要操作单个中断源的 enable/disable，可以使用 enable_irq(irq)/disable_irq(irq) 函数。

在 kernel\irq\manage.c

```
/**
 *    enable_irq - enable handling of an irq
 *    @irq: Interrupt to enable
 *
 *    Undoes the effect of one call to disable_irq().  If this
 *    matches the last disable, processing of interrupts on this
 *    IRQ line is re-enabled.
 *
 *    This function may be called from IRQ context only when
 *    desc->irq_data.chip->bus_lock and desc->chip->bus_sync_unlock are NULL !
 */
void enable_irq(unsigned int irq)
{
    unsigned long flags;
    struct irq_desc *desc = irq_get_desc_buslock(irq, &flags, IRQ_GET_DESC_CHECK_GLOBAL);

    if (! desc)
        return;
    if (WARN(! desc->irq_data.chip,
         KERN_ERR "enable_irq before setup/request_irq: irq %u\n", irq))
        goto out;

    __enable_irq(desc);
out:
    irq_put_desc_busunlock(desc, flags);
}
EXPORT_SYMBOL_GPL(enable_irq);

void __enable_irq(struct irq_desc *desc)
{
    switch (desc->depth) {
    case 0:
 err_out:
        WARN(1, KERN_WARNING "Unbalanced enable for IRQ %d\n",
             irq_desc_get_irq(desc));
        break;
    case 1: {
        if (desc->istate & IRQS_SUSPENDED)
            goto err_out;
        /* Prevent probing on this irq: */
        irq_settings_set_noprobe(desc);
        /*
         * Call irq_startup() not irq_enable() here because the
         * interrupt might be marked NOAUTOEN. So irq_startup()
         * needs to be invoked when it gets enabled the first
         * time. If it was already started up, then irq_startup()
         * will invoke irq_enable() under the hood.
         */
        irq_startup(desc, IRQ_RESEND, IRQ_START_FORCE);
        break;
    }
    default:
        desc->depth--;
    }
}
```

在 kernel\irq\chip.c

```
int irq_startup(struct irq_desc *desc, bool resend, bool force)
{
    struct irq_data *d = irq_desc_get_irq_data(desc);
    struct cpumask *aff = irq_data_get_affinity_mask(d);
    int ret = 0;

    desc->depth = 0;

    if (irqd_is_started(d)) {
        irq_enable(desc);
    } else {
        switch (__irq_startup_managed(desc, aff, force)) {
        case IRQ_STARTUP_NORMAL:
            ret = __irq_startup(desc);
            irq_setup_affinity(desc);
            break;
        case IRQ_STARTUP_MANAGED:
            irq_do_set_affinity(d, aff, false);
            ret = __irq_startup(desc);
            break;
        case IRQ_STARTUP_ABORT:
            irqd_set_managed_shutdown(d);
            return 0;
        }
    }
    if (resend)
        check_irq_resend(desc);

    return ret;
}
```

```
void irq_enable(struct irq_desc *desc)
{
	if (!irqd_irq_disabled(&desc->irq_data)) {
		unmask_irq(desc);
	} else {
		irq_state_clr_disabled(desc);
		if (desc->irq_data.chip->irq_enable) {
			desc->irq_data.chip->irq_enable(&desc->irq_data);
			irq_state_clr_masked(desc);
		} else {
			unmask_irq(desc);
		}
	}
}
```

此处会调用 irq_chip->irq_enable()回调函数。（如果有注册过 irq_enable）

这个函数可以由中断控制器注册的时候指定，例如：

```
static struct irq_chip atari_irq_chip = {
	.name		= "atari",
	.irq_startup	= atari_irq_startup,
	.irq_shutdown	= atari_irq_shutdown,
	.irq_enable	= atari_irq_enable,
	.irq_disable	= atari_irq_disable,
};
```

或者调用（如果没有注册过 irq_enable）

```
void unmask_irq(struct irq_desc *desc)
{
	if (!irqd_irq_masked(&desc->irq_data))
		return;

	if (desc->irq_data.chip->irq_unmask) {
		desc->irq_data.chip->irq_unmask(&desc->irq_data);
		irq_state_clr_masked(desc);
	}
}
```

还是以 GIC v3 为例:

```
static struct irq_chip gic_chip = {
	.name			= "GICv3",
	.irq_mask		= gic_mask_irq,
	.irq_unmask		= gic_unmask_irq,
	.irq_eoi		= gic_eoi_irq,
	.irq_set_type		= gic_set_type,
	.irq_set_affinity	= gic_set_affinity,
	.irq_get_irqchip_state	= gic_irq_get_irqchip_state,
	.irq_set_irqchip_state	= gic_irq_set_irqchip_state,
	.irq_nmi_setup		= gic_irq_nmi_setup,
	.irq_nmi_teardown	= gic_irq_nmi_teardown,
	.flags			= IRQCHIP_SET_TYPE_MASKED |
				  IRQCHIP_SKIP_SET_WAKE |
				  IRQCHIP_MASK_ON_SUSPEND,
};

static void gic_unmask_irq(struct irq_data *d)
{
	gic_poke_irq(d, GICD_ISENABLER);
}
```

到这个 gic_poke_irq，就是具体操作中断控制器了。

所以总结下就是 enable_irq 其核心，就是操作中断控制器的中断使能。

再来看 disable_irq()

```
/**
 *	disable_irq - disable an irq and wait for completion
 *	@irq: Interrupt to disable
 *
 *	Disable the selected interrupt line.  Enables and Disables are
 *	nested.
 *	This function waits for any pending IRQ handlers for this interrupt
 *	to complete before returning. If you use this function while
 *	holding a resource the IRQ handler may need you will deadlock.
 *
 *	This function may be called - with care - from IRQ context.
 */
void disable_irq(unsigned int irq)
{
	if (!__disable_irq_nosync(irq))
		synchronize_irq(irq);
}
```

此处注意这个函数是会等待 irq handler 完成以后，再退出。包括线程化中断的线程部分。

```
/**
 *    synchronize_irq - wait for pending IRQ handlers (on other CPUs)
 *    @irq: interrupt number to wait for
 *
 *    This function waits for any pending IRQ handlers for this interrupt
 *    to complete before returning. If you use this function while
 *    holding a resource the IRQ handler may need you will deadlock.
 *
 *    Can only be called from preemptible code as it might sleep when
 *    an interrupt thread is associated to @irq.
 *
 *    It optionally makes sure (when the irq chip supports that method)
 *    that the interrupt is not pending in any CPU and waiting for
 *    service.
 */
void synchronize_irq(unsigned int irq)
{
    struct irq_desc *desc = irq_to_desc(irq);

    if (desc) {
        __synchronize_hardirq(desc, true);
        /*
         * We made sure that no hardirq handler is
         * running. Now verify that no threaded handlers are
         * active.
         */
        wait_event(desc->wait_for_threads,
               !atomic_read(&desc->threads_active));
    }
}
```

wait_event 就是睡眠并等待其入参条件为真。