

# 1. 内核函数调用堆栈打印

我们将介绍内核函数调用堆栈打印的相关方法：

- ❑ `dump_stack` 函数
- ❑ `WARN_ON(condition)`宏
- ❑ `BUG_ON (condition)`宏
- ❑ `panic (fmt...)`函数

`BUG_ON` 和 `WARN_ON` 是 Linux 内核中的两个宏，用于在代码中进行调试和错误处理。它们在不同的情况下用于检查条件，并根据条件的真假触发不同的操作。下面是它们的区别：

❑ `BUG_ON` 用于严重错误检查，如果条件为真，则会导致内核 "BUG"，通常用于开发和调试。

❑ `WARN_ON` 用于警告潜在问题，如果条件为真，则会在系统日志中打印警告，通常用于生产系统中进行诊断。

## 1.1. 什么是堆栈

在 Linux 操作系统中，函数调用栈（Function Call Stack）与一般的计算机系统中的调用栈原理相同，用于管理函数调用和返回的过程。它在程序执行期间跟踪函数的调用层次、局部变量的分配和释放，以及函数之间的数据传递。函数调用栈在 Linux 内核中起着关键作用，帮助管理用户空间程序的执行。

下面是关于 Linux 函数调用栈的一些要点：

❑ **栈帧（Stack Frame）**：每次函数被调用时，一个新的栈帧会被压入调用栈的顶部。栈帧包含了函数的参数、局部变量、返回地址以及其他与函数执行相关的信息。当函数执行完毕后，对应的栈帧会被弹出，控制权返回给调用它的函数。

❑ **栈指针（Stack Pointer）**：栈指针是一个指向当前栈顶的指针，用于确定栈帧的位置。

❑ **栈溢出（Stack Overflow）**：函数调用栈的大小是有限的，过多的函数嵌套或递归调用可能导致栈溢出。栈溢出会导致程序异常终止，因此在编写程序时需要注意控制函数调用的层次和递归深度。

## 1.2. dump\_stack 函数

`dump_stack` 是一个在 Linux 内核中用于打印调试信息的函数。它通常用于故障排除和调试，以帮助开发人员识别发生问题的位置和上下文。

具体来说，`dump_stack` 函数的作用是在调试输出中显示当前的函数调用栈信息，包括函数调用的层次、调用链以及各个栈帧中的相关信息。这对于跟踪程序的执行路径、定位错误和了解程序的上下文非常有帮助。

`dump_stack` 函数通常在内核代码中使用，例如在驱动程序、中断处理程序、异常处理程序或其他需要进行故障排除的地方。它可以用于以下几种情况：

- ❑ **故障排除**：当发生异常或错误时，通过在相关位置调用 `dump_stack`，可以在控制

台或系统日志中打印出调用栈信息，以帮助开发人员快速定位问题。

❑ 分析崩溃：当内核发生崩溃（kernel panic）时，`dump_stack` 可以用于在崩溃信息中打印出最后的函数调用栈，帮助分析导致崩溃的原因。

❑ 性能分析：在性能分析场景中，`dump_stack` 可以用于捕获特定代码路径的函数调用栈，以帮助分析程序的性能瓶颈。

### 1.2.1. 什么是 `dump_stack`

`dump_stack` 函数的主要作用是将当前内核堆栈的跟踪信息打印到系统日志中。这有助于开发人员在调试内核代码时了解当前的函数调用路径，以及在哪里发生了特定的问题。通过分析 `dump_stack` 的输出，开发人员可以识别代码中的潜在问题，例如未预期的函数调用顺序、死锁等。

