

## 快速链接:

👉 👉 👉 [ARMv8/ARMv9架构入门到精通-\[目录\]](#) 👉 👉 👉

- 付费专栏-付费课程 **【购买须知】** :
- 联系方式-加入交流群 ---**联系方式-加入交流群**
- 个人博客笔记导读目录(全部)

**引流关键词:** armv8, armv9, gic, gicv2, gicv3, 异常, 中断, irq, fiq, serror, sync, 同步异常, 异步异常, 向量表, 向量表基地址, VBAR, vbar\_el3, 中断嵌套, 中断级联, Linux Kernel, optee, ATF, TF-A, optee, hypervisor, SPM

## 目录

- 1 Linux Kernel arm64中断向量表的定义
- 2 Linux Kernel arm64设置中断向量表的基地址
- 3 kernel\_ventry宏的介绍
- 4 未实现的异常向量: elx\_yyy\_invalid
- 5 el1\_irq的介绍 - 跳转到注册的handler函数
- 6 handle\_domain\_irq
- 7 关于中断级联的介绍

### 1 Linux Kernel arm64中断向量表的定义

```
1 (linux/arch/arm64/kernel/entry.S)
2
3 /*
4  * Exception vectors.
5  */
6     .pushsection ".entry.text", "ax"
7
8     .align 11
9     SYM_CODE_START(vectors)
10     kernel_ventry 1, sync_invalid      // Synchronous EL1t
11     kernel_ventry 1, irq_invalid      // IRQ EL1t
12     kernel_ventry 1, fiq_invalid      // FIQ EL1t
13     kernel_ventry 1, error_invalid    // Error EL1t
```

```

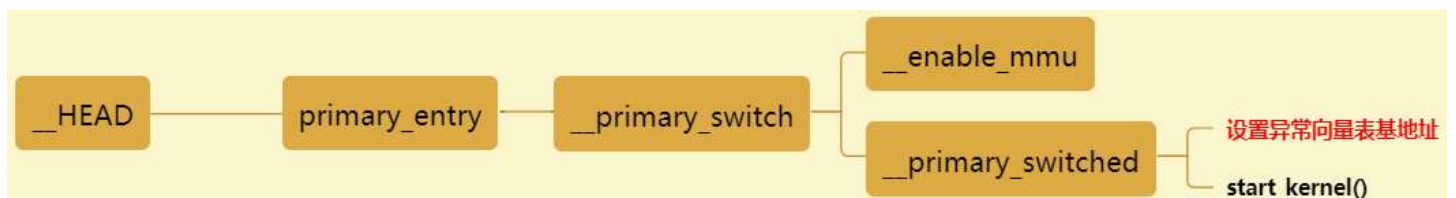
13
14     kernel_ventry    1, sync           // Synchronous EL1h
15     kernel_ventry    1, irq           // IRQ EL1h
16     kernel_ventry    1, fiq           // FIQ EL1h
17     kernel_ventry    1, error          // Error EL1h
18
19     kernel_ventry    0, sync           // Synchronous 64-bit EL0
20     kernel_ventry    0, irq           // IRQ 64-bit EL0
21     kernel_ventry    0, fiq           // FIQ 64-bit EL0
22     kernel_ventry    0, error          // Error 64-bit EL0
23
24 #ifdef CONFIG_COMPAT
25     kernel_ventry    0, sync_compat, 32 // Synchronous 32-bit EL0
26     kernel_ventry    0, irq_compat, 32  // IRQ 32-bit EL0
27     kernel_ventry    0, fiq_compat, 32  // FIQ 32-bit EL0
28     kernel_ventry    0, error_compat, 32 // Error 32-bit EL0
29 #else
30     kernel_ventry    0, sync_invalid, 32 // Synchronous 32-bit EL0
31     kernel_ventry    0, irq_invalid, 32  // IRQ 32-bit EL0
32     kernel_ventry    0, fiq_invalid, 32  // FIQ 32-bit EL0
33     kernel_ventry    0, error_invalid, 32 // Error 32-bit EL0
34 #endif
35 SYM_CODE_END(vectors)

```

## 思考:

- 1、这里有没有按照armv8定义的异常向量表排列？不是每一个offset只有128bytes地址空间吗，如何做到的？
- 2、Linux Kernel arm64体系中不是没有实现FIQ吗，这里为何实现了？
- 3、第一组异常向量为何没有实现？

