种中断在处理中，同一个中断再次到达时该如何去处理，何时屏蔽该中断，何时打开该中断，CPU何时回应中断控制器的一系列操作。该层实现了与体系和硬件无关的中断流控处理操作，针对不同的中断传入不同的中断流控函数，最终会把中断控制权传递到驱动程序注册中断时传入的中断处理函数后者中断线程函数。

- 中断通用逻辑层 ： 该层主要实现了对中断系统几个重要数据的管理，并提供一系列的辅助管理函数。同时该层还实现中断线程的实现和管理，共享中断和嵌套中断的实现（目前Linux系统不支持中断嵌套），另外该层提供了一些借口函数，作为硬件封装层和流程层以及驱动程序API之间的桥梁，其中的某些API如下：
    - generic_handle_irq()
    - irq_to_desc()
    - irq_set_chip
    - irq_set_chained_handler()
- 驱动程序API ： 向驱动程序提供一系列的API，用于向系统申请/释放中断，打开和关闭中断，设置中断类型等。驱动程序的开发者一般只会调用这一层提供的API函数来完成驱动程序的中断开发工作，其他的中断细节实现原理则由前面的三层隐藏起来，驱动开发者不需要关注底层的实现，但是作为一个底层开发人员，要想写出好的中断代码，则需要了解中断底层的实现原理，驱动程序API层提供的一些API如下：
    - enable_irq()
    - disable_irq()
    - disable_irq_bosync()
    - request_thread_irq()
    - request_irq()
    - irq_set_affinity()

中断处理状态流程图：

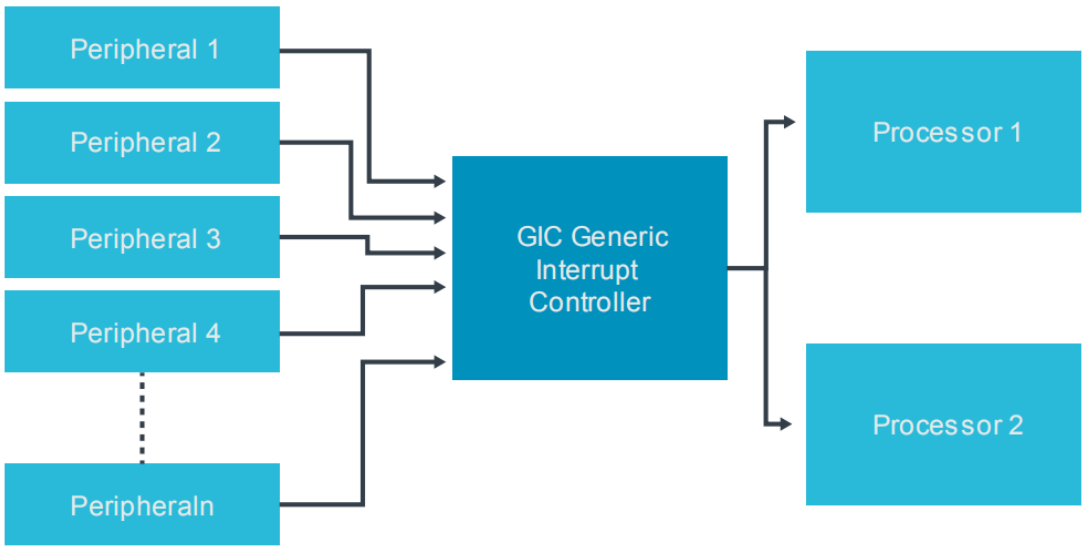


图1 中断源-->GIC-->cpu_interface

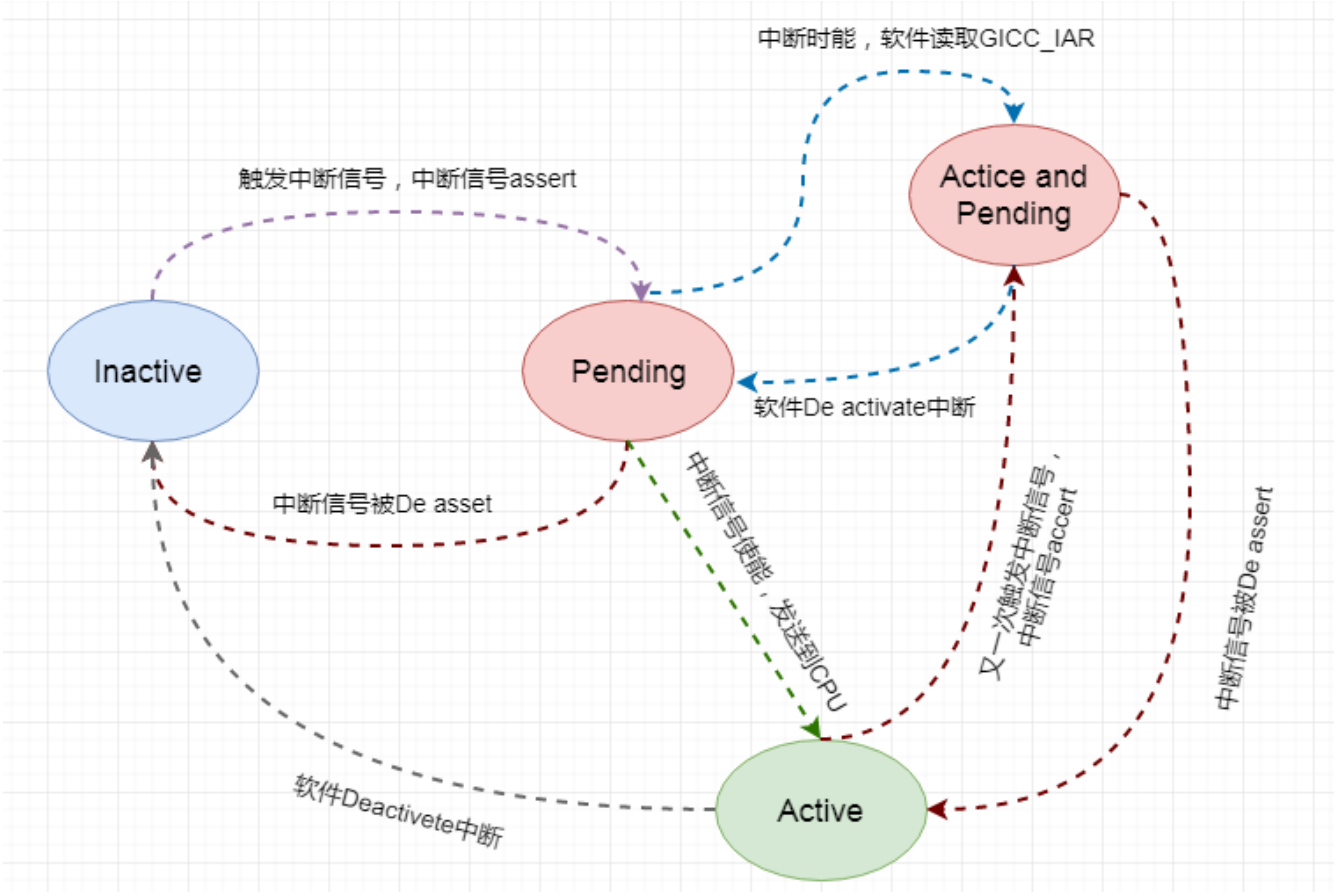


图2 中断处理状态流程图

从图1知道，硬件中断或者软件触发了中断，都会传递给GIC控制器，然后通过GIC控制器告知CPU发生了中断，CPU通过去读取GIC控制器去判断发生了那类中断和具体的中断号。

分析图2的中断处理状态路程图：

- Inactive : 无中断状态发生；
- Pending : 硬件或者软件触发了中断，但尚未到达CPU，在电平触发的模式下，产生中断的同时保持Pending状态；
- Active : 发生了中断并将其传递给目标CPU，并去目标CPU可以处理该中断
- Active andPending : 发生了中断并将其传递给目标CPU，同时发生了相同的中断且该中断正在处理；

GIC检测中断的流程如下：

- GIC捕获到中断信号，中断信号assert，标记为Pending状态；
- Distributpr确定好CPU后，将中断信号发送到目标CPU上，同时，对于每个CPU，Distributor会从Pending信号中选择最高优先级中断发送至CPU Interface；
- CPU Interface : 决定是否将中断信号发送至目标CPU；
- CPU完成中断处理后，发送一个完成信号EOI（end of Interrupt）给GIC；

4.3 CPU的中断入口

我们知道，arm的异常和复位向量表有两种选择，一种是低端向量，向量地址位于0x00000000，另一种是高端向量，向量地址位于0xffff0000，Linux选择使用高端向量模式，就是说，当异常发生时，CPU会把PC指针自动跳转到始于0xffff0000开始的某一个地址上，ARM的异常向量表：

地址	异常种类
FFFF0000	复位
FFFF0004	未定义指令
FFFF0008	软中断（swi）
FFFF000C	Prefetch abort
FFFF0010	Data abort
FFFF0014	保留
FFFF0018	IRQ
FFFF001C	FIQ

中断向量表在`arch/arm/kernel/entry_armv.S`中定义，下面只列出部分关键代码：

```
.globl  __stubs_start
__stubs_start:

    vector_stub irq, IRQ_MODE, 4

    .long   __irq_usr           @  0  (USR_26 / USR_32)
    .long   __irq_invalid           @  1  (FIQ_26 / FIQ_32)
    .long   __irq_invalid           @  2  (IRQ_26 / IRQ_32)
    .long   __irq_svc           @  3  (SVC_26 / SVC_32)

    vector_stub dabt, ABT_MODE, 8

    .long   __dabt_usr          @  0  (USR_26 / USR_32)
    .long   __dabt_invalid          @  1  (FIQ_26 / FIQ_32)
    .long   __dabt_invalid          @  2  (IRQ_26 / IRQ_32)
    .long   __dabt_svc          @  3  (SVC_26 / SVC_32)

vector_fiq:
    disable_fiq
    subs    pc, lr, #4
    ......
    .globl  __stubs_end
__stubs_end:



    .equ    stubs_offset, __vectors_start + 0x200 - __stubs_start
    .globl  __vectors_start
__vectors_start:
 ARM(   swi SYS_ERROR0  )
 THUMB( svc #0      )
 THUMB( nop         )
    W(b)    vector_und + stubs_offset
    W(ldr)  pc, .LCvswi + stubs_offset
```

```
    W(b)    vector_pabt + stubs_offset
    W(b)    vector_dabt + stubs_offset
    W(b)    vector_addrexcptn + stubs_offset
    W(b)    vector_irq + stubs_offset
    W(b)    vector_fiq + stubs_offset

    .globl  __vectors_end
__vectors_end:
```

代码被分成两部分：

- 第一部分 ： 中断向量跳转表，位于`__vectors_start`和`__vectors_end`之间
- 第二部分 ： 处理跳转部分，位于`__stubs_start`和`__stubs_end`之间

```
vector_stub irq, IRQ_MODE, 4
```

