中断的总入口为 el0_irq()或者 el1_irq()。具体进哪个函数取决于当前的异常等级是 EL0 还是 EL1。

入口函数是:

arch\arm64\kernel\entry.S 的

```
el0_irq:
	kernel_entry 0
el0_irq_naked:
	gic_prio_irq_setup pmr=x20, tmp=x0
	enable_da_f

#ifdef CONFIG_TRACE_IRQFLAGS
	bl	trace_hardirqs_off
#endif

	ct_user_exit
#ifdef CONFIG_HARDEN_BRANCH_PREDICTOR
	tbz	x22, #55, 1f
	bl	do_el0_irq_bp_hardening
1:
#endif
	irq_handler

#ifdef CONFIG_TRACE_IRQFLAGS
	bl	trace_hardirqs_on
#endif
	b	ret_to_user
ENDPROC(el0_irq)
```

或者

```
el1_irq:
	kernel_entry 1
	gic_prio_irq_setup pmr=x20, tmp=x1
	enable_da_f

#ifdef CONFIG_ARM64_PSEUDO_NMI
	test_irqs_unmasked	res=x0, pmr=x20
	cbz	x0, 1f
	bl	asm_nmi_enter
1:
#endif

#ifdef CONFIG_TRACE_IRQFLAGS
	bl	trace_hardirqs_off
#endif

	irq_handler

#ifdef CONFIG_PREEMPT
	ldr	x24, [tsk, #TSK_TI_PREEMPT]	// get preempt count
alternative_if ARM64_HAS_IRQ_PRIO_MASKING
	/*
	 * DA_F were cleared at start of handling. If anything is set in DAIF,
	 * we come back from an NMI, so skip preemption
	 */
	mrs	x0, daif
	orr	x24, x24, x0
alternative_else_nop_endif
	cbnz	x24, 1f			// preempt count != 0 || NMI return path
	bl	preempt_schedule_irq		// irq en/disable is done inside
1:
#endif

#ifdef CONFIG_ARM64_PSEUDO_NMI
	/*
	 * When using IRQ priority masking, we can get spurious interrupts while
	 * PMR is set to GIC_PRIO_IRQOFF. An NMI might also have occurred in a
	 * section with interrupts disabled. Skip tracing in those cases.
	 */
	test_irqs_unmasked	res=x0, pmr=x20
	cbz	x0, 1f
	bl	asm_nmi_exit
1:
#endif

#ifdef CONFIG_TRACE_IRQFLAGS
#ifdef CONFIG_ARM64_PSEUDO_NMI
	test_irqs_unmasked	res=x0, pmr=x20
	cbnz	x0, 1f
#endif
	bl	trace_hardirqs_on
1:
#endif

	kernel_exit 1
ENDPROC(el1_irq)
```

可以看到，上面代码，最终都会执行到 irq_handler 宏。此函数定义如下：

```
/*
 * Interrupt handling.
 */
	.macro	irq_handler
	ldr_l	x1, handle_arch_irq
	mov	x0, sp
	irq_stack_entry
	blr	x1
	irq_stack_exit
	.endm
```

可以看到真正干活的是 handle_arch_irq 全局函数指针。这个函数指针，指向 ARM64 平台专用的 GIC 中断控制器的处理函数，即 gic_handle_irq()。

这个注册的过程，以 gic-v3 控制器为例：

在 drivers\irqchip\irq-gic-v3.c 的 gic_init_bases

```
static int __init gic_init_bases(void __iomem *dist_base,
				 struct redist_region *rdist_regs,
				 u32 nr_redist_regions,
				 u64 redist_stride,
				 struct fwnode_handle *handle)
{
```

内部调用了 set_handle_irq():

```
gic_data.has_rss = !!(typer & GICD_TYPER_RSS);
pr_info("Distributor has %sRange Selector support\n",
	gic_data.has_rss ? "" : "no ");

if (typer & GICD_TYPER_MBIS) {
	err = mbi_init(handle, gic_data.domain);
	if (err)
		pr_err("Failed to initialize MBIs\n");
}

set_handle_irq(gic_handle_irq);

gic_update_vlpi_properties();

gic_smp_init();
gic_dist_init();
gic_cpu_init();
gic_cpu_pm_init();

if (gic_dist_supports_lpis()) {
	its_init(handle, &gic_data.rdists, gic_data.domain);
	its_cpu_init();
}
```

在 irq\handle.c 里面:

```
int __init set_handle_irq(void (*handle_irq)(struct pt_regs *))
{
	if (handle_arch_irq)
		return -EBUSY;

	handle_arch_irq = handle_irq;
	return 0;
}
```