## 2 Linux Kernel arm64设置中断向量表的基地址



```

1 (linux/arch/arm64/kernel/head.S)
2
3 SYM_FUNC_START_LOCAL(__primary_switched)
4     adrp    x4, init_thread_union
5     add sp, x4, #THREAD_SIZE
6     adr_l   x5, init_task
7     msr sp_el0, x5           // Save thread_info
8
9

```

```

9      adr_l    x8, vectors          // load VBAR_EL1 with virtual
10      msr vbar_el1, x8            // vector table address
11      isb
12
13      .....
14      b      start_kernel
15  SYM_FUNC_END(__primary_switched)

```

## 思考:

1、设置VBAR\_EL1，如果系统系统里有8个ARM Core，那么8个Core都需要设置吗，分别如何设置的？

## 3 kernel\_ventry宏的介绍

```

1  (linux/arch/arm64/kernel/entry.S)
2
3      .macro kernel_ventry, el, label, regsize = 64
4      .align 7
5  #ifdef CONFIG_UNMAP_KERNEL_AT_EL0
6      .if \el == 0
7  alternative_if ARM64_UNMAP_KERNEL_AT_EL0
8      .if \regsize == 64
9      mrs x30, tpidrro_el0
10     msr tpidrro_el0, xzr
11     .else
12     mov x30, xzr
13     .endif
14 alternative_else_nop_endif
15     .endif
16 #endif
17
18     sub sp, sp, #PT_REGS_SIZE
19 #ifdef CONFIG_VMAP_STACK
20     /*
21      * Test whether the SP has overflowed, without corrupting a GPR.
22      * Task and IRQ stacks are aligned so that SP & (1 << THREAD_SHIFT)
23      * should always be zero.
24      */
25     add sp, sp, x0          // sp' = sp + x0
26     sub x0, sp, x0          // x0' = sp' - x0 = (sp + x0) - x0 = sp
27     tbnz    x0, #THREAD_SHIFT, 0f
28     sub x0, sp, x0          // x0'' = sp' - x0' = (sp + x0) - sp = x0
29     sub sp, sp, x0          // sp'' = sp' - x0 = (sp + x0) - x0 = sp
30     b     el\()\el\()\_label
31

```

```

32  0:
33      /*
34       * Either we've just detected an overflow, or we've taken an exception
35       * while on the overflow stack. Either way, we won't return to
36       * userspace, and can clobber EL0 registers to free up GPRs.
37       */
38
39      /* Stash the original SP (minus PT_REGS_SIZE) in tpidr_el0. */
40      msr tpidr_el0, x0
41
42      /* Recover the original x0 value and stash it in tpidrro_el0 */
43      sub x0, sp, x0
44      msr tpidrro_el0, x0
45
46      /* Switch to the overflow stack */
47      adr_this_cpu sp, overflow_stack + OVERFLOW_STACK_SIZE, x0
48
49      /*
50       * Check whether we were already on the overflow stack. This may happen
51       * after panic() re-enables interrupts.
52       */
53      mrs x0, tpidr_el0          // sp of interrupted context
54      sub x0, sp, x0            // delta with top of overflow stack
55      tst x0, #~(OVERFLOW_STACK_SIZE - 1) // within range?
56      b.ne    __bad_stack       // no? -> bad stack pointer
57
58      /* We were already on the overflow stack. Restore sp/x0 and carry on. */
59      sub sp, sp, x0
60      mrs x0, tpidrro_el0
61  #endif
62      b    el\()\el\()\_label
63  .endm

```

注意.align=7, 说明该段代码是以 $2^7=128$ 字节对其的, 这和向量表中每一个offset的大小是一致的  
 代码看似非常复杂, 其实最终跳转到了 `b el\()\el\()\_label`, 翻译一下, 其实就是跳转到了如下这样的函数中

```

1  el1_sync_invalid
2  el1_irq_invalid
3  el1_fiq_invalid
4  el1_error_invalid
5
6  el1_sync
7  el1_irq
8  el1_fiq
9  el1_error
10