以上这一句把宏展开后实际上就是定义了vector_irq，根据进入中断前的cpu模式，分别跳转到**irq_usr或**irq_svc。

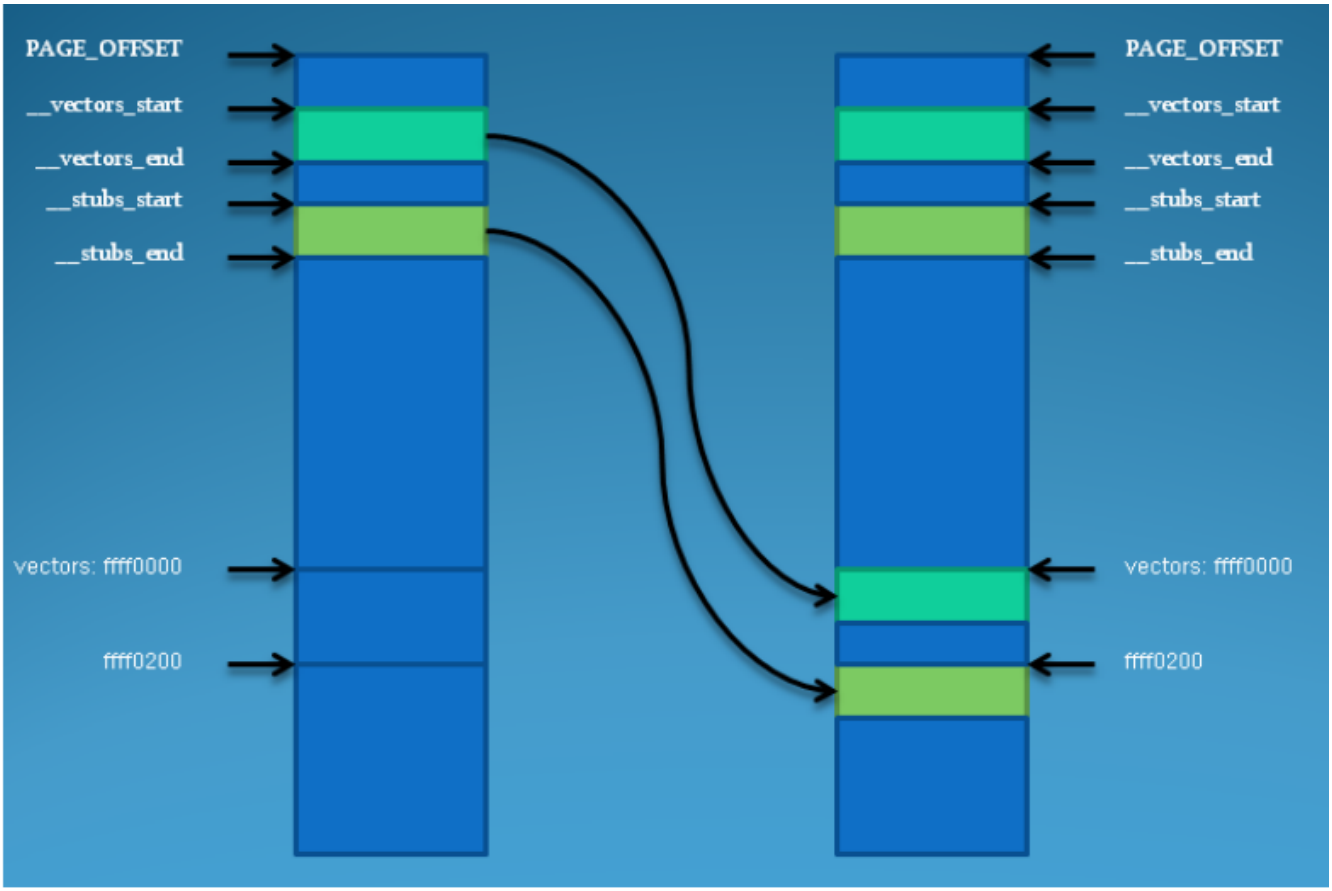
```
vector_stub dabt, ABT_MODE, 8
```

以上这一句把宏展开后实际上就是定义了vector_dabt，根据进入中断前的cpu模式，分别跳转到**dabt_usr或**dabt_svc。

系统启动阶段，位于arch/arm/kernel/traps.c中的early_trap_init()被调用：

```
void __init early_trap_init(void){    ......    /*   * Copy the vectors, stubs and kuser helpers (in entry-armv.S)   * into the vector page, mapped at 0xffff0000, and ensure these * are visible to the instruction stream.    */ memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);  memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);    ......}
```

以上两个memcpy会把**vectors_start开始的代码拷贝到0xffff0000处，把**stubs_start开始的代码拷贝到0xFFFF0000+0x200处，这样，异常中断到来时，CPU就可以正确地跳转到相应中断向量入口并执行他们，下图展示拷贝过程：



对于系统外部设备来说，通常是使用IRQ中断，所以我们只关注`__irq_usr`和`irq_svc`，两者的区别是进入和退出中断时是否进行用户栈和内核栈之间的切换，还有进程调度和抢占的处理等，最终这两个函数都会进入到irq_handler这个函数：

```
    .macro  irq_handler
#ifdef CONFIG_GENERIC_IRQ_MULTI_HANDLER
    ldr r1, =handle_arch_irq
    mov r0, sp
    badr    lr, 9997f
    ldr pc, [r1]
#else
    arch_irq_handler_default
#endif
```

如果选择了`CONFIG_GENERIC_IRQ_MULTI_HANDLER`这个配置项，则使用平台动态设置的irq处理程序，如果不配置则使用默认的irq实现方式，arch_irq_handler_default。这里分析的是AM-5728的sdk包的源码程序，内核源码中平台动态设置了irq的处理程序，即GIC控制器的中断处理程序。具体的分析见下一章节的GIC控制器的初始化。这里分析不动态设置，默认情况下的arch_irq_handler_default， 它位于arch/arm/include/asm/entry_macro_multi.S中:

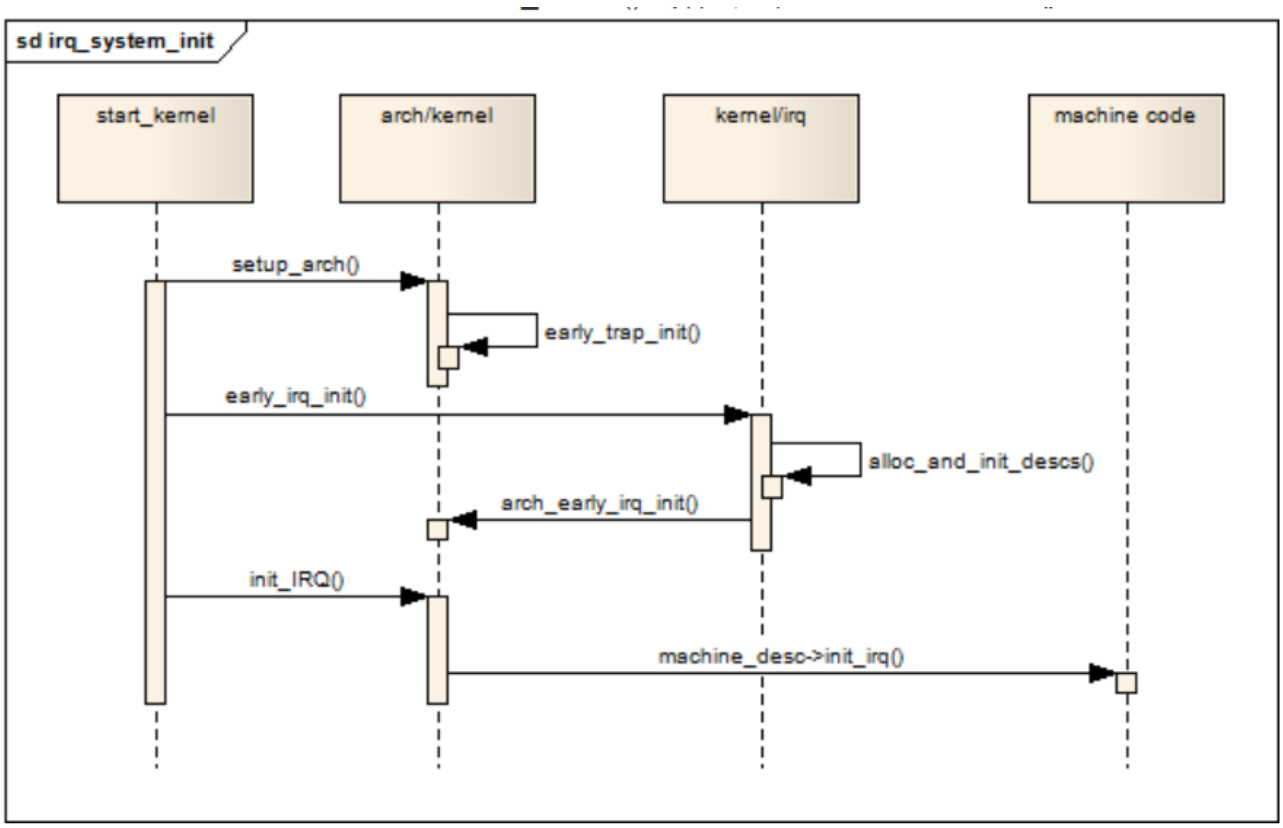
```
    .macro  arch_irq_handler_default
    get_irqnr_preamble r6, lr
1:  get_irqnr_and_base r0, r2, r6, lr
    movne   r1, sp
    @
    @ routine called with r0 = irq number, r1 = struct pt_regs *
    @
    adrne   lr, BSYM(1b)
    bne asm_do_IRQ
```

get_irqnr_preamble和get_irqnr_and_base两个宏由machine级的代码定义，目的就是从中断控制器中获得IRQ编号，紧接着就调用asm_do_IRQ，从这个函数开始，中断程序进入C代码中，传入的参数是IRQ编号和寄存器结构指针，这个函数在arch/arm/kernel/irq.c中实现：

```
/*
 * asm_do_IRQ is the interface to be used from assembly code.
 */
asmlinkage void __exception_irq_entry
asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
{
    handle_IRQ(irq, regs);
}
```

代码执行到这里，中断程序完成了汇编到c代码的传递，并且获得了引起中断的irq_num。

通用中断子系统的相关初始化由start_kerel()函数发起，调用流程如下图所示：



- 在setup_arch()函数中，调用aerly_trap_init()完成了中断向量的拷贝和重定位工作
- early_irq_init()属于与硬件和平台无关的通用逻辑层，它完成irq_desc结构的内存申请，并为他们其中的某些字段填充默认值，然后返回调用arch_early_irq_init函数完成后调用体系相关的完成进一步的初始化工作，不过arm体系没有实现arch_early_irq_init函数。
- start_kernel函数发出init_IRQ调用，它会直接调动所属板子machine_desc结构体的init_irq回调，machine_desc通常在板子的特定代码中，使用MACHINE_START和MACHINE_END宏进行定义。
- machine_desc->init_irq()完成对中断控制器的初始化，为每个irq_desc结构安装合适的流控handler，为每个irq_desc结构安装irq_chip指针，使他指向正确的中断控制器所对应的irq_chip结构的实例，同时，如果该平台中的中断线有多路复用（多个中断公用一个irq中断线）的情况，还应该初始化irq_desc中相应的字段和标志，以便实现中断控制器的级联。

这里以TI-omap1为列子说明，MACHINE_START定义的宏为：

```
MACHINE_START(OMAP_INNOVATOR, "TI-Innovator")
    /* Maintainer: MontaVista Software, Inc. */
    .atag_offset    = 0x100,
    .map_io     = innovator_map_io,
    .init_early     = omap1_init_early,
    .init_irq   = omap1_init_irq,
    .handle_irq = omap1_handle_irq,
    .init_machine   = innovator_init,
    .init_late  = omap1_init_late,
    .init_time  = omap1_timer_init,
    .restart    = omap1_restart,
MACHINE_END
```

从宏中可以看到当发生IRQ中断时，会直接调用omap1_handle_irq函数，下面来分析一下此函数：

```
asmlinkage void __exception_irq_entry omap1_handle_irq(struct pt_regs *regs)
{
    void __iomem *l1 = irq_banks[0].va;
    void __iomem *l2 = irq_banks[1].va;
    u32 irqnr;

    do {
        irqnr = readl_relaxed(l1 + IRQ_ITR_REG_OFFSET);
        irqnr &= ~(readl_relaxed(l1 + IRQ_MIR_REG_OFFSET) & 0xffffffff);
        if (!irqnr)
            break;

        irqnr = readl_relaxed(l1 + IRQ_SIR_FIQ_REG_OFFSET);
        if (irqnr)
            goto irq;
        /*通过读取寄存器来获取到irqnr，后面加上32是因为前面有16个SGI和16个PPI*/
        irqnr = readl_relaxed(l1 + IRQ_SIR_IRQ_REG_OFFSET);
        if (irqnr == omap_l2_irq) {
            irqnr = readl_relaxed(l2 + IRQ_SIR_IRQ_REG_OFFSET);
            if (irqnr)
                irqnr += 32;
        }
irq:
```

```
        if (irqnr)
            handle_domain_irq(domain, irqnr, regs);
        else
            break;
    } while (irqnr);
}
```