在 drivers\irqchip\irq-gic-v3.c 打开看 gic_handle_irq():

```
static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
	u32 irqnr;

	irqnr = gic_read_iar();

	if (gic_supports_nmi() &&
	    unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
		gic_handle_nmi(irqnr, regs);
		return;
	}

	if (gic_prio_masking_enabled()) {
		gic_pmr_mask_irqs();
		gic_arch_enable_irqs();
	}

	if (likely(irqnr > 15 && irqnr < 1020) || irqnr >= 8192) {
		int err;

		if (static_branch_likely(&supports_deactivate_key))
			gic_write_eoir(irqnr);
		else
			isb();

		err = handle_domain_irq(gic_data.domain, irqnr, regs);
		if (err) {
			WARN_ONCE(true, "Unexpected interrupt received!\n");
			gic_deactivate_unhandled(irqnr);
		}
		return;
	}
	if (irqnr < 16) {
		gic_write_eoir(irqnr);
		if (static_branch_likely(&supports_deactivate_key))
			gic_write_dir(irqnr);
#ifdef CONFIG_SMP
		/*
		 * Unlike GICv2, we don't need an smp_rmb() here.
		 * The control dependency from gic_read_iar to
		 * the ISB in gic_write_eoir is enough to ensure
		 * that any shared data read by handle_IPI will
		 * be read after the ACK.
		 */
		handle_IPI(irqnr, regs);
#else
		WARN_ONCE(true, "Unexpected SGI received!\n");
#endif
	}
} ? end gic_handle_irq ?
```

调用了 gic_read_iar。CPU 通过读取 GIC 控制器的 GICC_IAR(Interrupt Acknowledge Register) 寄存器，应答（ACK）该中断（GIC 的这个中断便从 pending 变成 active），并且可以得到当前发生中断的是哪一个硬件中断号 irqnr。
根据 irqnr 中断号的不同，处理函数不同。
当在 15-1020 之间或者大于 8192，属于 PPI 和 SPI 中断。使用 handle_domain_irq 处理。
当小于 16，属于 IPI 中断，使用 handle_IPI 处理。
PPI 和 SPI 中断是啥?

- PPI(Private Peripheral Interrupt)，Interrupt IDs16-31。私有中断，这种中断对每个 CPU 都是独立一份的，比如 per-core timer 中断。
- SPI(Shared Peripheral Interrupt)，Interrupt numbers 32-1020。最常用的外设中断，中断可以发给一个或者多个 CPU。

-即 per cpu 中断，还有外设中断。
IPI 中断是啥?
SGI(Software Generated Interrupt)，Interrupt IDs 0-15。系统一般用其来实现 IPI 中断。
-即软中断。(软件产生的中断)
另外还有 LPI:
LPI(Locality-specific Peripheral Interrupt)。
-基于 message 的中断，GICv2 和 GICv1 中不支持。
不管是用 handle_domain_irq 还是 handle_IPI，在此之前都涉及要调用 gic_write_eoir 和 gic_write_dir.
这两个函数分别往 ICC_EOIR1_EL1 和 ICC_DIR_EL1 寄存器中写入硬件中断号。
EOIR 和 DIR 寄存器的定义如下:
ICC_EOIR1_EL1, Interrupt Controller End Of Interrupt Register 1，对寄存器的写操作表示中断的结束
ICC_DIR_EL1, Interrupt Controller Deactivate Interrupt Register，对该寄存器的写操作将 deactive 指定的中断
那么问题来了? 假设读取到了一个 SPI 中断， 为什么一开始就写 EOI 表示中断结束，此时中断处理不是还没有执行么?
在 GIC v3 协议中定义， 处理完中断后，软件必须通知中断控制器已经处理了中断，以便状态机可以转换到下一个状态。
GICv3 架构将中断的完成分为 2 个阶段:
Priority Drop: 将运行优先级降回到中断之前的值。
Deactivation:更新当前正在处理的中断的状态机。 从活动状态转换到非活动状态。
这两个阶段可以在一起完成，也可以分为 2 步完成。 却决于 EOImode 的值。
如果 EOIMode = 0，对 ICC_EOIR1_EL1 寄存器的操作代表 2 个阶段（priority drop 和 deactivation）一起完成。
如果 EOImode = 1，对 ICC_EOIR1_EL1 寄存器的操作只会导致 Priority Drop， 如果想要表示中断已经处理完
成，还需要写 ICC_DIR_EL1。
所以回答上面的问题， 当前 Linux GIC 的代码，默认 irq chip 是 EIOmode=1,

```
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_branch_likely(&supports_deactivate_key))
        chip = &gic_eoimode1_chip;
```

所以单独的写 EOIR1_EL1 不是代表中断结束。

最后打开最常见的 handle_domain_irq 进一步分析:

```
static inline int handle_domain_irq(struct irq_domain *domain,
                unsigned int hwirq, struct pt_regs *regs)
{
    return __handle_domain_irq(domain, hwirq, true, regs);
}

/**
 * __handle_domain_irq - Invoke the handler for a HW irq belonging to a domain
 * @domain: The domain where to perform the lookup
 * @hwirq:  The HW irq number to convert to a logical one
 * @lookup: Whether to perform the domain lookup or not
 * @regs: Register file coming from the low-level handling code
 *
 * Returns:   0 on success, or -EINVAL if conversion has failed
 */
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
    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 {
        generic_handle_irq(irq);
    }

    irq_exit();
    set_irq_regs(old_regs);
    return ret;
} ?end __handle_domain_irq ?
```