```

```

11
12 el0_sync
13 el0_irq
14 el0_fiq
   el0_error

```

#### 4 未实现的异常向量: elx\_yyy\_invalid

未实现的向量定义为了elx\_yyy\_invalid函数, 该invalid函数其实也是一种实现, 它最终调用了panic函数  
例如el1\_irq\_invalid的Flow : el1\_irq\_invalid --> b1 bad\_mode --> panic("bad mode")

```

1  SYM_CODE_START_LOCAL(el1_irq_invalid)
2      inv_entry 1, BAD_IRQ
3  SYM_CODE_END(el1_irq_invalid)
4
5
6  /*
7   * Bad Abort numbers
8   *-----
9   */
10 #define BAD_SYNC      0
11 #define BAD_IRQ       1
12 #define BAD_FIQ       2
13 #define BAD_ERROR     3
14
15 /*
16  * Invalid mode handlers
17  */
18 .macro inv_entry, el, reason, regsize = 64
19     kernel_entry \el, \regsize
20     mov x0, sp
21     mov x1, #\reason
22     mrs x2, esr_el1
23     b1 bad_mode
24     ASM_BUG()
25 .endm
26
27
28 /*
29  * bad_mode handles the impossible case in the exception vector. This is always
30  * fatal.
31  */
32 asmlinkage void notrace bad_mode(struct pt_regs *regs, int reason, unsigned int esr)
33 {
34     arm64_enter_nmi(regs);
35
36     console_verbose();
37
38     pr_crit("Bad mode in %s handler detected on CPU%d, code 0x%08x -- %s\n",
39             handler[reason], smp_processor_id(), esr,

```

```

13     esr_get_class_string(esr));
14
15     __show_regs(regs);
16     local_daif_mask();
17     panic("bad mode");
18 }

```

## 5 el1\_irq的介绍 - 跳转到注册的handler函数

抛开事务看本质，`el1_interrupt_handler handle_arch_irq` 其实就是调用`handle_arch_irq`，而`handle_arch_irq`指向`irq-gic-v3.c`中定义的handler函数

```

1     .align    6
2     SYM_CODE_START_LOCAL_NOALIGN(el1_irq)
3     kernel_entry 1
4     el1_interrupt_handler handle_arch_irq
5     kernel_exit 1
6     SYM_CODE_END(el1_irq)

```

这里我们就不再深究`kernel_entry`和`kernel_exit`，它俩里面干得事情非常多。当前我们需要了解，一个是保存general purpose寄存器，一个是恢复就可以了。

```

1     .macro    kernel_entry, el, regsize = 64
2     .if \regsize == 32
3     mov w0, w0          // zero upper 32 bits of x0
4     .endif
5     stp x0, x1, [sp, #16 * 0]
6     stp x2, x3, [sp, #16 * 1]
7     stp x4, x5, [sp, #16 * 2]
8     stp x6, x7, [sp, #16 * 3]
9     stp x8, x9, [sp, #16 * 4]
10    stp x10, x11, [sp, #16 * 5]
11    stp x12, x13, [sp, #16 * 6]
12    stp x14, x15, [sp, #16 * 7]
13    stp x16, x17, [sp, #16 * 8]
14    stp x18, x19, [sp, #16 * 9]
15    stp x20, x21, [sp, #16 * 10]
16    stp x22, x23, [sp, #16 * 11]
17    stp x24, x25, [sp, #16 * 12]
18    stp x26, x27, [sp, #16 * 13]
19    stp x28, x29, [sp, #16 * 14]
20    .....

1     .macro    kernel_exit, el
2     .....
3     msr elr_el1, x21      // set up the return data

```

```
4  msr spsr_el1, x22
5  ldp x0, x1, [sp, #16 * 0]
6  ldp x2, x3, [sp, #16 * 1]
7  ldp x4, x5, [sp, #16 * 2]
8  ldp x6, x7, [sp, #16 * 3]
9  ldp x8, x9, [sp, #16 * 4]
10 ldp x10, x11, [sp, #16 * 5]
11 ldp x12, x13, [sp, #16 * 6]
12 ldp x14, x15, [sp, #16 * 7]
13 ldp x16, x17, [sp, #16 * 8]
14 ldp x18, x19, [sp, #16 * 9]
15 ldp x20, x21, [sp, #16 * 10]
16 ldp x22, x23, [sp, #16 * 11]
17 ldp x24, x25, [sp, #16 * 12]
18 ldp x26, x27, [sp, #16 * 13]
19 ldp x28, x29, [sp, #16 * 14]
20 ldr lr, [sp, #S_LR]
21 add sp, sp, #PT_REGS_SIZE    // restore sp
22 .....
```