### 1.2.2. 如何使用 `dump_stack`

我们以分析 USB 调用关系为例：

```
#include <asm/ptrace.h>

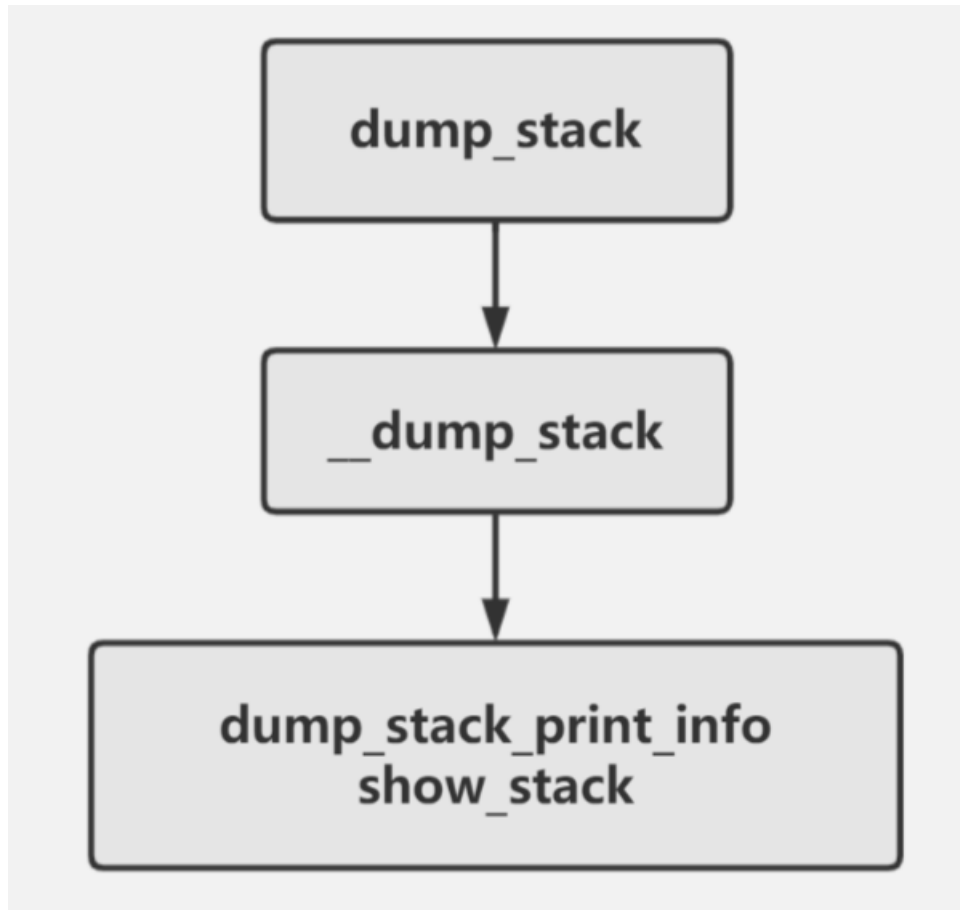
static int composite_bind(struct usb_gadget *gadget,
                          struct usb_gadget_driver *gdriver)
{
...
+   printk("[xxx-dump] in %s, line = %d, dump start\n", __func__, __LINE__);
+   dump_stack();
+   printk("[xxx-dump] in %s, line = %d, dump end\n", __func__, __LINE__);
...
}
```

打印结果如下：

```
[ 3.872143] Call trace:
[ 3.872155] [< (ptrval)>] dump_backtrace+0x0/0x520
[ 3.872159] [< (ptrval)>] show_stack+0x14/0x20
[ 3.872164] [< (ptrval)>] dump_stack+0x8c/0xac
[ 3.872170] [< (ptrval)>] composite_bind+0x3c/0x1c8
[ 3.872173] [< (ptrval)>] udc_bind_to_driver+0x8c/0x118
[ 3.872177] [< (ptrval)>] usb_gadget_probe_driver+0xc4/0xf0
[ 3.872180] [< (ptrval)>] usb_composite_probe+0xb0/0xd8
[ 3.872185] [< (ptrval)>] init+0x13c/0x270
[ 3.872188] [< (ptrval)>] do_one_initcall+0x4c/0x150
[ 3.872193] [< (ptrval)>] kernel_init_freeable+0x138/0x1dc
[ 3.872198] [< (ptrval)>] kernel_init+0x10/0x110
[ 3.872201] [< (ptrval)>] ret_from_fork+0x10/0x18
[ 3.872204] [xxx-dump] in composite_bind, line = 2263, dump end
```