(1) irq_enter 显式的告诉 Linux 内核现在要进入中断上下文了。

```
#define __irq_enter() \
do { \
account_irq_enter_time(current); \
preempt_count_add(HARDIRQ_OFFSET); \
trace_hardirq_enter(); \
} while (0)
```

__irq_enter 宏通过 preempt_count_add()增加当前进程 struct thread_info 中的 preempt_count 成员里的 HARDIRQ 域的值。

(2) irq_find_mapping()通过硬件中断号去查找 IRQ 中断号。

(3) irq_exit() 表示硬件中断处理已经完成。与 irq_enter()相反， 通过 preempt_count_sub(), 减少 HARDIRQ 域的值。

```
/**
 * generic_handle_irq - Invoke the handler for a particular irq
 * @irq:  The irq number to handle
 *
 */
int generic_handle_irq(unsigned int irq)
{
    struct irq_desc *desc = irq_to_desc(irq);

    if (! desc)
        return -EINVAL;
    generic_handle_irq_desc(desc);
    return 0;
}
EXPORT_SYMBOL_GPL(generic_handle_irq);
```

```c
/*
 * Architectures call this to let the generic IRQ layer
 * handle an interrupt.
 */
static inline void generic_handle_irq_desc(struct irq_desc *desc)
{
	desc->handle_irq(desc);
}
```

一路看到这里，调用了 irq_desc{}的 handle_irq()回调函数。这个回调函数注册的流程比较长。

在 drivers\irqchip\irq-gic-v3.c，刚刚提到的 gic_init_bases

```c
static int __init gic_init_bases(void __iomem *dist_base,
				 struct redist_region *rdist_regs,
				 u32 nr_redist_regions,
				 u64 redist_stride,
				 struct fwnode_handle *handle)
{
	u32 typer;
	int gic_irqs;
	int err;

	if (!is_hyp_mode_available())
		static_branch_disable(&supports_deactivate_key);

	if (static_branch_likely(&supports_deactivate_key))
		pr_info("GIC: Using split EOI/Deactivate mode\n");

	gic_data.fwnode = handle;
	gic_data.dist_base = dist_base;
	gic_data.redist_regions = rdist_regs;
	gic_data.nr_redist_regions = nr_redist_regions;
	gic_data.redist_stride = redist_stride;

	/*
	 * Find out how many interrupts are supported.
	 * The GIC only supports up to 1020 interrupt sources (SGI+PPI+SPI)
	 */
	typer = readl_relaxed(gic_data.dist_base + GICD_TYPER);
	gic_data.rdists.gicd_typer = typer;
	gic_irqs = GICD_TYPER_IRQS(typer);
	if (gic_irqs > 1020)
		gic_irqs = 1020;
	gic_data.irq_nr = gic_irqs;

	gic_data.domain = irq_domain_create_tree(handle, &gic_irq_domain_ops,
						 &gic_data);
```

irq_domain_create_tree 的时候会注册 gic_irq_domain_ops

```c
static const struct irq_domain_ops gic_irq_domain_ops = {
	.translate = gic_irq_domain_translate,
	.alloc = gic_irq_domain_alloc,
	.free = gic_irq_domain_free,
	.select = gic_irq_domain_select,
};
```

针对每个 irq，都做一次 gic_irq_domain_map

```c
static int gic_irq_domain_alloc(struct irq_domain *domain, unsigned int virq,
				unsigned int nr_irqs, void *arg)
{
	int i, ret;
	irq_hw_number_t hwirq;
	unsigned int type = IRQ_TYPE_NONE;
	struct irq_fwspec *fwspec = arg;

	ret = gic_irq_domain_translate(domain, fwspec, &hwirq, &type);
	if (ret)
		return ret;

	for (i = 0; i < nr_irqs; i++) {
		ret = gic_irq_domain_map(domain, virq + i, hwirq + i);
		if (ret)
			return ret;
	}

	return 0;
} ?end gic_irq_domain_alloc ?
```

可以看到每类 irq，都有对应的 irq_desc{}的 handle_irq()

```c
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_branch_likely(&supports_deactivate_key))
		chip = &gic_eoimode1_chip;

	/* SGIs are private to the core kernel */
	if (hw < 16)
		return -EPERM;
	/* Nothing here */
	if (hw >= gic_data.irq_nr && hw < 8192)
		return -EPERM;
	/* Off limits */
	if (hw >= GIC_ID_NR)
		return -EPERM;

	/* PPIs */
	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);
	}
	/* SPIs */
	if (hw >= 32 && hw < gic_data.irq_nr) {
		irq_domain_set_info(d, irq, hw, chip, d->host_data,
				    handle_fasteoi_irq, NULL, NULL);
		irq_set_probe(irq);
		irqd_set_single_target(irq_desc_get_irq_data(irq_to_desc(irq)));
	}
	/* LPIs */
	if (hw >= 8192 && hw < GIC_ID_NR) {
		if (!gic_dist_supports_lpis())
			return -EPERM;
		irq_domain_set_info(d, irq, hw, chip, d->host_data,
				    handle_fasteoi_irq, NULL, NULL);
	}

	return 0;
} ?end gic_irq_domain_map ?
```