获取到硬件IRQ ID后，则执行`handle_domain_irq`，最终会调用`__handle_domain_irq`：

```
int __handle_domain_irq(struct irq_domain *domain, unsigned int hwirq,
            bool lookup, struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);
    unsigned int irq = hwirq;
    int ret = 0;

    irq_enter();

#ifdef CONFIG_IRQ_DOMAIN
    /*之前已经建立好virq和hwirq之间的关系映射，所以现在通过hwirq找到virq*/
    if (lookup)
        irq = irq_find_mapping(domain, hwirq);
#endif

    /*
     * Some hardware gives randomly wrong interrupts.  Rather
     * than crashing, do something sensible.
     */
    if (unlikely(!irq || irq >= nr_irqs)) {
        ack_bad_irq(irq);
        ret = -EINVAL;
    } else {
        /*最后调用此函数去执行struct irq_desc->handle_irq*/
        generic_handle_irq(irq);
    }

    irq_exit();
    set_irq_regs(old_regs);
    return ret;
}
```

通过hwirq找到之前和virq已经建立好的映射关系，然后调用generic_handle_irq函数：

```
static inline void generic_handle_irq_desc(struct irq_desc *desc)
{
    desc->handle_irq(desc);
}
```

然后直接调用struct irq_desc结构体中的handle_irq，即流控层的中断回调函数，GIC-V2中IRQ的流控层的中断函数为，调用关系为：

```
gic_irq_domain_map-->
    irq_set_chip_and_handler_name-->/*此函数用于设置控制器级联的中断函数，不在
* gic_irq_domain_map中被调用*/
    irq_domain_set_info-->/*直接设置GIC的中断入口函数*/
        __irq_set_handler-->
            __irq_do_set_handler-->/*设置irq_desc->handler，即流控层的中断函数*/
                desc->handle_irq = handle
```

我们看下gic_irq_domain_map函数：

```
static int gic_irq_domain_map(struct irq_domain *d, unsigned int irq,
                irq_hw_number_t hw)
{
    struct irq_chip *chip = &gic_chip;

    if (static_key_true(&supports_deactivate)) {
        if (d->host_data == (void *)&gic_data[0])
            chip = &gic_eoimode1_chip;
    }

    if (hw < 32) {
        irq_set_percpu_devid(irq);
        irq_domain_set_info(d, irq, hw, chip, d->host_data,
                    handle_percpu_devid_irq, NULL, NULL);
        irq_set_status_flags(irq, IRQ_NOAUTOEN);
    } else {
        irq_domain_set_info(d, irq, hw, chip, d->host_data,
                    handle_fasteoi_irq, NULL, NULL);
        irq_set_probe(irq);
    }
    return 0;
}
```

可以看到，当`hwirq<32`时，GIC控制器的中断函数为`handle_percpu_devid_irq`,当`hwirq>23`时，即外设IRQ，GIC的中断回调函数为`handle_fasteoi_irq`：

```
void handle_fasteoi_irq(struct irq_desc *desc)
{
    ......
    handle_irq_event(desc);
    ......
}
```

handle_irq_event回去执行handle_irq_event_percpu函数,在此函数中主要是遍历action链表，就是我们驱动函数中注册的中断函数都会加入到这个链表里，这里重点说明一个，这个action链表是同一个irq_num中断号注册的中断函数，即多个外设设备共享一个中断线，如果该irq_num没有共享，则action链表中只有一个驱动程序申请的一个中断函数，并不是说所有的驱动程序函数注册的函数都连接在一个action链表，是一个irq_desc描述符代表一个中断，然后一个irq_desc中就有一个action链表。

```
irqreturn_t handle_irq_event_percpu(struct irq_desc *desc)
```

```
{
    irqreturn_t retval = IRQ_NONE;
    unsigned int flags = 0, irq = desc->irq_data.irq;
    struct irqaction *action = desc->action;

    /* action might have become NULL since we dropped the lock */
    while (action) {
        irqreturn_t res;

        trace_irq_handler_entry(irq, action);
        /*执行驱动程序注册的handler函数*/
        res = action->handler(irq, action->dev_id);
        trace_irq_handler_exit(irq, action, res);

        if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pF enabled interrupts\n",
                  irq, action->handler))
            local_irq_disable();

        switch (res) {
        case IRQ_WAKE_THREAD:
            /*
             * Catch drivers which return WAKE_THREAD but
             * did not set up a thread function
             */
            if (unlikely(!action->thread_fn)) {
                warn_no_thread(irq, action);
                break;
            }

            __irq_wake_thread(desc, action);

            /* Fall through to add to randomness */
        case IRQ_HANDLED:
            flags |= action->flags;
            break;

        default:
            break;
        }

        retval |= res;
        action = action->next;
    }

    add_interrupt_randomness(irq, flags);

    if (!noirqdebug)
        note_interrupt(desc, retval);
    return retval;
}
```

中断控制器的软件抽象 ： struct irq_chip

```
struct irq_chip {
    struct device   *parent_device;     //指向父设备
    const char  *name;      //  /proc/interrupts中显示的名字
    unsigned int    (*irq_startup)(struct irq_data *data);  //启动中断，如果设置成NULL，则默认为
enable
    void        (*irq_shutdown)(struct irq_data *data);     //关闭中断，如果设置成NULL，则默认为
disable
    void        (*irq_enable)(struct irq_data *data);   //中断使能，如果设置成NULL，则默认为
chip->unmask
    void        (*irq_disable)(struct irq_data *data);  //中断禁止

    void        (*irq_ack)(struct irq_data *data);  //开始新的中断
    void        (*irq_mask)(struct irq_data *data); //中断源屏蔽
    void        (*irq_mask_ack)(struct irq_data *data); //应答并屏蔽中断
    void        (*irq_unmask)(struct irq_data *data);   //解除中断屏蔽
    void        (*irq_eoi)(struct irq_data *data);  //中断处理结束后调用

    int     (*irq_set_affinity)(struct irq_data *data, const struct cpumask *dest, bool
force); //在SMP中设置CPU亲和力
    int     (*irq_retrigger)(struct irq_data *data);    //重新发送中断到CPU
    int     (*irq_set_type)(struct irq_data *data, unsigned int flow_type); //设置中断触发类
型
    int     (*irq_set_wake)(struct irq_data *data, unsigned int on);    //使能/禁止电源管理中
的唤醒功能

    void        (*irq_bus_lock)(struct irq_data *data); //慢速芯片总线上的锁
    void        (*irq_bus_sync_unlock)(struct irq_data *data);  //同步释放慢速总线芯片的锁

    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);

    void        (*irq_compose_msi_msg)(struct irq_data *data, struct msi_msg *msg);
    void        (*irq_write_msi_msg)(struct irq_data *data, struct msi_msg *msg);

    int     (*irq_get_irqchip_state)(struct irq_data *data, enum irqchip_irq_state which,
bool *state);
    int     (*irq_set_irqchip_state)(struct irq_data *data, enum irqchip_irq_state which,
bool state);

    int     (*irq_set_vcpu_affinity)(struct irq_data *data, void *vcpu_info);
```

```
    void        (*ipi_send_single)(struct irq_data *data, unsigned int cpu);
    void        (*ipi_send_mask)(struct irq_data *data, const struct cpumask *dest);

    unsigned long   flags;
};

struct irq_domain {
    struct list_head link;  //用于添加到全局链表irq_domain_list中
    const char *name;   //IRQ domain的名字
    const struct irq_domain_ops *ops;   //IRQ domain映射操作函数集
    void *host_data;    //在GIC驱动中，指向了irq_gic_data
    unsigned int flags;
    unsigned int mapcount;  //映射中断的个数

    /* Optional data */
    struct fwnode_handle *fwnode;
    enum irq_domain_bus_token bus_token;
    struct irq_domain_chip_generic *gc;
#ifdef  CONFIG_IRQ_DOMAIN_HIERARCHY
    struct irq_domain *parent;  //支持级联的话，指向父设备
#endif
#ifdef CONFIG_GENERIC_IRQ_DEBUGFS
    struct dentry       *debugfs_file;
#endif

    /* reverse map data. The linear map gets appended to the irq_domain */
    irq_hw_number_t hwirq_max;  //IRQ domain支持中断数量的最大值
    unsigned int revmap_direct_max_irq;
    unsigned int revmap_size;   //线性映射的大小
    struct radix_tree_root revmap_tree; //Radix Tree映射的根节点
    unsigned int linear_revmap[];   //线性映射用到的查找表
};

struct irq_domain_ops {
    int (*match)(struct irq_domain *d, struct device_node *node,
             enum irq_domain_bus_token bus_token);      // 用于中断控制器设备与IRQ domain的匹配
    int (*select)(struct irq_domain *d, struct irq_fwspec *fwspec,
              enum irq_domain_bus_token bus_token);
    int (*map)(struct irq_domain *d, unsigned int virq, irq_hw_number_t hw);    //用于硬件中断号与Linux中断号的映射
    void (*unmap)(struct irq_domain *d, unsigned int virq);
    int (*xlate)(struct irq_domain *d, struct device_node *node,
             const u32 *intspec, unsigned int intsize,
             unsigned long *out_hwirq, unsigned int *out_type);     //通过device_node，解析硬件中断号和触发方式

#ifdef  CONFIG_IRQ_DOMAIN_HIERARCHY
    /* extended V2 interfaces to support hierarchy irq_domains */
    int (*alloc)(struct irq_domain *d, unsigned int virq,
             unsigned int nr_irqs, void *arg);
    void (*free)(struct irq_domain *d, unsigned int virq,
             unsigned int nr_irqs);
    void (*activate)(struct irq_domain *d, struct irq_data *irq_data);
```

```
    void (*deactivate)(struct irq_domain *d, struct irq_data *irq_data);
    int (*translate)(struct irq_domain *d, struct irq_fwspec *fwspec,
             unsigned long *out_hwirq, unsigned int *out_type);
#endif
};
```

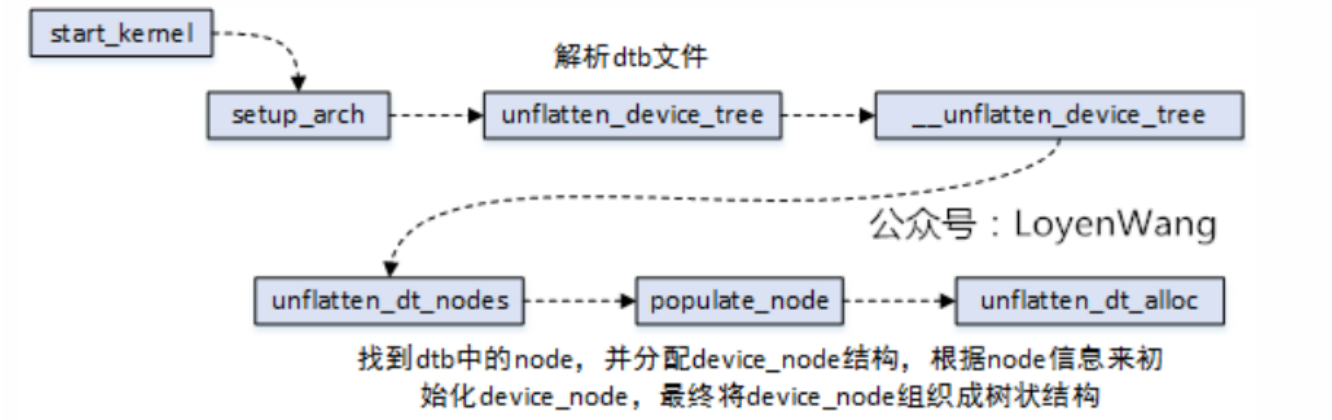
4.4 GIC控制器的初始化

前两节介绍了的GIC的硬件工作原理和Linux系统的通用中断子系统的框架基本实现原理。ARM平台的设备信息，都是通过`Device Tree`设备树来添加，对于TI-AM5728的设备树信息放在`arch/arm/boot/dts/`下，描述GIC的中断控制器信息为：

```
/*arch\arm\boot\dts\dra7.dtsi*/
gic: interrupt-controller@48211000 {
        compatible = "arm,cortex-a15-gic";
        interrupt-controller;
        #interrupt-cells = <3>;
        reg = <0x0 0x48211000 0x0 0x1000>,
              <0x0 0x48212000 0x0 0x2000>,
              <0x0 0x48214000 0x0 0x2000>,
              <0x0 0x48216000 0x0 0x2000>;
        interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(2) | IRQ_TYPE_LEVEL_HIGH)>;
        interrupt-parent = <&gic>;
};
```