### 1.2.3. dump\_stack 原理

通过 grep，发现 dump\_stack 函数原型存在于 kernel/lib/dump\_stack.c 文件中。它的实现流程如下图所示：



```
static void __dump_stack(void)
{
    dump_stack_print_info(KERN_DEFAULT);
    show_stack(NULL, NULL);
}
```

可以看到关键的两个函数分别是 dump\_stack\_print\_info 和 show\_stack。其中第一个函数是用来打印 info 信息的，而第二个函数是用来打印 Call trace 的。

## 1.3. WARN\_ON 宏的使用

WARN\_ON (condition)函数作用：在括号中的条件成立时，内核会抛出栈回溯，打印函数的调用关系。通常用于内核抛出一个警告，暗示某种不太合理的事情发生了。

WARN\_ON 实际上也是调用 dump\_stack，只是多了参数 condition 判断条件是否成立，例如 WARN\_ON(1)则条件判断成功，函数会成功执行。

### 1.3.1. WARN\_ON 宏的原型

**WARN\_ON** 是 Linux 内核代码中的一个宏，用于在代码中进行警告检查。它用于检查某些条件是否满足，如果条件不满足，它会在内核日志中输出一条警告消息。**WARN\_ON** 宏的原型如下：

```
#define WARN_ON(condition) ({ \
    int __ret_warn_on = !(condition); \
    if (unlikely(__ret_warn_on)) \
        printk(KERN_WARNING "WARNING: Condition %s at %s:%d/%s\n", \
            #condition, __FILE__, __LINE__, __func__); \
    unlikely(__ret_warn_on); \
})
```

这个宏在编译时会将条件 **condition** 转换为一个非零值（如果条件满足），然后在运行时检查这个值，如果为真（即条件满足），就会输出一条警告信息。这种警告通常用于标识一些可能会导致问题的代码路径，以便在开发和调试过程中提醒开发人员进行进一步的检查。

以下是 **WARN\_ON** 宏的一些关键部分解释：

- ❑ **condition**: 要检查的条件表达式。
- ❑ **!(condition)**: 将条件表达式转换为整数值 0 或 1。
- ❑ **\_\_ret\_warn\_on**: 用于存储条件的检查结果。
- ❑ **unlikely(\_\_ret\_warn\_on)**: 将条件检查的结果传递给 **unlikely** 宏，用于优化分支预测。

测。

### 1.3.2. WARN\_ON 案例

```
#include <linux/module.h>
#include <linux/kernel.h>
static int __init helloworld_init(void)
{
    printk(KERN_EMERG "helloworld_init\r\n");
    WARN_ON(1);
    return 0;
}
static void __exit helloworld_exit(void)
{
    printk(KERN_EMERG "helloworld_exit\r\n");
}
module_init(helloworld_init);
module_exit(helloworld_exit);
MODULE_LICENSE("GPL v2");
```