可以看到针对 PPI,SPI,LPI，分别写了 handle_percpu_devid_irq/handle_fasteoi_irq。打开 irq_domain_set_info：

```c
/**
 * irq_domain_set_info - Set the complete data for a @virq in @domain
 * @domain:		Interrupt domain to match
 * @virq:		IRQ number
 * @hwirq:		The hardware interrupt number
 * @chip:		The associated interrupt chip
 * @chip_data:		The associated interrupt chip data
 * @handler:		The interrupt flow handler
 * @handler_data:	The interrupt flow handler data
 * @handler_name:	The interrupt handler name
 */
void irq_domain_set_info(struct irq_domain *domain, unsigned int virq,
			 irq_hw_number_t hwirq, struct irq_chip *chip,
			 void *chip_data, irq_flow_handler_t handler,
			 void *handler_data, const char *handler_name)
{
	irq_domain_set_hwirq_and_chip(domain, virq, hwirq, chip, chip_data);
	__irq_set_handler(virq, handler, 0, handler_name);
	irq_set_handler_data(virq, handler_data);
}
EXPORT_SYMBOL

void
__irq_set_handler(unsigned int irq, irq_flow_handler_t handle, int is_chained,
		  const char *name)
{
	unsigned long flags;
	struct irq_desc *desc = irq_get_desc_buslock(irq, &flags, 0);

	if (!desc)
		return;

	__irq_do_set_handler(desc, handle, is_chained, name);
	irq_put_desc_busunlock(desc, flags);
}
```

```
static void
__irq_do_set_handler(struct irq_desc *desc, irq_flow_handler_t handle,
		     int is_chained, const char *name)
{
	if (! handle) {
		handle = handle_bad_irq;
	} else {
		struct irq_data *irq_data = &desc->irq_data;
#ifdef CONFIG_IRQ_DOMAIN_HIERARCHY
		/*
		 * With hierarchical domains we might run into a
		 * situation where the outermost chip is not yet set
		 * up, but the inner chips are there.  Instead of
		 * bailing we install the handler, but obviously we
		 * cannot enable/startup the interrupt at this point.
		 */
		while (irq_data) {
			if (irq_data->chip != &no_irq_chip)
				break;
			/*
			 * Bail out if the outer chip is not set up
			 * and the interrupt supposed to be started
			 * right away.
			 */
			if (WARN_ON(is_chained))
				return;
			/* Try the parent */
			irq_data = irq_data->parent_data;
		}
#endif
		if (WARN_ON(! irq_data || irq_data->chip == &no_irq_chip))
			return;
	} ?end else ?

	/* Uninstall? */
	if (handle == handle_bad_irq) {
		if (desc->irq_data.chip != &no_irq_chip)
			mask_ack_irq(desc);
		irq_state_set_disabled(desc);
		if (is_chained)
			desc->action = NULL;
		desc->depth = 1;
	}
	desc->handle_irq = handle;
	desc->name = name;
```
啰里啰唆终于理清楚了，就是这里注册的 irq_desc{}的 handle_irq()

总之，根据中断号的范围，会告诉内核，要用以下哪些函数作为 desc->handle_irq()：

- `handle_fasteoi_irq()`
- `handle_simple_irq()`
- `handle_edge_irq()`
- `handle_level_irq()`
- `handle_percpu_irq()`
- `handle_percpu_devid_irq()`

现在接着以 handle_fasteoi_irq 为例打开：

```c
/**
 *    handle_fasteoi_irq - irq handler for transparent controllers
 *    @desc:   the interrupt description structure for this irq
 *
 *    Only a single callback will be issued to the chip: an ->eoi()
 *    call when the interrupt has been serviced. This enables support
 *    for modern forms of interrupt handlers, which handle the flow
 *    details in hardware, transparently.
 */
void handle_fasteoi_irq(struct irq_desc *desc)
{
      struct irq_chip *chip = desc->irq_data.chip;

      raw_spin_lock(&desc->lock);

      if (! irq_may_run(desc))
            goto out;

      desc->istate &= ~(IRQS_REPLAY | IRQS_WAITING);

      /*
       * If its disabled or no action available
       * then mask it and get out of here:
       */
      if (unlikely(! desc->action | | irqd_irq_disabled(&desc->irq_data))) {   ---------------(1)
            desc->istate | = IRQS_PENDING;
            mask_irq(desc);
            goto out;
      }

      kstat_incr_irqs_this_cpu(desc);    ---------------(2)
      if (desc->istate & IRQS_ONESHOT)
            mask_irq(desc);

      preflow_handler(desc);
      handle_irq_event(desc);

      cond_unmask_eoi_irq(desc, chip);    ------------------(4)

      raw_spin_unlock(&desc->lock);
      return;
out:
      if (! (chip->flags & IRQCHIP_EOI_IF_HANDLED))
            chip->irq_eoi(&desc->irq_data);
      raw_spin_unlock(&desc->lock);
} ?end handle_fasteoi_irq ?
```