- `compatible`字段：用于与具体的驱动来进行匹配，比如图片中`arm,cortex-a15-gic`，可以根据这个名字去匹配对应的驱动程序；
- `interrupt-cells`字段：用于指定编码一个中断源所需要的单元个数，这个值为3。比如在外设在设备树中添加中断信号时，通常能看到类似`interrupts = <0 23 4>;`的信息，第一个单元0，表示的是中断类型（`1：PPI，0：SPI`），第二个单元23表示的是中断号，第三个单元4表示的是中断触发的类型；
- `reg`字段：描述中断控制器的地址信息以及地址范围，比如图片中分别制定了`GIC Distributor(GICD)`和`GIC CPU Interface(GICC)`的地址信息；
- `interrupt-controller`字段：表示该设备是一个中断控制器，外设可以连接在该中断控制器上；
- 关于设备数的各个字段含义，详细可以参考`Documentation/devicetree/bindings`下的对应信息；

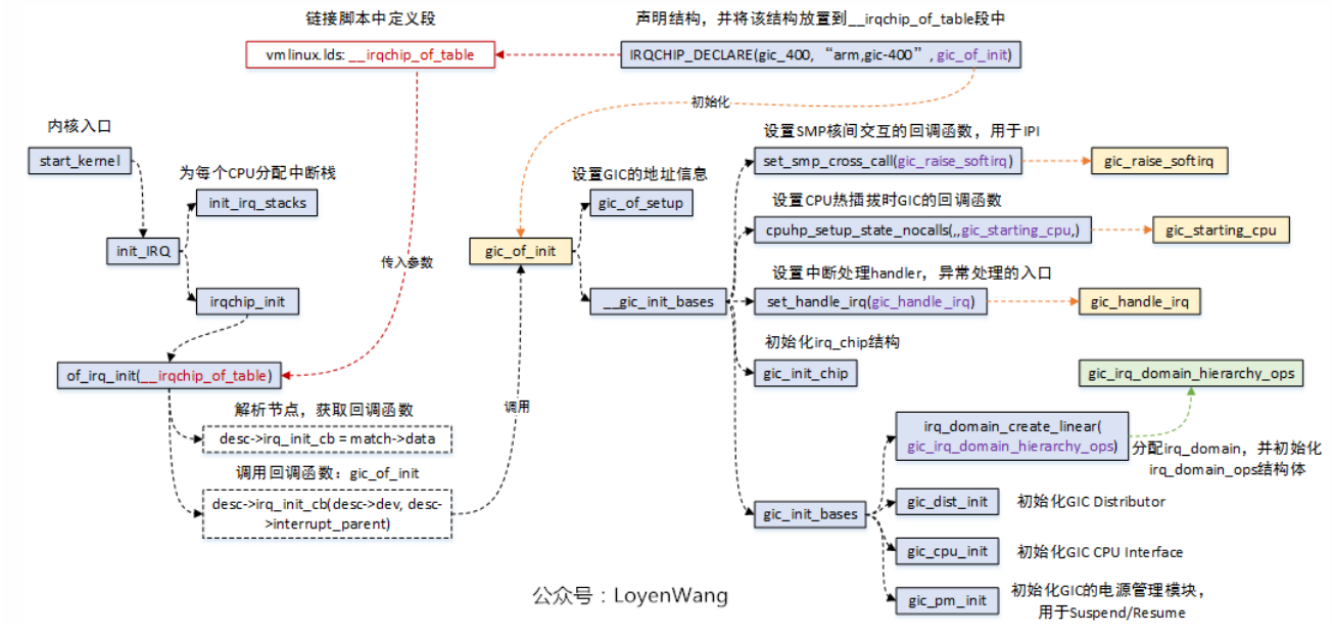
设备树的信息，是怎么添加到系统中的呢？`Device Tree`最终会编译成`dtb`文件，并通过Uboot传递给内核，在内核启动后会将`dtb`文件解析成`device_node`结构,此处引用网上的一章流程图：



- 设备树的节点信息，最终会变成`device_node`结构，在内存中维持一个树状结构；
- 设备与驱动会根据compatible字段进行匹配

4.5 GIC驱动流程分析

GIC驱动的执行流程如下图所示：

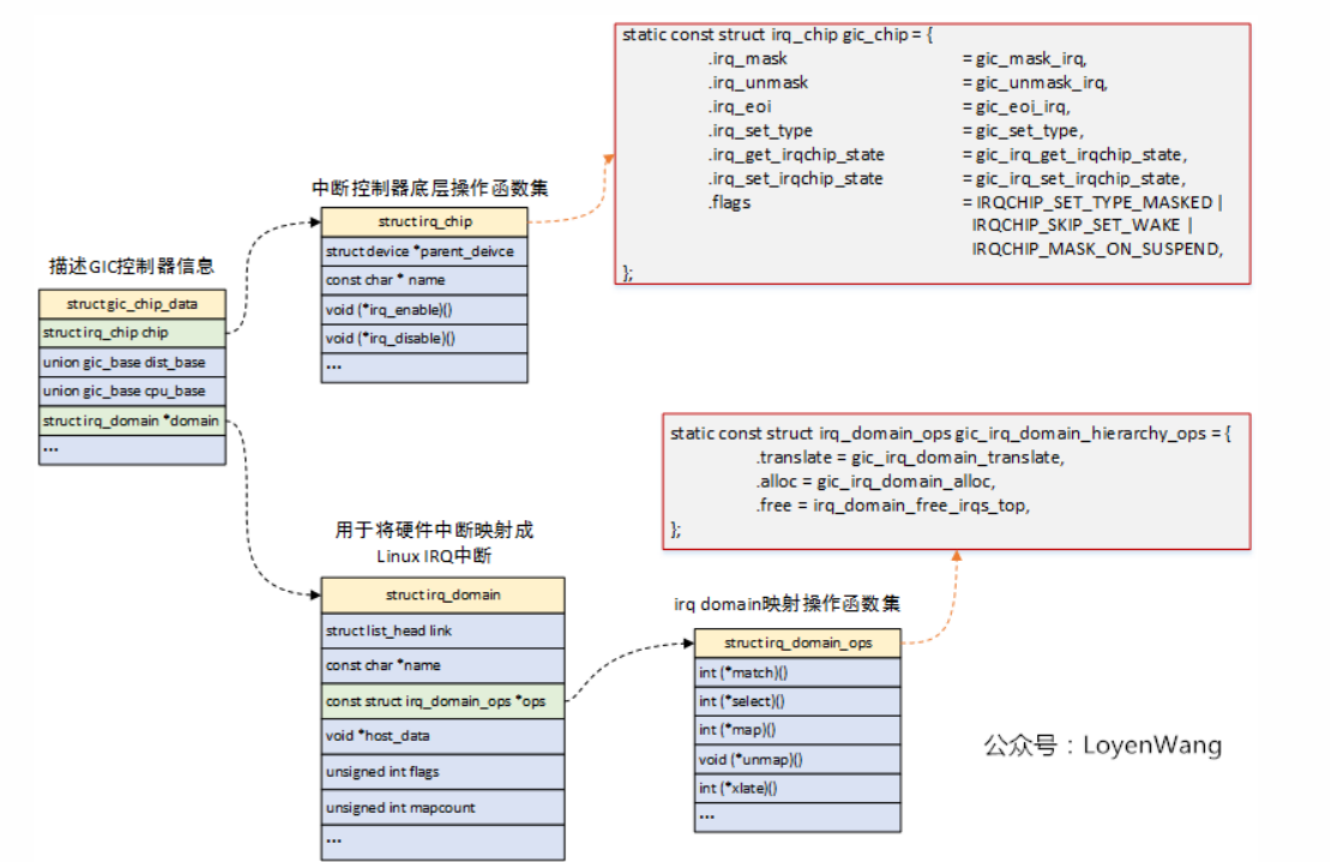


- 在连接脚本中vmlinux.lds，定义了一个`__irqchip_of_table`段，该段用于存放中断控制器信息，最终用来匹配设备；
- 在GIC驱动程序中，使用`IRQCHIP_DECLARE`宏来声明结构信息，包括`compatible`字段和回调函数，该宏会将这个结构放置到`__irqchip_of_table`字段中；
- 在内核启动初始化中断的函数中，`of_irq_init`函数回去查找设备节点信息，该函数的传入参数就是`__irqchip_of_table`段，由于`IQCHIP_DECLARE`已经将信息填充好了，`of_irq_init`函数会根据`arm,cortex-a15-gic`去查找相应的设备节点，并获取设备的信息。中断控制器也存在级联的情况，`of_irq_init`函数也处理了这种情况；
- `or_irq_init`函数中，最终会回调`IRQCHIP_DECLARE`声明的回调函数，也就是`gic_of_init`，而这个函数就是GIC驱动的初始化入口函数了；

- GIC的工作，本质上是由中断信号来驱动，因此驱动本身的工作就是完成各类信息的初始化，注册好相应的回调函数，以便能在信号到来之时去执行；
- `set_smp_process_call` 设置 `__smp_cross_call` 函数指向 `gic_raise_softirq`，本质上就是通过软件来触发GIC的 `SGI中断`，用于核间交互；
- `cpuhp_setup_state_nocalls` 函数，设置好CPU进行热插拔时GIC的回调函数，以便在CPU热插拔时做相应处理；
- `set_handle_irq` 函数的设置很关键，它将全局函数指针 `handle_arch_irq` 指向了 `gic_handle_irq`，而处理器在进入中断异常时，会跳转到 `handle_arch_irq` 执行，所以，可以认为它就是中断处理的入口函数了；
- 驱动中完成了各类函数的注册，此外还完成了 `irq_chip`，`irq_domain` 等结构体的初始化，这些结构在下文会进一步分析；
- 最后，完成GIC硬件模块的初始化设置，以及电源管理相关的注册等工作；

```
IRQCHIP_DECLARE(cortex_a15_gic, "arm,cortex-a15-gic", gic_of_init);
```

4.6 数据结构分析

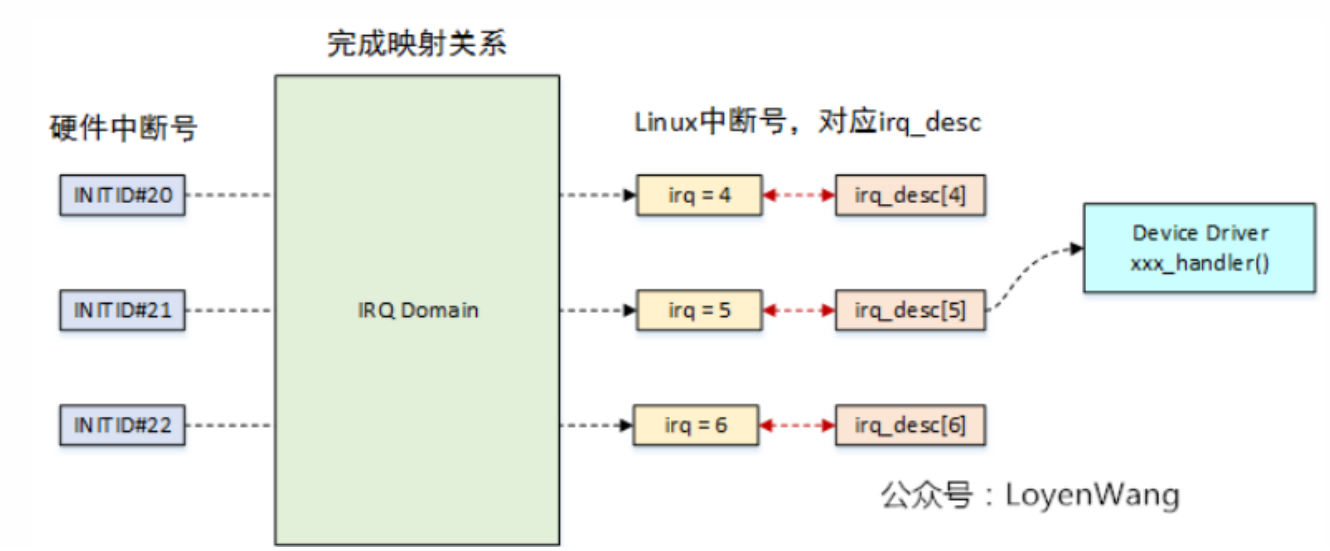


- GIC驱动中，使用 `struct gic_chip_data` 结构体来描述GIC控制器的信息，整个驱动都是围绕着该结构体的初始化，驱动中将函数指针都初始化好，实际的工作是由中断信号触发，也就是在中断来临的时候去进行回调；
- `struct irq_chip` 结构，描述的是中断控制器的底层操作函数集，这些函数集最终完成对控制器硬件的操作；
- `struct irq_domain` 结构，用于硬件中断号和Linux IRQ中断号（virq，虚拟中断号）之间的映射；