xref: /linux/arch/arm64/kernel/entry.S

```
715 SYM_CODE_START_LOCAL_NOALIGN(el1_irq)
716     kernel_entry 1
717     el1_interrupt_handler handle_arch_irq
718     kernel_exit 1
719 SYM_CODE_END(el1_irq)
```

xref: /linux/arch/arm64/kernel/entry.S

```
555     .macro el1_interrupt_handler, handler:req
556     enable_da
557
558     mov     x0, sp
559     bl      enter_el1_irq_or_nmi
560
561     irq_handler    \handler
```

xref: /linux/arch/arm64/kernel/entry.S

```
538     .macro irq_handler, handler:req
539     ldr_l    x1, \handler
540     mov     x0, sp
541     irq_stack_entry
542     blr     x1
543     irq_stack_exit
544     .endm
```

其实就是跳转到  
handle\_arch\_irq  
CSDN @代码改变世界ctw

xref: /linux/drivers/irqchip/irq-gic-v3.c

```
1668 static int __init gic_init_bases(void __iomem *dist_base,
1669     struct redistrib_region *rdist_regs,
1670     u32 nr_redist_regions,
1671     u64 redistrib_stride,
1672     struct fwnode_handle *handle)
1673 {
1733     set_handle_irq(gic_handle_irq);
```

xref: /linux/arch/arm64/kernel/irq.c

```
87 int __init set_handle_irq(void (*handle_irq)(struct pt_regs *))
88 {
89     if (handle_arch_irq != default_handle_irq)
90         return -EBUSY;
91
92     handle_arch_irq = handle_irq;
93     pr_info("Root IRQ handler: %ps\n", handle_irq);
94     return 0;
95 }
```

其实就是将gic的中定义的中断处理函数，  
保存到了handle\_arch\_irq全局变量中

xref: /linux/drivers/irqchip/irq-gic-v3.c

```
679 static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
680 {
681     u32 irqnr;
682
683     irqnr = do_read_iar(regs);
684
685     /* Check for special IDs first */
686     if ((irqnr >= 1020 && irqnr <= 1023))
687         return;
688
689     if (gic_supports_nmi() &&
690         unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
691         gic_handle_nmi(irqnr, regs);
692         return;
693     }
694
695     if (gic_prio_masking_enabled()) {
696         gic_pmr_mask_irqs();
697         gic_arch_enable_irqs();
698     }
699
700     if (static_branch_likely(&supports_deactivate_key))
701         gic_write_eoir(irqnr);
702     else
703         isb();
704
705     if (handle_domain_irq(gic_data.domain, irqnr, regs)) {
706         WARN_ONCE(true, "Unexpected interrupt received!\n");
707         gic_deactivate_unhandled(irqnr);
708     }
709 }
```

CSDN @代码改变世界ctw

我们再来剖析gic\_handle\_irq()函数，其实就是涉及gic的读写了，从gic中读取硬件中断号，然后调用handle\_domain\_irq函数，找到相匹配的中断handler函数，然后回调。

```
1 | (linux/drivers/irqchip/irq-gic-v3.c)
2 |
3 | static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
4 | {
5 |     u32 irqnr;
6 |
7 |     irqnr = do_read_iar(regs);
8 |
9 |     /* Check for special IDs first */
10 |    if ((irqnr >= 1020 && irqnr <= 1023))
11 |        return;
```



```

12
13     if (gic_supports_nmi() &&
14         unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
15         gic_handle_nmi(irqnr, regs);
16         return;
17     }
18
19     if (gic_prio_masking_enabled()) {
20         gic_pmr_mask_irqs();
21         gic_arch_enable_irqs();
22     }
23
24     if (static_branch_likely(&supports_deactivate_key))
25         gic_write_eoir(irqnr);
26     else
27         isb();
28
29     if (handle_domain_irq(gic_data.domain, irqnr, regs)) {
30         WARN_ONCE(true, "Unexpected interrupt received!\n");
31         gic_deactivate_unhandled(irqnr);
32     }
33 }