在第 6 行添加了 **WARN\_ON(1)**，驱动加载之后打印信息如下：

```
1054.509100] pstate: 40400009 (nZcv daIf +PAN -UAO)
1054.509117] pc : helloworld_init+0x28/0x1000 [helloworld]
1054.509131] lr : helloworld_init+0x28/0x1000 [helloworld]
1054.509142] sp : ffffffff800c8ebb60
1054.509152] x29: ffffffff800c8ebb60 x28: ffffffff80011b2000
1054.509165] x27: 0000000000000000 x26: 0000000000000000
1054.509177] x25: ffffffff80011ae250 x24: ffffffff80097a7000
1054.509189] x23: ffffffff80097a7000 x22: 0000000000000000
1054.509200] x21: ffffffff80011b1000 x20: ffffffff8009546000
1054.509212] x19: ffffffff800993cca0 x18: 000000000000000a
1054.509221] x17: 0000000000000000 x16: 0000000000000000
1054.509233] x15: 0000000000007c3d4 x14: ffffffff80099c0e37
1054.509245] x13: ffffffff80099c0e37 x12: 0000000000000030
1054.509257] x11: 00000000ffffffffff x10: ffffffff80099c0e3f
1054.509268] x9 : 0000000005f5e0ff x8 : 2d5d206572656820
1054.509279] x7 : 747563205b2d2d2d x6 : 00000000000038530
1054.509291] x5 : 0000000000000001 x4 : ffffffff8007fed1900
1054.509302] x3 : ffffffff8007fed1900 x2 : acdae0e19d26f800
1054.509313] x1 : 0000000000000000 x0 : 0000000000000024
1054.509329]
1054.509329] PC: 0xfffffff80011b0fa8:
1054.509340] 0fa8 *****
1054.509376] 0fc8 *****
1054.509399] 0fe8 ***** a9bf7bfd 91003fd
1054.509420] 1008 aa1e03e0 d503201f 90ffffe0 91011800 95fa6138 90ffffe0 91015c00 95fa6135
1054.509437] 1028 d4210000 52800000 a8c17bfd d65f03c0 00000000 00000000 00000000 00000000
1054.509456] 1048 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1054.509474] 1068 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
1054.509490] 1088 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

可以看到 helloworld\_init 函数的调用关系以及寄存器值就都打印了出来。  
至此关于 WARN\_ON 函数的测试就完成了。

## 1.4. BUG\_ON 宏的使用

内核中有许多地方调用类似 BUG\_ON()的语句，它非常像一个内核运行时的断言，意味着本来不该执行到 BUG\_ON()这条语句，一旦 BUG\_ON()执行内核就会立刻抛出 oops，导致栈的回溯和错误信息的打印。大部分体系结构把 BUG()和 BUG\_ON()定义成某种非法操作，这样自然会产生需要的 oops。参数 condition 判断条件是否成立，例如 BUG\_ON(1)则条件判断成功，函数会成功执行。

### 1.4.1. BUG\_ON 宏的原型

```
#define BUG_ON(condition) do { if (condition) BUG(); } while (0)
```

该宏采用了一个条件表达式作为参数，并在条件为真时触发内核的 "BUG"。

### 1.4.2. BUG\_ON 案例

这里仍然以最简单的 helloworld 驱动为例进行 BUG\_ON 函数演示：

```
#include <linux/module.h>
#include <linux/kernel.h>
static int __init helloworld_init(void)
{
    printk(KERN_EMERG "helloworld_init\r\n");
    BUGON(1);
    return 0;
}
static void __exit helloworld_exit(void)
```

```
{
    printk(KERN_EMERG "helloworld_exit\r\n");
}
module_init(helloworld_init);
module_exit(helloworld_exit);
MODULE_LICENSE("GPL v2");
```

```
[ 56.922491] pstate: 40400009 (nZcv daif +PAN -UAO)
[ 56.927323] pc : helloworld_init+0x1c/0x1000 [helloworld]
[ 56.932758] lr : helloworld_init+0x1c/0x1000 [helloworld]
[ 56.938170] sp : ffffffff800c773b60
[ 56.941503] x29: ffffffff800c773b60 x28: ffffffff80011a2000
[ 56.946837] x27: 0000000000000000 x26: 0000000000000000
[ 56.952166] x25: ffffffff800119e250 x24: ffffffff80097a7000
[ 56.957494] x23: ffffffff80097a7000 x22: 0000000000000000
[ 56.962820] x21: ffffffff80011a1000 x20: ffffffff8009546000
[ 56.968147] x19: ffffffff800993cca0 x18: 000000000000000a
[ 56.973472] x17: 0000000000000000 x16: 0000000000000000
[ 56.978801] x15: 000000000000d55fc x14: ffffffff80899c0e37
[ 56.984133] x13: ffffffff8000000000 x12: 0000000000000030
[ 56.989467] x11: 00000000ffffffffff x10: ffffffff80099c0e3f
[ 56.994794] x9 : 0000000005f5e0ff x8 : 6c726f776f6c6c65
[ 57.000120] x7 : 00000000017ed73b x6 : ffffffff807feb9908
[ 57.005449] x5 : ffffffff807feb9908 x4 : 0000000000000000
[ 57.010777] x3 : ffffffff807fec2648 x2 : d9912f59efd61900
[ 57.016104] x1 : 0000000000000000 x0 : 000000000000000f
[ 57.021431]
```

可以看到 `helloworld_init` 函数的调用关系以及寄存器值就都打印了出来。

## 1.5. panic (fmt...)函数

`panic(fmt...)`函数:输出打印会造成系统死机并将函数的调用关系以及寄存器值就都打印了出来。

### 1.5.1. panic 的原型

`panic` 函数是 Linux 内核中用于触发系统崩溃（kernel panic）的函数。当内核代码遇到无法继续执行的严重错误情况时，可以调用 `panic` 函数，导致系统停止运行并输出崩溃信息。这有助于开发人员在故障排除过程中获取关键信息。