(1) 如果某个中断没有定义 action 描述符或者该中断被关闭了 IRQD_IRQ_DISABLED， 那么设置该中断状态为 IRQS_PENDING， 并调用 irq_mask()函数屏蔽该中断。

(2)通过 cat /proc/intterrupts 查看中断计数，是在这里进行增加的。

(4) cond_unmask_eoi_irq().写 EOI 寄存器，表示完成了硬中断处理。

```c
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
      irqreturn_t ret;

      desc->istate &= ~IRQS_PENDING;
      irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
      raw_spin_unlock(&desc->lock);

      ret = handle_irq_event_percpu(desc);

      raw_spin_lock(&desc->lock);
      irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
      return ret;
}
```

把pending标志位清除，设置IRQD_IRQ_INPROGRESS标志，表示正在处理中断。

```c
irqreturn_t handle_irq_event_percpu(struct irq_desc *desc)
{
      irqreturn_t retval;
      unsigned int flags = 0;

      retval = __handle_irq_event_percpu(desc, &flags);

      add_interrupt_randomness(desc->irq_data.irq, flags);

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

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

	record_irq_time(desc);

	for_each_action_of_desc(desc, action) {
		irqreturn_t res;

		trace_irq_handler_entry(irq, action);
		res = action->handler(irq, action->dev_id);
		trace_irq_handler_exit(irq, action, res);

		if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pS 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;
		} ?end switch res ?

		retval |= res;
	}

	return retval;
} ?end __handle_irq_event_percpu ?
```

这里注意到三处代码:

1.for_each_action_of_desc 遍历每个 irq_desc 的 action

2.调用 action->handler

3.唤醒 irq 线程部分，也就是 threaded irq 部分。

之前提过 `desc->handle_irq()`

这个其实是 GIC 中断控制器，根据中断号，去实现给你规定号的 irq 处理函数。

而 `desc->action->handler()`

是用户注册实现具体设备的驱动服务程序，都是和 GIC 操作无关的代码。

也就是 request_irq 的时候注册的 handler。

所以现在概念很清晰了，中断处理函数这地方有两个层次:

1.GIC 的通用处理函数 `desc->handle_irq()`

2.用户自定义的处理函数 `desc->action->handler()`

现在打开看下 request_irq 到底是怎么注册 action 的 handler 的:

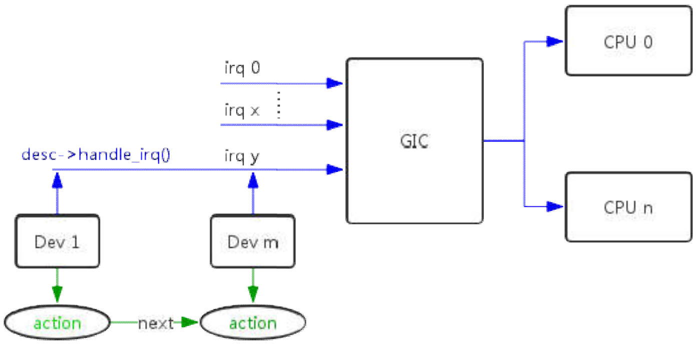
```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
	    const char *name, void *dev)
{
	return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

```c
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)
{
	struct irqaction *action;
	struct irq_desc *desc;
	int retval;

	if (irq == IRQ_NOTCONNECTED)
		return -ENOTCONN;

	/*
	 * Sanity-check: shared interrupts must pass in a real dev-ID,
	 * otherwise we'll have trouble later trying to figure out
	 * which interrupt is which (messes up the interrupt freeing
	 * logic etc).
	 *
	 * Also IRQF_COND_SUSPEND only makes sense for shared interrupts and
	 * it cannot be set along with IRQF_NO_SUSPEND.
	 */
	if (((irqflags & IRQF_SHARED) && !dev_id) ||
	    (!(irqflags & IRQF_SHARED) && (irqflags & IRQF_COND_SUSPEND)) ||
	    ((irqflags & IRQF_NO_SUSPEND) && (irqflags & IRQF_COND_SUSPEND)))
		return -EINVAL;

	desc = irq_to_desc(irq);
	if (!desc)
		return -EINVAL;

	if (!irq_settings_can_request(desc) ||
	    WARN_ON(irq_settings_is_per_cpu_devid(desc)))
		return -EINVAL;

	if (!handler) {
		if (!thread_fn)
			return -EINVAL;
		handler = irq_default_primary_handler;
	}

	action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
	if (!action)
		return -ENOMEM;

	action->handler = handler;
	action->thread_fn = thread_fn;
	action->flags = irqflags;
	action->name = devname;
	action->dev_id = dev_id;


	retval = __setup_irq(irq, desc, action);

	if (retval) {
		irq_chip_pm_put(&desc->irq_data);
		kfree(action->secondary);
		kfree(action);
	}