之前的章节已经对中断控制器的抽象struct irq_chip的结构进行了说明，此处不再说明。

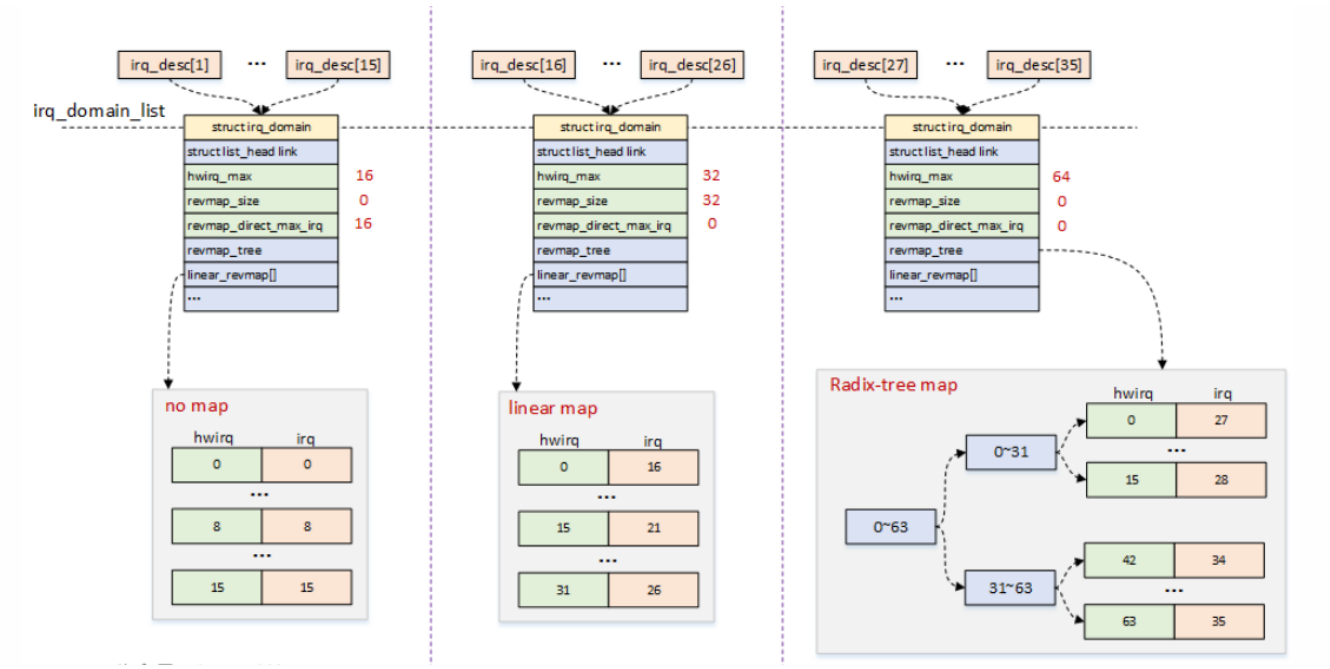
4.6.1 IRQ domian

IRQ domain用于将硬件的中断号转换成Linux系统中的中断号（`virtual irq,virq`），如下图：



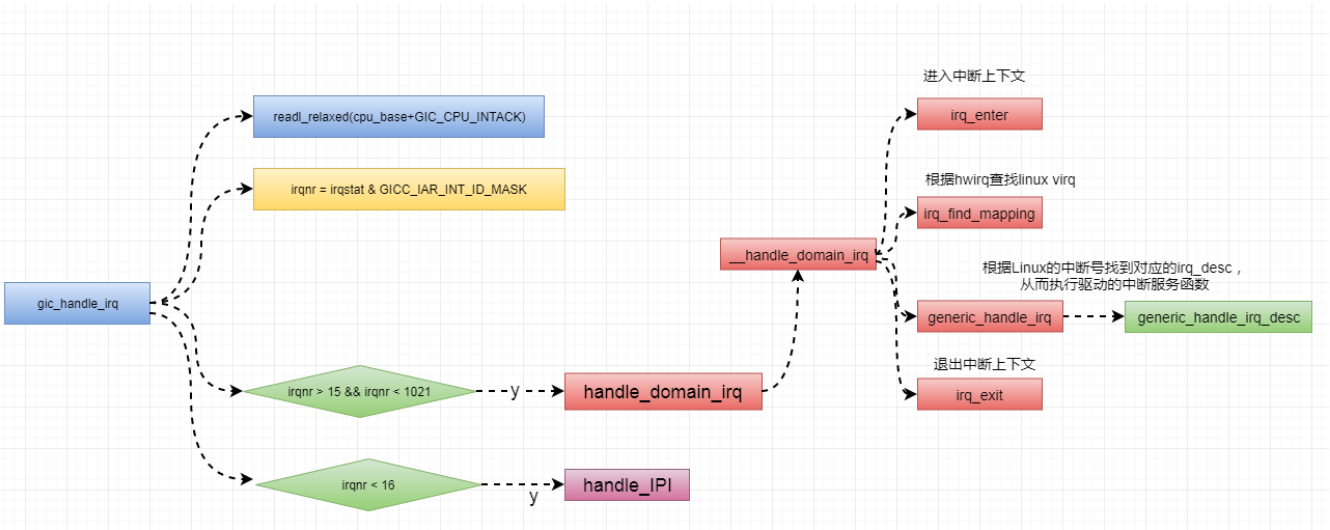
- 每个中断控制器都对应一个IRQ Domain；
- 中断控制器驱动通过`irq_domain_add_*()`接口来创建IRQ Domain；
- IRQ Domain支持三种映射方式：linear map（线性映射），tree map（树映射），no map（不映射）；
    1. linear map：维护固定大小的表，索引是硬件中断号，如果硬件中断最大数量固定，并且数值不大，可以选择线性映射；
    2. tree map：硬件中断号可能很大，可以选择树映射；
    3. no map：硬件中断号直接就是Linux的中断号；

这三种映射方式如下图：



- 图中描述了三种中断控制器对用三种不同的映射方式
- 不管使用哪种映射方式，控制器中的硬件中断号是一样，只是最后得到的Linux的虚拟中断号算法不一样，但是硬件中断号所对应的虚拟中断号也是唯一的。

在4.3节中说过中断入口函数的分析，有默认的中断处理函数，也可通过`set_handle_irq`来设置中断入口函数的函数指针，在GIC控制器初始化时，就调用了此函数重新传入了中断函数的指针，在4.5章节中已经说明，传入的中断函数为gic_handle_irq，分析此函数的执行流程，如下图：



- 当中断出发时，处理器会去异常向量表找到对应的入口函数，前面的章节已经分析过，会跳转到`irq_handler`然后执行`handle_arch_irq`的函数指针，在GIC控制器初始化的时候已经通过`set_handle_irq`设置它指向了`gic_handle_irq`，因为当中断发生时，会跳转到`gic_handle_irq`处执行。
- `gic_handle_irq`函数处理时会分为两种情况，一种是外设触发 的中断，硬件中断号在`16~1020`之间，一种是软件触发的中断，用于处理器之间的交互，硬件中断号在16以内。

```
static void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
    u32 irqstat, irqnr;
    struct gic_chip_data *gic = &gic_data[0];
    void __iomem *cpu_base = gic_data_cpu_base(gic);

    do {
        irqstat = readl_relaxed(cpu_base + GIC_CPU_INTACK);
        irqnr = irqstat & GICC_IAR_INT_ID_MASK;

        if (likely(irqnr > 15 && irqnr < 1021)) {
            if (static_key_true(&supports_deactivate))
                writel_relaxed(irqstat, cpu_base + GIC_CPU_EOI);
            handle_domain_irq(gic->domain, irqnr, regs);
            continue;
        }
        if (irqnr < 16) {
            writel_relaxed(irqstat, cpu_base + GIC_CPU_EOI);
            if (static_key_true(&supports_deactivate))
                writel_relaxed(irqstat, cpu_base + GIC_CPU_DEACTIVATE);
#ifdef CONFIG_SMP
```

```
......
            smp_rmb();
            handle_IPI(irqnr, regs);
#endif
            continue;
        }
        break;
    } while (1);
}
```

- 外设触发中断后，根据`irq domain`去查找对应的Linux IRQ中断号，进而得到中断描述符`irq_desc`，最终也就能调用到外设申请的中断处理函数了。

4.7 驱动程序分析

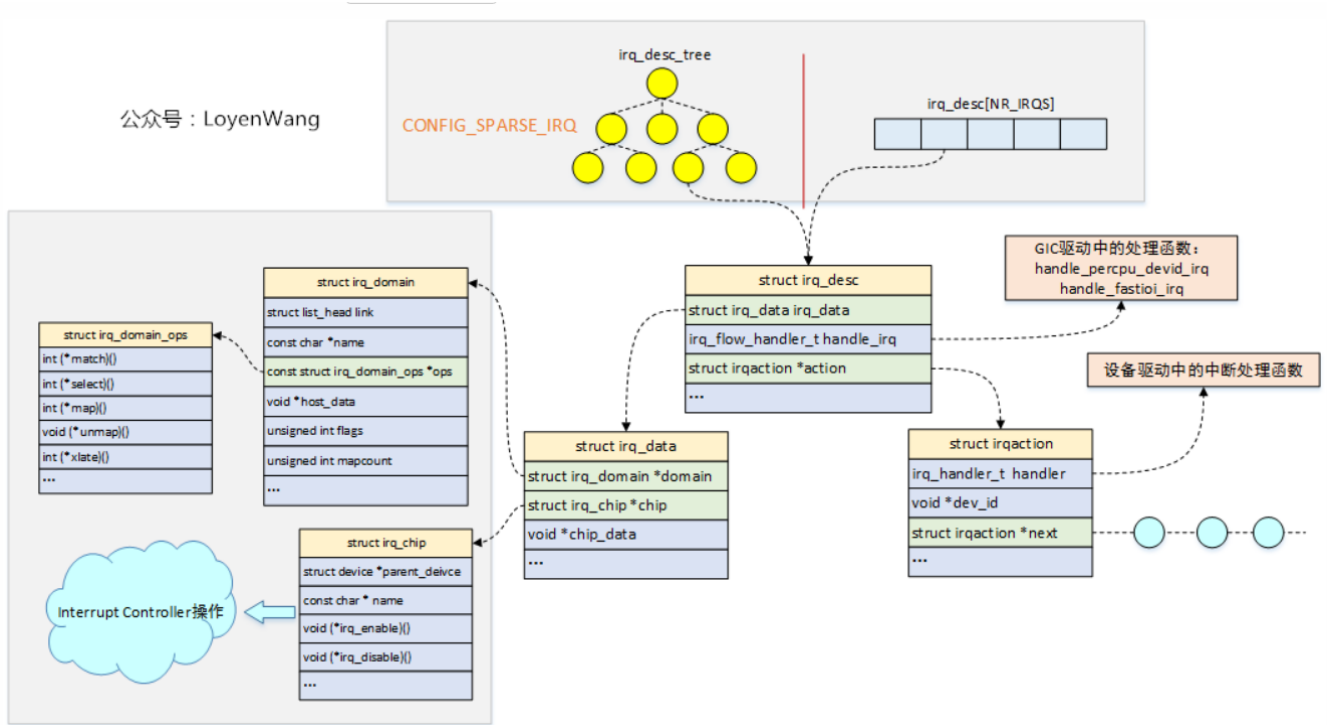
4.7.1 概述

这一小节主要围绕两个话题进行分析：

- 用户怎么使用中断，包括申请中断和注册？
- 外设触发中断信号时，是怎么调用到用户注册的中断处理函数？

4.7.2 数据结构分析

在内核中用一个结构体描述符来描述一个中断`struct irq_desc`，下面借用一张图来说明这个结构体在内核中的状态:



- Linux内核的中断处理，围绕着中断描述符结构`struct irq_desc`展开，内核提供了两种中断描述符组织形式：

- 如果内核配置了`CONFIG_SPARES_IRQ`宏（中断编号不连续），中断描述符以`radix-tree`来组织，用户在初始化时进行动态分配，然后再插入`radix-tree`中；
- 如果内核没有配置`CONFIG_SPEARSE_IRQ`宏（中断编号连续）,中断描述符以数组的形式组织，并且在内核初始化的时候已经分配好。
- 但不管是那种形式，最终都是通过唯一的`linux irq`来找到对应的中断描述符。

- 图的左侧灰色部分，主要在中断控制器驱动中进行初始化设置，包括各个结构中函数指针的指向等，其中`struct irq_chip`用于对中断控制器的硬件操作，`struct irq_domain`与中断控制器对应，完成的工作是硬件中断号到`Linux irq`的映射；
- 图的上侧灰色部分，中断描述符的创建（这里指`CONFIG_SPARSE_IRQ`），主要在获取设备中断信息的过程中完成的，从而让设备树中的中断能与具体的中断描述符`irq_desc`匹配；
- 图中剩余部分，在设备申请注册中断的过程中进行设置，比如`struct irqaction`中`handler`的设置，这个用于指向我们设备驱动程序中的中断处理函数了；

中断的处理主要有以下几个功能模块：

1. 硬件中断号到`Linux irq`中断号的映射，并创建好`irq_desc`中断描述符（在中断控制器初始化已创建，如果没有创建则需要在驱动程序申请中断的时候动态创建）；
2. 中断注册时，先获取设备的中断号，根据中断号找到对应的`irq_desc`，并将设备的中断处理函数添加到`irq_desc`中；
3. 设备触发中断信号时，根据硬件中断号得到`Linux irq`中断号，找到对应的`irq_desc`，最终调用到设备的中断处理函数；

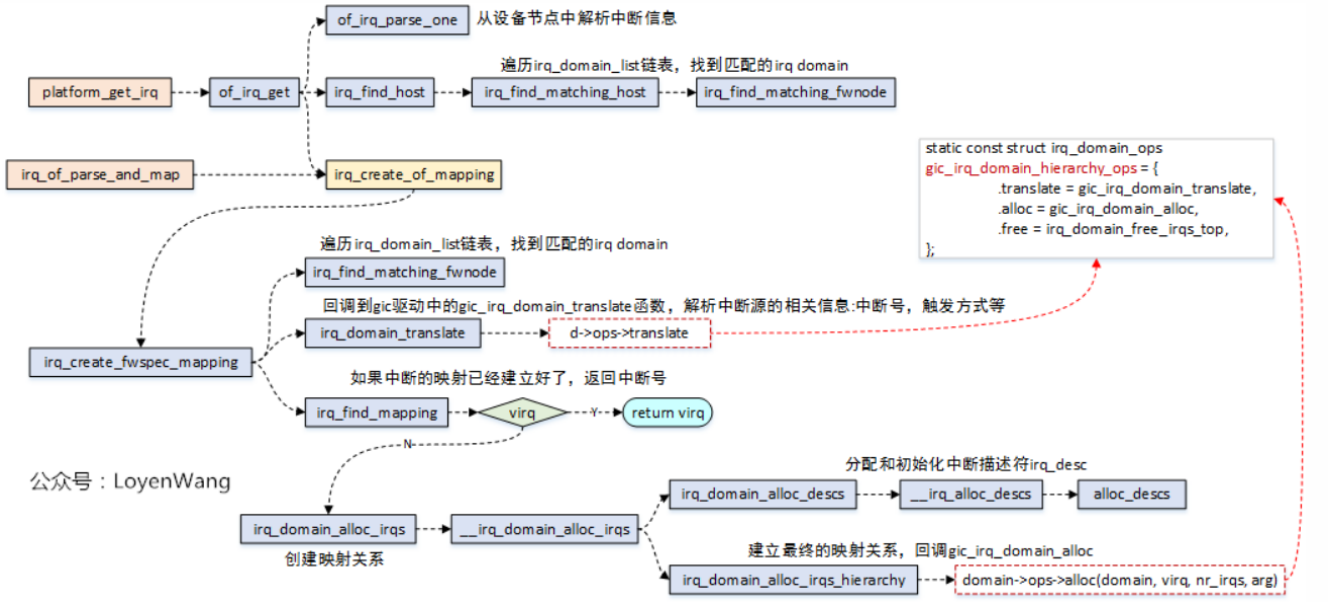
4.7.3 驱动程序中断流程分析

4.7.3.1 中断注册

在驱动程序中主要有两个接口API来注册申请中断，`request_irq()`和`request_threaded_irq()`。而这两个函数的参数都需要用到Linux irq中断号作为参数,它是怎么获取的呢？

> 1、先看下硬件中断号到`Linux irq`中断号的映射

写过设备树驱动程序的都知道，获取到设备树中的中断号的函数为`platform_get_irq`，下面对此函数进行分析,如下图：



- 硬件设备的中断信息都在设备树 `device tree` 中进行了描述，在系统启动过程中，这些信息都已经加载到内存中并得到了解析；
- 驱动中通常会使用 `platform_get_irq` 或 `irq_of_parse_and_map` 接口，去根据设备树的信息去创建映射关系（硬件中断号到 `linux irq` 中断号映射）；
- `struct irq_domain` 用于完成映射工作，因此在 `irq_create_fwspec_mapping` 接口中，会先去找到匹配的 `irq domain`，再去回调该 `irq domain` 中的函数集，通常 `irq domain` 都是在中断控制器驱动中初始化的，以 `ARM GICv2` 为例，最终回调到 `gic_irq_domain_hierarchy_ops` 中的函数；
- 如果已经创建好了映射，那么可以直接进行返回 `linux irq` 中断号了，否则的话需要 `irq_domain_alloc_irqs` 来创建映射关系；
- `irq_domain_alloc_irqs` 完成两个工作：
  - 针对 `linux irq` 中断号创建一个 `irq_desc` 中断描述符；
  - 调用 `domain->ops->alloc` 函数来完成映射，在 `ARM GICv2` 驱动中对应 `gic_irq_domain_alloc` 函数，这个函数很关键，所以下文介绍一下,函数执行流程如下：

  - `gic_irq_domain_translate`：负责解析出设备树中描述的中断号和中断触发类型（边缘触发、电平触发等）；
  - `gic_irq_domain_map`：将硬件中断号和linux中断号绑定到一个结构中，也就完成了映射，此外还绑定了 `irq_desc` 结构中的其他字段，最重要的是设置了 `irq_desc->handle_irq` 的函数指针，这个最终是中断响应时往上执行的入口，这个是关键，下文讲述中断处理过程时还会提到；
  - 根据硬件中断号的范围设置 `irq_desc->handle_irq` 的指针，共享中断入口为 `handle_fasteoi_irq`，私有中断入口为 `handle_percpu_devid_irq`；

  上述函数执行完成后，完成了两大工作：

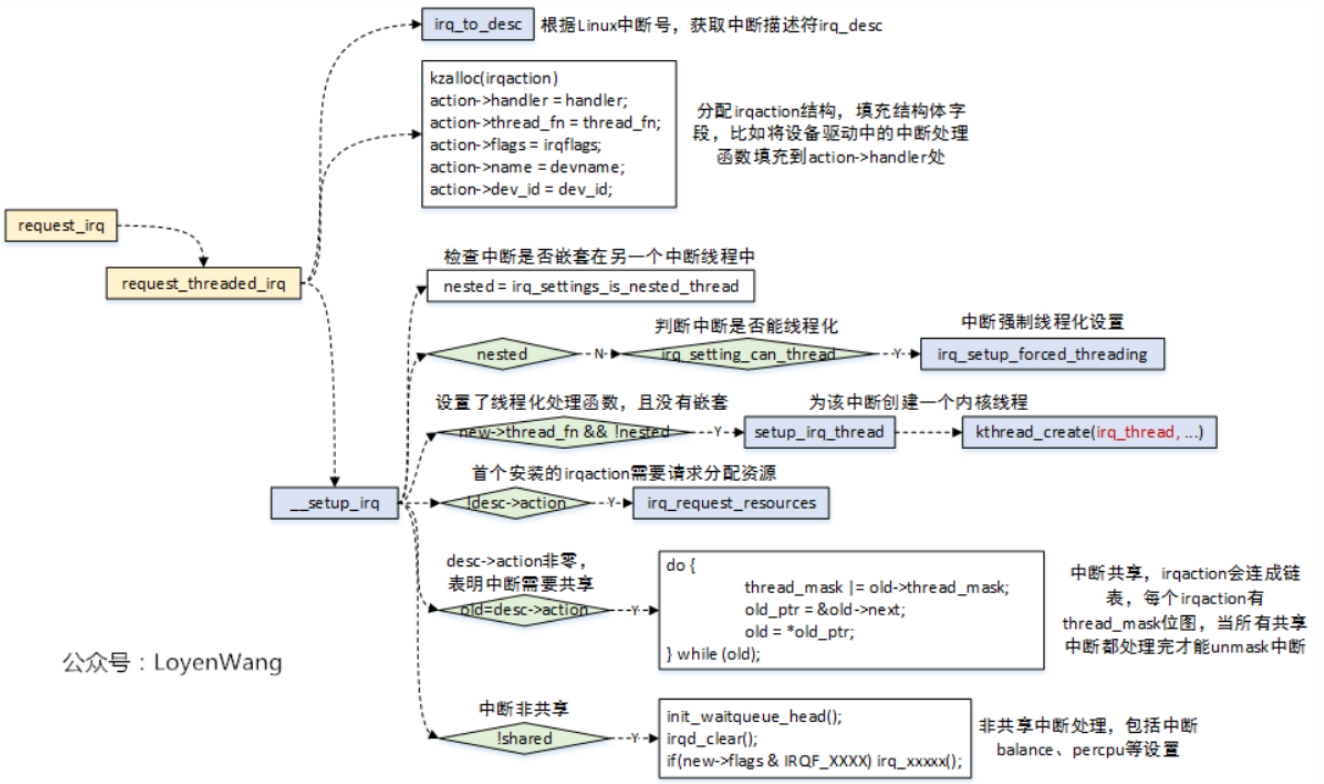
  1. 硬件中断号与Linux中断号完成映射，并为Linux中断号创建了 `irq_desc` 中断描述符；
  2. 数据结构的绑定及初始化，关键的地方是设置了中断处理往上执行的入口；

> 2、中断是怎么来注册的

设备驱动中，获取到了 `irq` 中断号后，通常就会采用 `request_irq/request_threaded_irq` 来注册中断，其中 `request_irq` 用于注册普通处理的中断，`request_threaded_irq` 用于注册线程化处理的中断，在注册中断函数中会传入中断标志位，标志位如下：

```
#define IRQF_SHARED      0x00000080              //多个设备共享一个中断号，需要外设硬件支持
#define IRQF_PROBE_SHARED    0x00000100          //中断处理程序允许sharing mismatch发生
#define __IRQF_TIMER         0x00000200         //时钟中断
#define IRQF_PERCPU      0x00000400              //属于特定CPU的中断
#define IRQF_NOBALANCING     0x00000800              //禁止在CPU之间进行中断均衡处理
#define IRQF_IRQPOLL         0x00001000              //中断被用作轮训
#define IRQF_ONESHOT         0x00002000              /*一次性触发的中断，不能嵌套，
```

```
                                                *1）在硬件中断处理完成后才能打开中断；
* 2）在中断线程化中保持关闭状态，直到该                                                *中断
源上的所有thread_fn函数都执行完
                                                */
#define IRQF_NO_SUSPEND      0x00004000        //系统休眠唤醒操作中，不关闭该中断
#define IRQF_FORCE_RESUME    0x00008000        //系统唤醒过程中必须强制打开该中断
#define IRQF_NO_THREAD       0x00010000        //禁止中断线程化
#define IRQF_EARLY_RESUME    0x00020000        /*系统唤醒过程中在syscore阶段resume，
  *而不用等到设备resume阶段
                                                */
#define IRQF_COND_SUSPEND    0x00040000        /*与NO_SUSPEND的用户共享中断时，执行本
*设备的中断处理函数
                                                */
```