```
void panic(const char *fmt, ...) __noreturn __cold;
```

❑ `void panic(const char *fmt, ...)`: 这部分声明了 `panic` 函数，它接受一个格式化字符串 `fmt` 和可变参数列表 `...`，用于指定触发内核崩溃时要输出的错误信息。

❑ `__noreturn`: 这是一个编译器属性，表示函数不会返回。一旦 `panic` 函数被调用，系统将无法继续正常执行，因此这个属性用于告诉编译器不要生成函数返回的相关代码。这是因为一旦触发内核崩溃，不应该尝试从 `panic` 函数返回到正常执行。

❑ `__cold`: 这也是一个编译器属性，用于标记函数为“冷”函数，即很少被调用的函数。在内核代码中，一些函数可能只在特定的情况下被调用，因此标记为冷函数可以优化代码，以便更好地适应 CPU 缓存。`panic` 函数通常是在严重错误情况下才会被调用，因此被标记为冷函数是合适的。



### 1.5.2. panic 的案例

这里仍然以最简单的 helloworld 驱动为例进行 panic 函数的演示:

```
#include <linux/module.h>
#include <linux/kernel.h>
static int __init helloworld_init(void)
{
    printk(KERN_EMERG "helloworld_init\r\n");
    panic("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
    return 0;
}
static void __exit helloworld_exit(void)
{
    printk(KERN_EMERG "helloworld_exit\r\n");
}
module_init(helloworld_init);
module_exit(helloworld_exit);
MODULE_LICENSE("GPL v2");
```

和原 helloworld 驱动程序相比, 在第 6 行添加了 panic("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"), 驱动加载之后打印信息如下

```
[ 70.102261] Call trace:
[ 70.104741] dump_backtrace+0x0/0x188
[ 70.108418] show_stack+0x24/0x30
[ 70.111746] dump_stack+0x8c/0xb4
[ 70.115067] panic+0x138/0x2b0
[ 70.118137] helloworld_init+0x28/0x1000 [helloworld]
[ 70.123205] do_one_initcall+0xa0/0x1c0
[ 70.127047] do_init_module+0x54/0x1d8
[ 70.130809] load_module+0x1ae4/0x1c30
[ 70.134566] __se_sys_finit_module+0xd8/0xf4
[ 70.138847] __arm64_sys_finit_module+0x24/0x30
[ 70.143377] el0_svc_common.constprop.0+0xe8/0x168
[ 70.148171] el0_svc_handler+0x70/0x8c
[ 70.151928] el0_svc+0x8/0xc
[ 70.154827] SMP: stopping secondary CPUs
[ 70.158833] PMU CRU:
[ 70.161045] 00000000: 00006064 00001481 00000000 00000007 00007f00 00000000 00000000 00000000
[ 70.169575] 00000020: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