#ifdef CONFIG_DEBUG_SHIRQ_FIXME
	if (!retval && (irqflags & IRQF_SHARED)) {
		/*
		 * It's a shared IRQ -- the driver ought to be prepared for it
		 * to happen immediately, so let's make sure....
		 * We disable the irq to make sure that a 'real' IRQ doesn't
		 * run in parallel with our fake.
		 */
		unsigned long flags;

		disable_irq(irq);
		local_irq_save(flags);

		handler(irq, dev_id);

		local_irq_restore(flags);
		enable_irq(irq);
	}
#endif
	return retval;
} ?end request_threaded_irq ?
```

可以看到，所有"用户自定义"的 handler，都是放在 action 里面的。包括中断的上半部分 action->handler 和下半部分 action->thread_fn

另外，同时一个中断源可以多个设备共享，所以一个 desc 可以挂载多个 action，由链表

结构组织起来。



这些个 action，通过__setup_irq 安装到 irq_desc 中。
下面打开看下_setup_irq()里面都完成了哪些工作。这个函数流程很长：

```
static int
__setup_irq(unsigned int irq, struct irq_desc *desc, struct irqaction *new)
{
		struct irqaction *old, **old_ptr;
		unsigned long flags, thread_mask = 0;
		int ret, nested, shared = 0;
		cpumask_var_t mask;

		if (!desc)
				return -EINVAL;

		if (desc->irq_data.chip == &no_irq_chip)
				return -ENOSYS;
		if (!try_module_get(desc->owner))
				return -ENODEV;

		/*
		 * Check whether the interrupt nests into another interrupt
		 * thread.
		 */
		nested = irq_settings_is_nested_thread(desc);
		// (4.1) 判断中断是否是支持嵌套
		if (nested) {
				if (!new->thread_fn) {
						ret = -EINVAL;
						goto out_mput;
				}
				/*
				 * Replace the primary handler which was provided from
				 * the driver for non nested interrupt handling by the
```

```
         * dummy function which warns when called.
         */
        new->handler = irq_nested_primary_handler;
    } else {
        // (4.2) 判断中断是否可以被线程化
        // 如果中断没有设置 _IRQ_NOTHREAD 标志 & 强制中断线程化标志被设置 (force_irqthreads=1)
        // 强制把中断线程化：
        // new->thread_fn = new->handler;new->handler = irq_default_primary_handler;
        if (irq_settings_can_thread(desc))
            irq_setup_forced_threading(new);
    }

    /*
     * Create a handler thread when a thread function is supplied
     * and the interrupt does not nest into another interrupt
     * thread.
     */
    // (4.3) 如果是线程化中断，创建线程化中断对应的线程
    if (new->thread_fn && !nested) {
        struct task_struct *t;
        static const struct sched_param param = {
            .sched_priority = MAX_USER_RT_PRIO/2,
        };

        // 创建线程
        t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
                   new->name);
        if (IS_ERR(t)) {
            ret = PTR_ERR(t);
            goto out_mput;
        }
        // 设置调度方式是 SCHED_FIFO
        sched_setscheduler_nocheck(t, SCHED_FIFO, &param);

        /*
         * We keep the reference to the task struct even if
         * the thread dies to avoid that the interrupt code
         * references an already freed task_struct.
         */
        get_task_struct(t);
        // 赋值给 ->thread 成员
        new->thread = t;
        /*
```

```
                 * Tell the thread to set its affinity. This is
                 * important for shared interrupt handlers as we do
                 * not invoke setup_affinity() for the secondary
                 * handlers as everything is already set up. Even for
                 * interrupts marked with IRQF_NO_BALANCE this is
                 * correct as we want the thread to move to the cpu(s)
                 * on which the requesting code placed the interrupt.
                 */
                set_bit(IRQTF_AFFINITY, &new->thread_flags);
}

if (!alloc_cpumask_var(&mask, GFP_KERNEL)) {
                ret = -ENOMEM;
                goto out_thread;
}

/*
 * Drivers are often written to work w/o knowledge about the
 * underlying irq chip implementation, so a request for a
 * threaded irq without a primary hard irq context handler
 * requires the ONESHOT flag to be set. Some irq chips like
 * MSI based interrupts are per se one shot safe. Check the
 * chip flags, so we can avoid the unmask dance at the end of
 * the threaded handler for those.
 */
if (desc->irq_data.chip->flags & IRQCHIP_ONESHOT_SAFE)
                new->flags &= ~IRQF_ONESHOT;