```

xref: /linux/drivers/irqchip/irq-gic-v3.c

```

679 static asminkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
680 {
681     u32 irqnr;
682
683     irqnr = do_read_iar(regs); 读gicd寄存器, 读取中断号
684
685     /* Check for special IDs first */
686     if ((irqnr >= 1020 && irqnr <= 1023)) 这4个中断号是在ATF中使用的, 有
687         return; 这特殊作用, 这里先不介绍
688
689     if (gic_supports_nmi() &&
690         unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
691         gic_handle_nmi(irqnr, regs);
692         return;
693     }
694
695     if (gic_prio_masking_enabled()) { 进入中断后, 异常默认都
696         gic_pmr_mask_irqs();
697         gic_arch_enable_irqs(); MASK了, 这里再打开下
698     }
699
700     if (static_branch_likely(&supports_deactivate_key))
701         gic_write_eoir(irqnr); 写gicd寄存器, 中断结束标志, 置为inactive
702     else
703         isb();
704
705     if (handle_domain_irq(gic_data.domain, irqnr, regs)) { 进入您注册的handler程序中去 处理中断
706         WARN_ONCE(true, "Unexpected interrupt received!\n");
707         gic_deactivate_unhandled(irqnr);
708     }
709 }

```

CSDN @代码改变世界ctw

另外注意一点, 在Linux Kernel 5.0之后, gic中的handler处理函数, 发生了一些细微的变化, 如下所

示:

## Linux Kernel 4.14

```

346 static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
347 {
348     u32 irqnr;
349
350     do {
351         irqnr = gic_read_iar();
352
353         if (likely(irqnr > 15 && irqnr < 1020) || irqnr >= 8192) {
354             int err;
355
356             if (static_key_true(&supports_deactivate))
357                 gic_write_eoir(irqnr);
358             else
359                 isb();
360
361             err = handle_domain_irq(gic_data.domain, irqnr, regs);
362             if (err) {
363                 WARN_ONCE(true, "Unexpected interrupt received!\n");
364                 log_abnormal_wakeup_reason(
365                     "unexpected HW IRQ %u", irqnr);
366                 if (static_key_true(&supports_deactivate)) {
367                     if (irqnr < 8192)
368                         gic_write_dir(irqnr);
369                     } else {
370                         gic_write_eoir(irqnr);
371                     }
372             }
373             continue;
374         }
375         if (irqnr < 16) {
376             gic_write_eoir(irqnr);
377             if (static_key_true(&supports_deactivate))
378                 gic_write_dir(irqnr);
379
380 #ifdef CONFIG_SMP
381         /*
382          * Unlike GICv2, we don't need an smp_rmb() here.
383          * The control dependency from gic_read_iar to
384          * the ISB in gic_write_eoir is enough to ensure
385          * that any shared data read by handle_IPI will
386          * be read after the ACK.
387          */
388         handle_IPI(irqnr, regs);
389 #else
390         WARN_ONCE(true, "Unexpected SGI received!\n");
391 #endif
392         continue;
393     } while (irqnr != ICC_IAR1_EL1_SPURIOUS);
394 }

```

linux Kernel 4.14单独处理SGI中断

## Linux Kernel 5.14

```

679 static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
680 {
681     u32 irqnr;
682
683     irqnr = do_read_iar(regs);
684
685     /* Check for special IDs first */
686     if ((irqnr >= 1020 && irqnr <= 1023))
687         return;
688
689     if (gic_supports_nmi() &&
690         unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
691         gic_handle_nmi(irqnr, regs);
692         return;
693     }
694
695     if (gic_prio_masking_enabled()) {
696         gic_pmr_mask_irqs();
697         gic_arch_enable_irqs();
698     }
699
700     if (static_branch_likely(&supports_deactivate_key))
701         gic_write_eoir(irqnr);
702     else
703         isb();
704
705     if (handle_domain_irq(gic_data.domain, irqnr, regs)) {
706         WARN_ONCE(true, "Unexpected interrupt received!\n");
707         gic_deactivate_unhandled(irqnr);
708     }
709 }