### 1.5.3. panic 的手动触发

执行 echo c > /proc/sysrq-trigger 可以触发内核 panic, 可借此研究内核 panic 子系统。

```
[root@imx6ull:~]# echo c > /proc/sysrq-trigger
[73521.235605] sysrq: SysRq : Trigger a crash
[73521.239864] Unable to handle kernel NULL pointer dereference at virtual address 00000000
[73521.249822] pgd = 891ac000
[73521.252600] [00000000] *pgd=88d39835, *pte=00000000, *ppte=00000000
[73521.260150] Internal error: Oops: 817 [#1] PREEMPT SMP ARM
[73521.265600] Modules linked in: inv_mpu6050_spi inv_mpu6050 evbug 100ask_adxl345_spi 100ask_spidev 100ask_irda 100ask_rc_nec 100ask_dht11 100ask_ds18b20 8723bu
[73521.280240] CPU: 0 PID: 330 Comm: sh Not tainted 4.9.80 #1
[73521.285782] Hardware name: Freescale i.MX6 UltraLite (Device Tree)
[73521.292012] task: 8877c000 task.stack: 885fe000
[73521.296607] PC is at sysrq_handle_crash+0x24/0x2c
[73521.301361] LR is at sysrq_handle_crash+0x20/0x2c
[73521.306120] pc : [<804943e0>] lr : [<804943dc>] psr: 600c0013
[73521.306120] sp : 885fe000 ip : 00000000 fp : 000057d9
[73521.317650] r10: 00000000 r9 : 00000002 r8 : 00000000
[73521.322024] r7 : 810309e4 r6 : 00000007 r5 : 00000063 r4 : 00000001
[73521.329498] r3 : 00000000 r2 : 00000000 r1 : 00000001 r0 : 00000063
[73521.336876] Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment none
[73521.343261] Control: 10c53c7d Table: 891ac06a DAC: 00000051
[73521.349054] Process sh (pid: 330, stack limit = 0x805fe210)
[73521.354673] Stack: (0x805fe000 to 0x805fe000)
[73521.359060] fea0: 010100dc 80494934 00000002 00000000 00000000 00000000 882ab080
[73521.367344] fe0: 00000000 00000002 00000000 80494da0 80494d4c 8025b0c4 89192e40 8025b068
[73521.375590] fee0: 0031e540 885fff80 00000000 80201dec 863e0778 00000301 00000000 00000060
[73521.383835] ffe0: 00000000 00000000 00000000 885fff10 885fe000 80213ce0 00000001 885fffb0
[73521.392080] ff20: 888bb000 0000000a 89192cc0 0000000a 00000000 88036d98 88036d98 89192e40
[73521.400325] ffa0: 00000002 0031e540 885fff80 00000000 00000002 80202bec 0000000a 8021fc78
[73521.408570] ff40: 00000000 89192e40 89192e40 00000000 00000000 0031e540 00000002 80203094
[73521.416816] ffb0: 00000000 00000000 0031f5e0 00000002 0031e540 76e9fd58 00000004 80108544
[73521.425060] ffa0: 885fe000 80108380 00000002 0031e540 00000001 0031e540 00000002 00000000
[73521.433306] ff0: 00000002 0031e540 76e9fd58 00000004 00000001 0009e980 000b0220 000b57d0
[73521.441550] ffe0: 00000000 7e873a34 76e036c9 76e3f006 000c0030 00000001 00000000 00000000
[73521.449826] [<804943e0>] (sysrq_handle_crash) from [<80494934>] (_handle_sysrq+0x124/0x180)
[73521.458339] [<80494934>] (_handle_sysrq) from [<80494da0>] (write_sysrq_trigger+0x54/0x64)
[73521.466793] [<80494da0>] (write_sysrq_trigger) from [<8025b0c4>] (proc_reg_write+0x5c/0x80)
[73521.475191] [<8025b0c4>] (proc_reg_write) from [<80201dec>] (_vfs_write+0x1c/0x114)
[73521.483600] [<80201dec>] (_vfs_write) from [<80202bec>] (vfs_write+0xa4/0x168)
[73521.490392] [<80202bec>] (vfs_write) from [<80203984>] (SyS_write+0x3c/0x90)
```