/*
 * The following block of code has to be executed atomically
 */
// (4.4) 找到最后一个 action 结构
raw_spin_lock_irqsave(&desc->lock, flags);
old_ptr = &desc->action;
old = *old_ptr;
if (old) {
                /*
                 * Can't share interrupts unless both agree to and are
                 * the same type (level, edge, polarity). So both flag
                 * fields must have IRQF_SHARED set and the bits which
                 * set the trigger type must match. Also all must
                 * agree on ONESHOT.
                 */
                if (!((old->flags & new->flags) & IRQF_SHARED) ||
```

```
            ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK) ||
            ((old->flags ^ new->flags) & IRQF_ONESHOT))
                goto mismatch;

        /* All handlers must agree on per-cpuness */
        if ((old->flags & IRQF_PERCPU) !=
            (new->flags & IRQF_PERCPU))
                goto mismatch;

        /* add new interrupt at end of irq queue */
        do {
                /*
                 * Or all existing action->thread_mask bits,
                 * so we can find the next zero bit for this
                 * new action.
                 */
                thread_mask |= old->thread_mask;
                old_ptr = &old->next;
                old = *old_ptr;
        } while (old);
        // 如果有多个 action，共享标志设为 1
        shared = 1;
}

/*
 * Setup the thread mask for this irqaction for ONESHOT. For
 * !ONESHOT irqs the thread mask is 0 so we can avoid a
 * conditional in irq_wake_thread().
 */
if (new->flags & IRQF_ONESHOT) {
        /*
         * Unlikely to have 32 resp 64 irqs sharing one line,
         * but who knows.
         */
        if (thread_mask == ~0UL) {
                ret = -EBUSY;
                goto out_mask;
        }
        /*
         * The thread_mask for the action is or'ed to
         * desc->thread_active to indicate that the
         * IRQF_ONESHOT thread handler has been woken, but not
         * yet finished. The bit is cleared when a thread
         * completes. When all threads of a shared interrupt
```

```
         * line have completed desc->threads_active becomes
         * zero and the interrupt line is unmasked. See
         * handle.c:irq_wake_thread() for further information.
         *
         * If no thread is woken by primary (hard irq context)
         * interrupt handlers, then desc->threads_active is
         * also checked for zero to unmask the irq line in the
         * affected hard irq flow handlers
         * (handle_[fasteoi|level]_irq).
         *
         * The new action gets the first zero bit of
         * thread_mask assigned. See the loop above which or's
         * all existing action->thread_mask bits.
         */
        new->thread_mask = 1 << ffz(thread_mask);

} else if (new->handler == irq_default_primary_handler &&
           !(desc->irq_data.chip->flags & IRQCHIP_ONESHOT_SAFE)) {
        /*
         * The interrupt was requested with handler = NULL, so
         * we use the default primary handler for it. But it
         * does not have the oneshot flag set. In combination
         * with level interrupts this is deadly, because the
         * default primary handler just wakes the thread, then
         * the irq lines is reenabled, but the device still
         * has the level irq asserted. Rinse and repeat....
         *
         * While this works for edge type interrupts, we play
         * it safe and reject unconditionally because we can't
         * say for sure which type this interrupt really
         * has. The type flags are unreliable as the
         * underlying chip implementation can override them.
         */
        pr_err("Threaded irq requested with handler=NULL and !ONESHOT for irq %d\n",
               irq);
        ret = -EINVAL;
        goto out_mask;
}

// (4.5)  如果是第一个  action，做一些初始化工作
if (!shared) {
        ret = irq_request_resources(desc);
        if (ret) {
                pr_err("Failed to request resources for %s (irq %d) on irqchip %s\n",
```

```
                        new->name, irq, desc->irq_data.chip->name);
                goto out_mask;
        }

        init_waitqueue_head(&desc->wait_for_threads);

        /* Setup the type (level, edge polarity) if configured: */
        if (new->flags & IRQF_TRIGGER_MASK) {
                ret = __irq_set_trigger(desc, irq,
                                new->flags & IRQF_TRIGGER_MASK);

                if (ret)
                        goto out_mask;
        }

        desc->istate &= ~(IRQS_AUTODETECT | IRQS_SPURIOUS_DISABLED | \
                          IRQS_ONESHOT | IRQS_WAITING);
        irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);

        if (new->flags & IRQF_PERCPU) {
                irqd_set(&desc->irq_data, IRQD_PER_CPU);
                irq_settings_set_per_cpu(desc);
        }

        if (new->flags & IRQF_ONESHOT)
                desc->istate |= IRQS_ONESHOT;

        if (irq_settings_can_autoenable(desc))
                irq_startup(desc, true);
        else
                /* Undo nested disables: */
                desc->depth = 1;

        /* Exclude IRQ from balancing if requested */
        if (new->flags & IRQF_NOBALANCING) {
                irq_settings_set_no_balancing(desc);
                irqd_set(&desc->irq_data, IRQD_NO_BALANCING);
        }