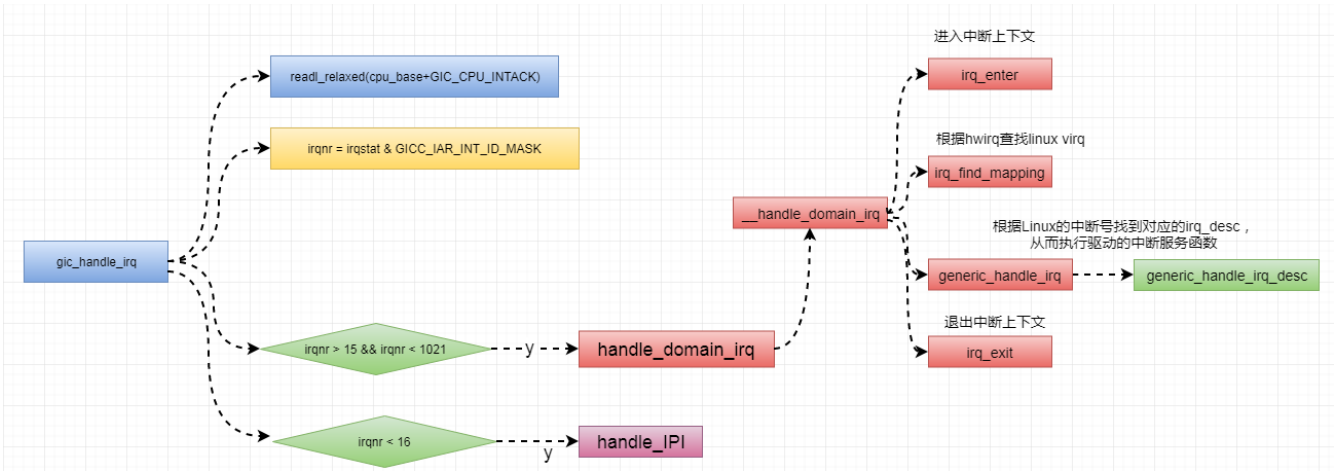


- `request_irq`也是调用`request_threaded_irq`，只是在传参的时候，线程处理函数`thread_fn`函数设置成NULL；
- 由于在硬件中断号和Linux中断号完成映射后，`irq_desc`已经创建好，可以通过`irq_to_desc`接口去获取对应的`irq_desc`；
- 创建`irqaction`，并初始化该结构体中的各个字段，其中包括传入的中断处理函数赋值给对应的字段；
- `__setup_irq`用于完成中断的相关设置，包括中断线程化的处理：
  - 中断线程化用于减少系统关中断的时间，增强系统的实时性；
  - ARM64默认开启了`CONFIG_IRQ_FORCED_THREADING`，引导参数传入`threadirqs`时，则除了`IRQF_NO_THREAD`外的中断，其他的都将强制线程化处理；

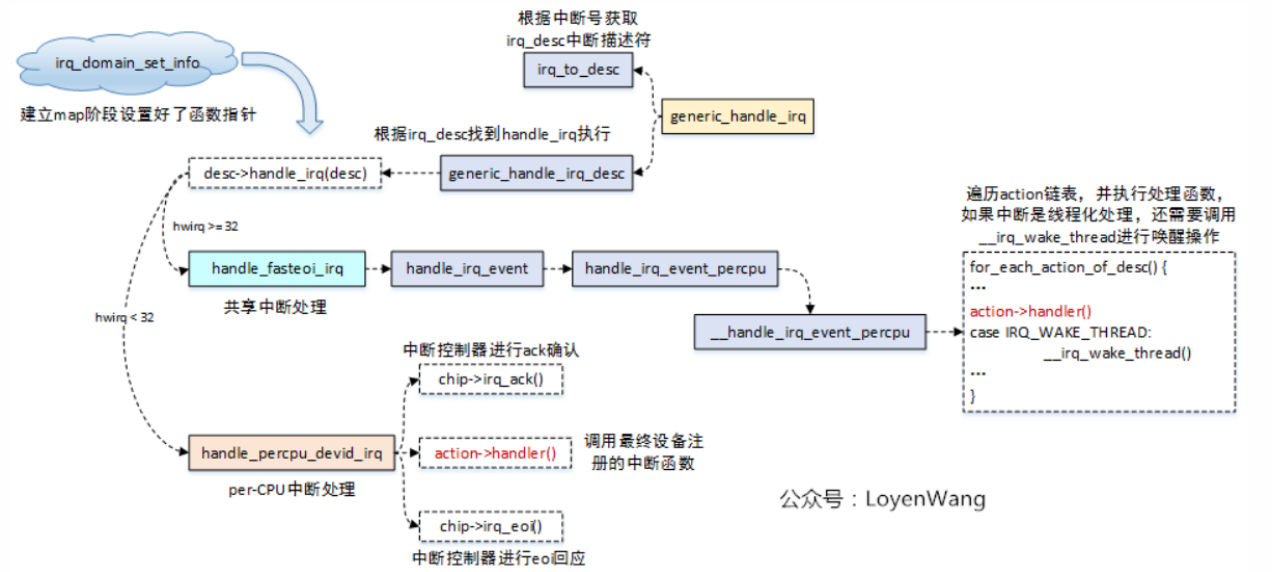
- 中断线程化会为每个中断都创建一个内核线程，如果中断进行共享，对应`irqaction`将连接成链表，每个`irqaction`都有`thread_mask`位图字段，当所有共享中断都处理完成后才能`unmask`中断，解除中断屏蔽；

3、中断处理

前面已经完成了中断控制器和中断信号的申请，当中断信号到来的时候，就会进行中断的处理工作，之前的章节已经分析过，当中断到来时会到`gic_handle_irq`处执行，处理流程图如下：

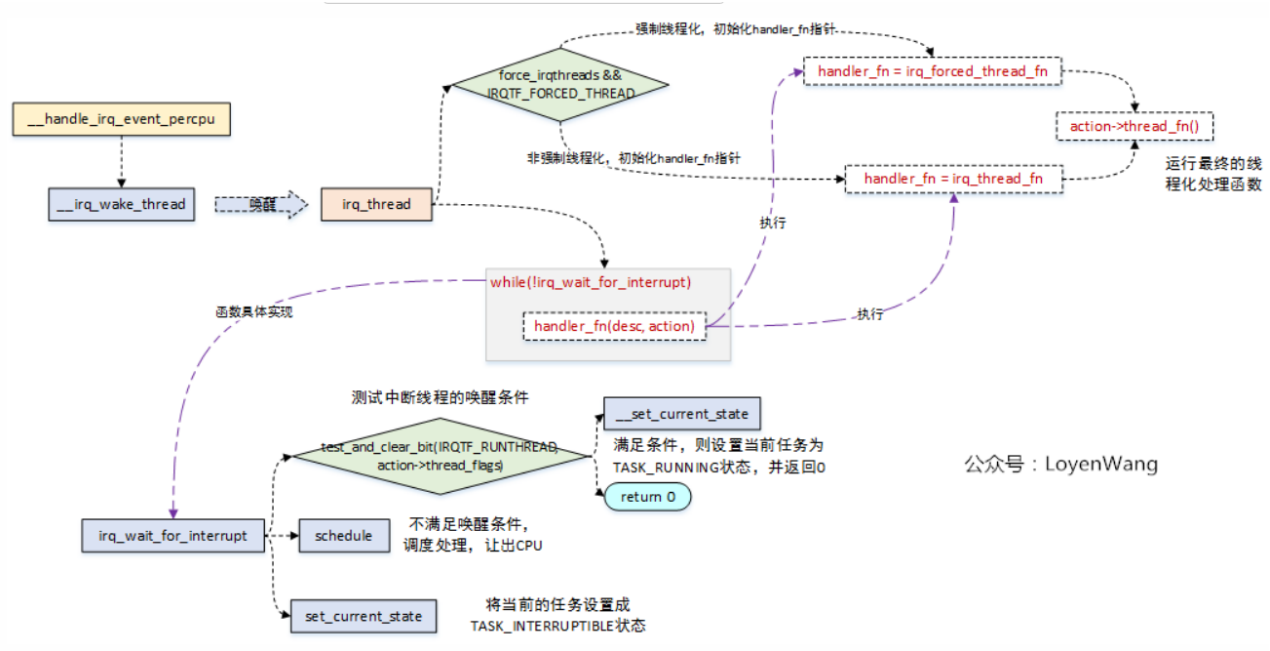


- 中断收到之后，首先会跳转到异常向量表的入口处，进而逐级进行回调处理，最终调用到`generic_handle_irq`来进行中断处理，处理流程图如下：



- `generic_handle_irq`函数最终会调用到`desc->handle_irq()`，这个也就是对应到上文中在建立映射关系的过程中，调用`irq_domain_set_info`函数，设置好了函数指针，也就是`handle_fasteoi_irq`和`handle_percpu_devid_irq`；
- `handle_fasteoi_irq`：处理共享中断，并且遍历`irqaction`链表，逐个调用`action->handler()`函数，这个函数正是设备驱动程序调用`request_irq/request_threaded_irq`接口注册的中断处理函数，此外如果中断线程化处理的话，还会调用`__irq_wake_thread()`唤醒内核线程；
- `handle_percpu_devid_irq`：处理per-CPU中断处理，在这个过程中会分别调用中断控制器的处理函数进行硬件操作，该函数调用`action->handler()`来进行中断处理；

- 中断线程的唤醒流程 `__handle_irq_event_percpu->__irq_wake_thread` ：



  - `__handle_irq_event_percpu->__irq_wake_thread` 将唤醒 `irq_thread` 中断内核线程；
  - `irq_thread` 内核线程，将根据是否为强制中断线程化对函数指针 `handler_fn` 进行初始化，以便后续进行调用；
  - `irq_thread` 内核线程将 `while(!irq_wait_for_interrupt)` 循环进行中断的处理，当满足条件时，执行 `handler_fn` ，在该函数中最终调用 `action->thread_fn` ，也就是完成了中断的处理；
  - `irq_wait_for_interrupt` 函数，将会判断中断线程的唤醒条件，如果满足了，则将当前任务设置成 `TASK_RUNNING` 状态，并返回0，这样就能执行中断的处理，否则就调用 `schedule()` 进行调度，让出CPU，并将任务设置成 `TASK_INTERRUPTIBLE` 可中断睡眠状态；

generic_handle_irq函数分析

```
int generic_handle_irq(unsigned int irq)
{
    /*通过传入的IRQ编号，找到对应的硬件中断号*/
    struct irq_desc *desc = irq_to_desc(irq);

    if (!desc)
        return -EINVAL;
    /*通过此函数执行相应的回调handle*/
    generic_handle_irq_desc(desc);
    return 0;
}
```

```
static inline void generic_handle_irq_desc(struct irq_desc *desc)
{
    /*执行中断流控函数gic-v2在初始化的时候，设置desc->handle_irq
     *绑定的是irq > 32,handle_fasteoi_irq,
     *irq < 32 : handle_percpu_devid_irq
     */
    desc->handle_irq(desc);
}
```