```

CSDN @代码改变世界ctw

## 6 handle\_domain\_irq

## 补充IRQ Domain介绍

在linux kernel中，我们使用下面两个ID来标识一个来自外设的中断：

1、IRQ number。CPU需要为每一个外设中断编号，我们称之IRQ Number。这个IRQ number是一个虚拟的 interrupt ID，和硬件无关，仅仅是被CPU用来标识一个外设中断。

2、HW interrupt ID。对于interrupt controller而言，它收集了多个外设的interrupt request line并向上传递，因此，interrupt controller需要对外设中断进行编码。Interrupt controller用HW interrupt ID来标识外设的中断。在interrupt controller级联的情况下，仅仅用HW interrupt ID已经不能唯一标识一个外设中断，还需要知道该HW interrupt ID所属的interrupt controller（HW interrupt ID在不同的Interrupt controller上是会重复编码的）。

这样，CPU和interrupt controller在标识中断上就有了一些不同的概念，但是，对于驱动工程师而言，我们和CPU视角是一样的，我们只希望得到一个IRQ number，而不关系具体是那个interrupt controller上的那个HW interrupt ID。这样一个好处是在中断相关的硬件发生变化时，驱动软件不需要修改。因此，linux kernel中的中断子系统需要提供一个将HW interrupt ID映射到IRQ number上来的机制...

(本段转载自:[http://www.wowotech.net/linux\\_kernel/irq-domain.html](http://www.wowotech.net/linux_kernel/irq-domain.html))

## 思考:

1、上文提到"在interrupt controller级联的情况下", 为什么会有中断级联, 一个gic控制器可以连接好几千个中断难道还不够吗?

handle\_domain\_irq的处理流程如下所示, 最终是调用到了我们request\_irq注册的中断处理函数.

```

173 static inline int handle_domain_irq(struct irq_domain *domain,
174                                     unsigned int hwirq, struct pt_regs *regs)
175 {
176     return __handle_domain_irq(domain, hwirq, true, regs);
177 }

667 int __handle_domain_irq(struct irq_domain *domain, unsigned int hwirq,
668                         bool lookup, struct pt_regs *regs)
669 {
670     struct pt_regs *old_regs = set_irq_regs(regs);
671     unsigned int irq = hwirq;
672     int ret = 0;
673
674     irq_enter();
675
676     #ifdef CONFIG_IRQ_DOMAIN
677         if (lookup)
678             irq = irq_find_mapping(domain, hwirq); 找到和硬件中断相关联的软中断号
679     #endif
680
681     /*
682      * Some hardware gives randomly wrong interrupts. Rather
683      * than crashing, do something sensible.
684      */
685     if (unlikely(!irq || irq >= nr_irqs)) {
686         ack_bad_irq(irq);
687         ret = -EINVAL;
688     } else {
689         generic_handle_irq(irq);
690     }
691
692     irq_exit();
693     set_irq_regs(old_regs);
694     return ret;
695 }

640 int generic_handle_irq(unsigned int irq)
641 {
642     struct irq_desc *desc = irq_to_desc(irq);
643     struct irq_data *data;
644
645     if (!desc)
646         return -EINVAL;
647
648     data = irq_desc_get_irq_data(desc);
649     if (WARN_ON_ONCE(!in_irq() && handle_enforce_irqctx(data)))
650         return -EPERM;
651
652     generic_handle_irq_desc(desc);
653     return 0;
654 }
655 EXPORT_SYMBOL_GPL(generic_handle_irq);

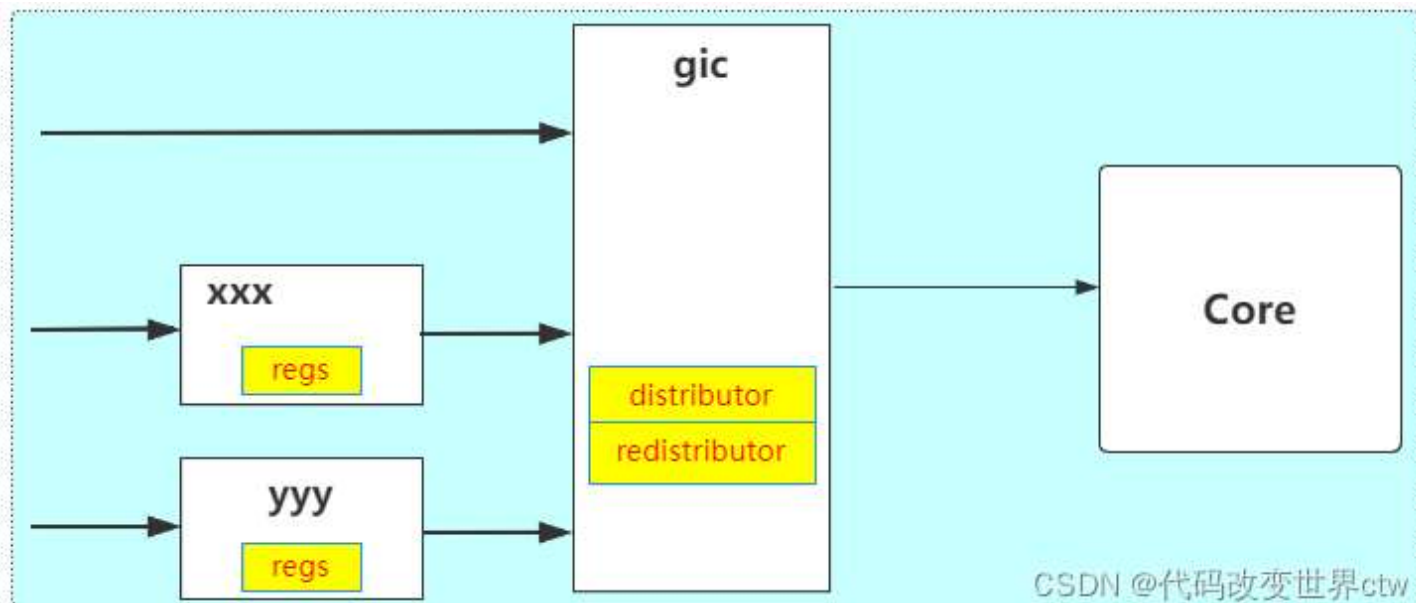
156 static inline void generic_handle_irq_desc(struct irq_desc *desc)
157 {
158     desc->handle_irq(desc); 回调注册的中断handler函数
159 }