        // 设置中断亲和力
        /* Set default affinity mask once everything is setup */
        setup_affinity(irq, desc, mask);
} else if (new->flags & IRQF_TRIGGER_MASK) {
        unsigned int nmsk = new->flags & IRQF_TRIGGER_MASK;
```

```c
		unsigned int omsk = irq_settings_get_trigger_mask(desc);

		if (nmsk != omsk)
			/* hope the handler works with current  trigger mode */
			pr_warning("irq %d uses trigger mode %u; requested %u\n",
				   irq, nmsk, omsk);
	}

	// (4.6) 将新的 action 插入到 desc 链表中
	new->irq = irq;
	*old_ptr = new;

	irq_pm_install_action(desc, new);

	/* Reset broken irq detection when installing new handler */
	desc->irq_count = 0;
	desc->irqs_unhandled = 0;

	/*
	 * Check whether we disabled the irq via the spurious handler
	 * before. Reenable it and give it another chance.
	 */
	// (4.7) 如果中断之前被虚假 disable 了，重新 enable 中断
	if (shared && (desc->istate & IRQS_SPURIOUS_DISABLED)) {
		desc->istate &= ~IRQS_SPURIOUS_DISABLED;
		__enable_irq(desc, irq);
	}

	raw_spin_unlock_irqrestore(&desc->lock, flags);

	/*
	 * Strictly no need to wake it up, but hung_task complains
	 * when no hard interrupt wakes the thread up.
	 */
	// (4.8) 唤醒线程化中断对应的线程
	if (new->thread)
		wake_up_process(new->thread);

	register_irq_proc(irq, desc);
	new->dir = NULL;
	register_handler_proc(irq, new);
	free_cpumask_var(mask);

	return 0;
```

这个过程中，创建了 irq thread。这里打开看下细节：

```
// 创建线程
t = kthread_create(irq_thread, new, "irq/%d-%s", irq, new->name);
```

1.不管是哪个中断的线程，其入口函数是统一的，都是 static int irq_thread(void *data)
2.线程的命名规则是: "irq/%d-%s"，也就是 irq/中断号-中断名。
3.线程的调度设置成了 SCHED_FIFO，实时的。

```
root@:/ # ps | grep "irq/"
root      171   2     0     0     irq_thread 0000000000 S irq/389-charger
root      239   2     0     0     irq_thread 0000000000 S irq/296-PS_int-
root      247   2     0     0     irq_thread 0000000000 S irq/297-1124000
root      1415  2     0     0     irq_thread 0000000000 S irq/293-goodix_
root@a0255:/ #
```

之前说了，在

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

	record_irq_time(desc);

	for_each_action_of_desc(desc, action) {
		irqreturn_t res;

		trace_irq_handler_entry(irq, action);
		res = action->handler(irq, action->dev_id);
		trace_irq_handler_exit(irq, action, res);

		if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pS 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;
		} ?end switch res ?

		retval |= res;
	}

	return retval;
} ?end __handle_irq_event_percpu ?
```

这里会唤醒线程 irq_thread：

中断对应的线程："irq/%d-%s"
kernel/irq/manage.c:

```
static int irq_thread(void *data)
{
	struct callback_head on_exit_work;
	struct irqaction *action = data;
	struct irq_desc *desc = irq_to_desc(action->irq);
	irqreturn_t (*handler_fn)(struct irq_desc *desc,
			struct irqaction *action);

	if (force_irqthreads &&
test_bit(IRQTF_FORCED_THREAD,
					&action->thread_flags))
		handler_fn = irq_forced_thread_fn;
	else
		handler_fn = irq_thread_fn;

	init_task_work(&on_exit_work, irq_thread_dtor);
	task_work_add(current, &on_exit_work, false);

	irq_thread_check_affinity(desc, action);

	//(2.1)等待唤醒和IRQTF_RUNTHREAD置位
	//__irq_wake_thread能唤醒
	while (!irq_wait_for_interrupt(action)) {
		irqreturn_t action_ret;

		irq_thread_check_affinity(desc, action);

		//(2.2)处理具体的action
		action_ret = handler_fn(desc, action);
		if (action_ret == IRQ_HANDLED)
			atomic_inc(&desc->threads_handled);

		//(2.3)唤醒需要和本线程同步的其他线程
		wake_threads_waitq(desc);
	}


	task_work_cancel(current, irq_thread_dtor);
	return 0;
}
```

kernel/irq/manage.c:

```
static irqreturn_t irq_thread_fn(struct irq_desc *desc,
		struct irqaction *action)
{
	irqreturn_t ret;

	ret = action->thread_fn(action->irq, action->dev_id);
	irq_finalize_oneshot(desc, action);
	return ret;
}
```

这里就调用到了 thread_fn，这个是每个 action 中用户自定义的函数。

参考文档：

《Linux Interrupt - 魅族内核团队》
《linux kernel 的中断子系统之（七）》

《linux kernel 的中断子系统之（八）》
《linux kernel 的中断子系统之（九）》
《Fundamentals of ARMv8-A》
Linux 5.2.5 内核代码