## 1.5.4. panic 相关调试节点

### □ /proc/sys/kernel/panic 节点

```
[root@imx6ull:~]# cat /proc/sys/kernel/panic
0
```

`/proc/sys/kernel/panic` 是 Linux 操作系统中的一个特殊文件，用于配置内核发生严重错误时的行为。当系统遇到无法处理的严重错误（例如内核崩溃或死锁）时，内核会尝试采取一些措施，其中之一就是触发系统崩溃（kernel panic）。`/proc/sys/kernel/panic` 节点允许您配置触发内核崩溃前的延迟时间。

该文件的内容是一个整数，表示在内核发生严重错误后触发系统崩溃之前的等待时间（以秒为单位）。如果将该值设置为 0，则内核将立即触发崩溃，而不会等待。如果设置为一个正整数，则内核会等待指定的秒数后触发崩溃。如果将该值设置为负数，则内核将不会自动触发崩溃，而是等待管理员手动干预。

以下是一些示例：

- `/proc/sys/kernel/panic` 的内容为 0，表示内核遇到严重错误会立即触发崩溃。
- `/proc/sys/kernel/panic` 的内容为 10，表示内核在遇到严重错误后会等待 10 秒后触发崩溃。
- `/proc/sys/kernel/panic` 的内容为 -1，表示内核在遇到严重错误后不会自动触发崩溃，而是等待管理员手动决定。

这是一个系统级的设置，对于确保系统的稳定性和可靠性非常重要。在生产环境中，根据需要进行适当的配置，以便在严重错误发生时能够及时采取适当的操作。

### □ /proc/sys/kernel/panic\_on\_oops 节点

```
[root@imx6ull:~]# cat /proc/sys/kernel/panic_on_oops
0
```

`panic_on_oops` 是 Linux 操作系统中的一个配置节点，用于控制在内核发生 oops 错误（较轻的内核错误）时是否触发系统崩溃（kernel panic）。当内核代码遇到一些未预期的错误情况，但仍然可以继续执行时，它可能会产生 oops 错误。`panic_on_oops` 节点允许您配置内核在遇到 oops 错误时的行为。



该节点的内容是一个整数，表示内核在发生 `oops` 错误时的行为方式：

- 0：不触发系统崩溃，只记录 `oops` 错误日志，内核继续执行。
- 1：触发系统崩溃（`kernel panic`），即使发生较轻的 `oops` 错误。

默认情况下，大多数 Linux 系统中的 `panic_on_oops` 节点的值为 0，这意味着内核在遇到 `oops` 错误时不会触发系统崩溃，而只会记录错误信息。

在调试和故障排除期间，将 `panic_on_oops` 设置为 1 可能会更有帮助，因为它会在发生 `oops` 错误时立即停止系统并生成调试信息，以便进行更深入的分析。但在生产环境中，通常建议将其保持为默认值 0，以确保系统的稳定性。

❏ `/proc/sys/kernel/panic_on_oops` 节点

```
[root@imx6ull:~]# cat /proc/sys/kernel/panic_on_warn  
0
```

`/proc/sys/kernel/panic_on_oops` 是 Linux 操作系统中的一个配置节点，用于控制在内核发生 `oops` 错误（较轻的内核错误）时是否触发系统崩溃（`kernel panic`）。当内核代码遇到一些未预期的错误情况，但仍然可以继续执行时，它可能会产生 `oops` 错误。`panic_on_oops` 节点允许您配置内核在遇到 `oops` 错误时的行为。

该节点的内容是一个整数，表示内核在发生 `oops` 错误时的行为方式：

- 0：不触发系统崩溃，只记录 `oops` 错误信息，内核继续执行。
- 1：触发系统崩溃（`kernel panic`），即使发生较轻的 `oops` 错误。

默认情况下，大多数 Linux 系统中的 `panic_on_oops` 节点的值为 0，这意味着内核在遇到 `oops` 错误时不会触发系统崩溃，而只会记录错误信息。