```



## 7 关于中断级联的介绍

这也是我想不通的地方，一个gic控制器可以连接好几千个中断难道还不够吗？也许是为了SOC方便设计。例如某平台就使用到了级联的方式



(具体代码，就不能做介绍了)

 <p><b>【*】ARMv8/ARMv9架构从入门到精通（一期）</b> 57节课，23.5h 2023.5 专栏</p>	 <p><b>【*】Trustzone/TEE/安全从入门到精通-标准版</b> 当前:38h, 55节课</p>
 <p><b>【*】ARMv8/ARMv9架构从入门到精通（二期）</b> 当前40h,100节,持续更新中</p>	 <p><b>【*】Trustzone/TEE/安全从入门到精通-高配版</b> Trustzone/TEE/安全从入门到精...</p>
 <p><b>Arm8/Arm9架构从入门到精通(三期)</b> 三期持续更新中....</p>	 <p><b>Trustzone/TEE/安全从入门到精通-（二期）</b></p>

添加v: arm2023, 获取更多信息

Arm8/Arm9架构从入门到精通, Arm8/Arm9架构从入门到精通（一期）, Arm8/Arm9架构从入门到精通（二期）  
Arm8/Arm9架构从入门到精通（三期）, Arm一期、Arm二期、学习资料、免费、下载, 全套资料, Secureboot从入门到精通,  
secureboot训练营, ATF架构从入门到精通、optee系统精讲、secureboot精讲, Trustzone/TEE/安全快速入门班, Trustzone/TEE/安全  
标准版, Trustzone/TEE/安全高配版。全套资料。周贺贺, baron, 代码改变世界, coding\_the\_world, Arm精选, arm\_2023, 安全启  
动, 加密启动  
optee、ATF、TF-A、Trustzone、optee3.14、MMU、VMSA、cache、TLB、arm、armv8、armv9、TEE、安全、内存管理、页表,

Non-cacheable,Cacheable, non-shareable,inner-shareable,outer-shareable, optee、ATF、TF-A、Trustzone、optee3.14、MMU、VMSA、cache、TLB、arm、armv8、armv9、TEE、安全、内存管理、页表...