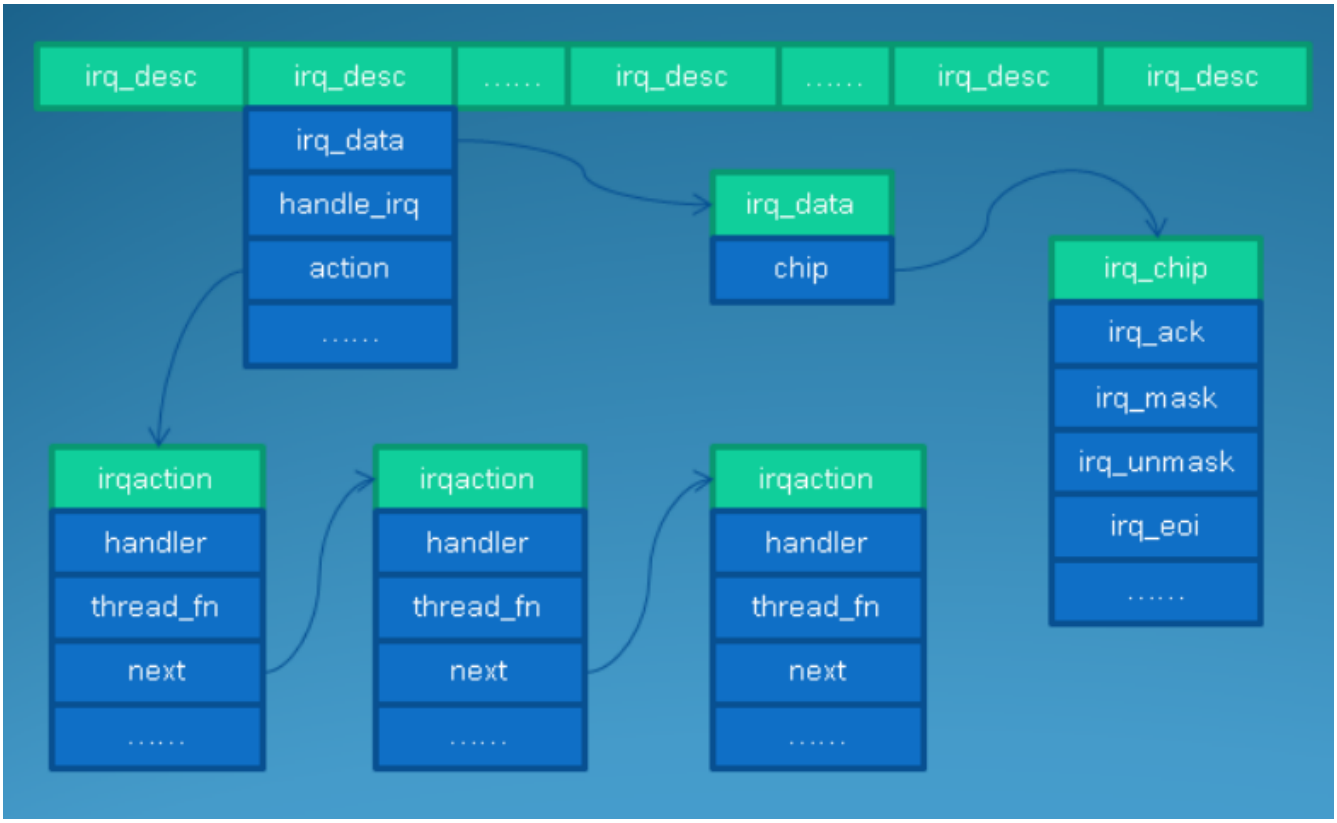
```
void handle_fasteoi_irq(struct irq_desc *desc)
{
    ......
    /*通过此函数去找到驱动程序中的中断服务函数*/
    handle_irq_event(desc);
    /*调用chip->eoi函数告知GIC控制器中断处理完成*/
    cond_unmask_eoi_irq(desc, chip);
    ......
}
```

```
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
    irqreturn_t ret;
    ......
    ret = handle_irq_event_percpu(desc);
    ......
    return ret;
}
```

```
irqreturn_t handle_irq_event_percpu(struct irq_desc *desc)
{
    irqreturn_t retval = IRQ_NONE;
    unsigned int flags = 0, irq = desc->irq_data.irq;
    /*通过irq_desc获取到action链表*/
    struct irqaction *action = desc->action;

    /* action might have become NULL since we dropped the lock */
    /*只有当多个设备共享同一个外设中断线时，action链表中才会有多个中断处理函数*/
    while (action) {
        irqreturn_t res;

        trace_irq_handler_entry(irq, action);
        /*执行驱动函数注册的中断服务函数*/
        res = action->handler(irq, action->dev_id);
        trace_irq_handler_exit(irq, action, res);

        if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pF enabled interrupts\n",
                  irq, action->handler))
            local_irq_disable();
```

```
        switch (res) {
        case IRQ_WAKE_THREAD:
            /*
             * Catch drivers which return WAKE_THREAD but
             * did not set up a thread function
             */
            if (unlikely(!action->thread_fn)) {
                warn_no_thread(irq, action);
                break;
            }
            /*如果注册中断函数时，传入了线程函数，则在此处唤醒线程服务函数*/
            __irq_wake_thread(desc, action);

            /* Fall through to add to randomness */
        case IRQ_HANDLED:
            flags |= action->flags;
            break;

        default:
            break;
        }

        retval |= res;
        action = action->next;
    }

    add_interrupt_randomness(irq, flags);

    if (!noirqdebug)
        note_interrupt(desc, retval);
    return retval;
}
```

最后`struct irq_desc`结构体被申请后的结构体内的数据结构关系如下图;



从图中我们可以知道gic控制器中对`irq > 32`的中断号的`irq_desc->handle_irq`的函数为`handle_fasteoi_irq`，`irq_data`保存了与控制器操作相关的数据结构，而`action`链表保存了驱动程序申请的中断函数指针`handler`和中断线程函数指针`thread_fn`。

> 4、总结

中断的处理，总体来说可以分为两部分来看：

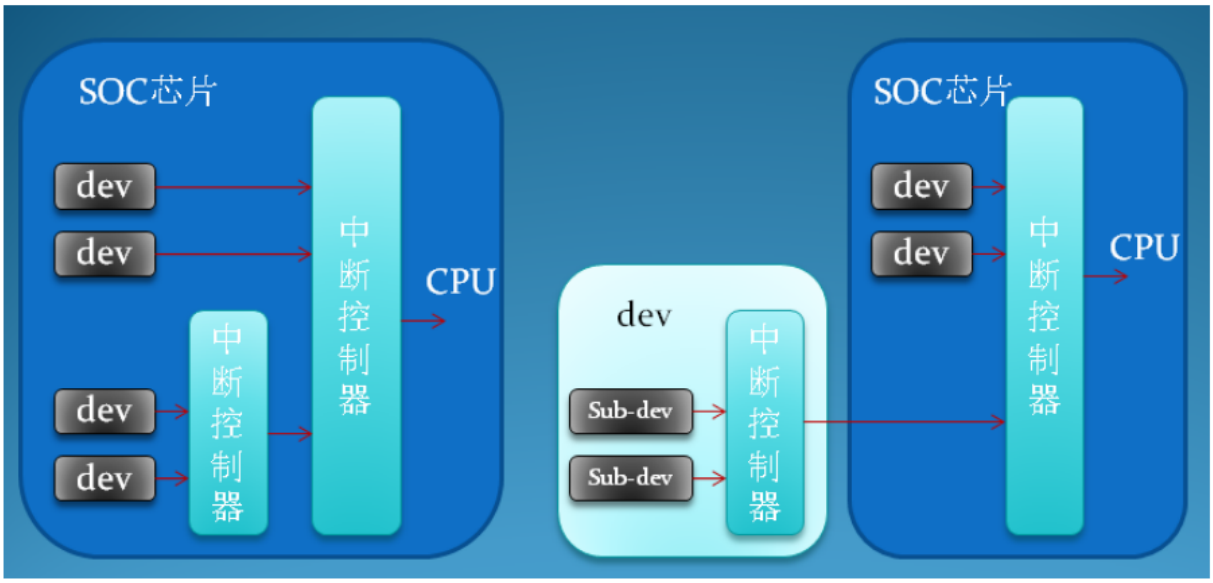
1. 从上到下：围绕`irq_desc`中断描述符建立好连接关系，这个过程就包括：中断源信息的解析（设备树），硬件中断号到Linux中断号的映射关系、`irq_desc`结构的分配及初始化（内部各个结构的组织关系）、中断的注册（填充`irq_desc`结构，包括handler处理函数）等，总而言之，就是完成静态关系创建，为中断处理做好准备；
2. 从下到上，当外设触发中断信号时，中断控制器接收到信号并发送到处理器，此时处理器进行异常模式切换，并逐步从处理器架构相关代码逐级回调。如果涉及到中断线程化，则还需要进行中断内核线程的唤醒操作，最终完成中断处理函数的执行。

5中断 控制器级联

在实际的SOC设备中，经常存在多个中断控制器，就会存在中断控制器的级联。我们把直接和CPU相连的中断控制器叫做根控制器，比如ARM-SOC的GIC中断控制器就称为根控制器。另外和根控制器相连的叫子控制器。根据子控制器的位置，我们把它们分为两种类型：

- 机器级别的级联，子控制器位于SOC内部，或者子控制器在SOC的外部，但是是某个板子系统的标准配置，如下图的左边所示；

- 设备级别的级联，子控制器位于某个外部设备中，用于汇集该设备发出的多个中断，如下图的右边所示；



对于机器级别的级联，一般在内核初始化GIC中断控制器的时候初始化该子控制器，因为事先已经知道子控制器的硬件连接信息，内核可以方便的为子控制器保留相应的`irq_desc`结构和`irq`编号，处理起来就比较简单。而对于设备级别的控制器的级联，驱动程序需要动态的决定组合设备中各个子设备的`irq`编号和`irq_desc`结构。下面我们只讨论机器级别的级联，但是设备级别的级联可以使用同样的原理来动态添加。要实现中断控制器的级联，需要使用内核提供的结构关键数据结构字段和通用逻辑层的API:

- `irq_desc->handle_irq`，irq的流控处理回调函数，子控制器在把多个irq汇集起来后，输出端连接到根控制器的其中一个irq中断线输入引脚，这意味着，每个子控制器的中断发生时，CPU一开始只会得到根控制器的irq编号，然后进入该irq编号对应的`irq_desc->handle_irq`回调函数，该回调函数不能设置为`handle_fasteoi_irq`，而是需要自己实现一个函数，该函数负责从子控制器中获取到子控制器发生中断的irq中断源，并计算出新的irq编号，然后调用新的irq编号所对应的`irq_desc->handle_irq`的回调，这时的回调。这个回调可以使用流控层提供的标准实现。
- `irq_set_chinaed_handler()`该函数用于设置根控制器与子控制器相连的irq所对应的`irq_desc->handle_irq`回调函数，并且设置IRQ_NOPROBE和IRQ_NOTHREAD以及IRQ_NOREQUEST标志，这几个标志保证驱动程序不会错误的申请该irq，因为该irq已经被作为级联irq使用。
- `irq_set_chip_and_handler()`该函数同时设置irq_desc中的handle_irq的回调和irq_chip指针。

下面以一个简单的代码列子来说明情况，

```
int __init xxx_init_irq_int(void)
{
  int irq;

  for (irq = IRQ_EINT(0); irq <= IRQ_EINT(15); irq++)
      irq_set_chip(irq, &xxx_irq_gic_int);

  for (irq = IRQ_EINT(16); irq <= IRQ_EINT(31); irq++) {
      irq_set_chip_and_handler(irq, &xxx_sub_irq_eint, handle_level_irq);
      set_irq_flags(irq, IRQF_VALID);
  }
```

```
    irq_set_chained_handler(IRQ_EINT16_31, sub_irq_demux_int16_31);
    return 0;
```

该SOC芯片的外部中断：`IRQ_EINT(0)`到`IRQ_EINT(15)`，每个引脚对应一个根控制器的irq中断线，它们是正常的irq，无需级联。`IRQ_EINT(16)`到`IRQ_EINT(31)`经过子控制器汇集后，统一连接到根控制器编号为`IRQ_EINT16_31`这个中断线上。可以看到，子控制器对应的`irq_chip`是`xxx_sub_irq_eint`，子控制器的irq默认设置为电平中断的流控处理函数`handle_level_irq`，它们通过API：irq_set_chained_handler进行设置。如果根控制器有128个中断线，`IRQ_EINT0--IRQ_EINT15`通常占据128内的某段连续范围，这取决于实际的物理连接。`IRQ_EINT16_31`因为也属于跟控制器，所以它的值也会位于128以内，但是`IRQ_EINT16--IRQ_EINT31`通常会在128以外的某段范围，这时，代表irq数量的常量NR_IRQS，必须考虑这种情况，定义出超过128的某个足够的数值。级联的实现主要依靠编号为`IRQ_EINT16_31`的流控处理程序：`sub_irq_demux_int16_31`，它的最终实现类似于以下代码：

```
static inline void sub_irq_demux_int16_31(unsigned int start)
{
    /*这里是去读中断子控制器，然后去判断IRQ_EINT(16)~IRQ_EINT(31)是具体哪个中断源触发中断*/
  u32 status = __raw_readl(xxx_EINT_PEND(EINT_REG_NR(start)));
  u32 mask = __raw_readl(xxx_EINT_MASK(EINT_REG_NR(start)));
  unsigned int irq;

  status &= ~mask;
  status &= 0xff;

  while (status) {
      irq = fls(status) - 1;
      generic_handle_irq(irq + start);
      status &= ~(1 << irq);
  }
}
```

在获得新的irq编号后，它的最关键的一句是调用了通用中断逻辑层的API：`generic_handle_irq`，这时它才真正地把中断控制权传递到中断流控层中来。