网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

linux 驱动头文件说明驱动程序：

#include <linux/***.h> 是在linux-2.6.29/include/linux下面寻找源文件。
#include <asm/***.h> 是在linux-2.6.29/arch/arm/include/asm下面寻找源文件。
#include <mach/***.h> 是在linux-2.6.29/arch/arm/mach-s3c2410/include/mach下面寻找源文件。

#include <plat/regs-adc.h>在linux-2.6.31_TX2440A20100510\linux-2.6.31_TX2440A\arch\arm\plat-s3c\include\plat
#include <linux/module.h> //最基本的文件，支持动态添加和卸载模块。Hello World驱动要这一个文件就可以了
#include <linux/fs.h> //包含了文件操作相关struct的定义，例如大名鼎鼎的struct file_operations
//包含了struct inode 的定义，MINOR、MAJOR的头文件。
#include <linux/errno.h> //包含了对返回值的宏定义，这样用户程序可以用perror输出错误信息。

```
#include <linux/types.h> //对一些特殊类型的定义，例如dev_t, off_t, pid_t.其实这些类型大部分都是unsigned int型通过一连串的typedef变过来的，只是为了方便阅读。
#include <linux/cdev.h> //对字符设备结构cdev以及一系列的操作函数的定义。//包含了cdev 结构及相关函数的定义。
#include <linux/wait.h> //等代队列相关头文件//内核等待队列，它包含了自旋锁的头文件
```

```
#include <linux/init.h>
#include <linux/kernel.h>

#include <linux/slab.h>                 //包含了kcalloc、kzalloc内存分配函数的定义。
#include <linux/uaccess.h>          //包含了copy_to_user、copy_from_user等内核访问用户进程内存地址的函数定义。
#include <linux/device.h>             //包含了device、class 等结构的定义
#include <linux/io.h>                     //包含了ioremap、iowrite等内核访问IO内存等函数的定义。
#include <linux/miscdevice.h>     //包含了miscdevice结构的定义及相关的操作函数。
#include <linux/interrupt.h>        //使用中断必须的头文件
#include <mach/irqs.h>               //使用中断必须的头文件
#include <asm/bitops.h>             //包含set_bit等位操作函数，实现Input子系统时可用。
#include <linux/semaphore.h>    //使用信号量必须的头文件
#include <linux/spinlock.h>         //自旋锁

#include <linux/sched.h>            //内核等待队列中要使用的TASK_NORMAL、TASK_INTERRUPTIBLE包含在这个头文件
#include <linux/kfifo.h>              //fifo环形队列
#include <linux/timer.h>             //内核定时器
#include <linux/input.h>             //中断处理
```

头文件主目录 include

头文件目录中总共有 32 个.h 头文件。其中主目录下有 13 个，asm 子目录中有 4 个，linux 子目录中有 10 个，sys 子目录中有 5 个。这些头文件各自的功能如下，具体的作用和所包含的信息请参见第 14 章。

<a.out.h>：a.out 头文件，定义了 a.out 执行文件格式和一些宏。
<const.h>：常数符号头文件，目前仅定义了 i 节点中 i_mode 字段的各标志位。
<ctype.h>：字符类型头文件，定义了一些有关字符类型判断和转换的宏。
<errno.h>：错误号头文件，包含系统中各种出错号。(Linus 从 minix 中引进的)。
<fcntl.h>：文件控制头文件，用于文件及其描述符的操作控制常数符号的定义。
<signal.h>：信号头文件，定义信号符号常量，信号结构以及信号操作函数原型。
<stdarg.h>：标准参数头文件，以宏的形式定义变量参数列表。主要说明了一个类型（va_list）和 3 个宏（va_start, va_arg 和 va_end），用于 vsprintf、vprintf、vfprintf 函数。

<stddef.h>：标准定义头文件，定义了 NULL， offsetof(TYPE， MEMBER)。
<string.h>：字符串头文件，主要定义了一些有关字符串操作的嵌入函数。
<termios.h>：终端输入输出函数头文件，主要定义控制异步通信口的终端接口。
<time.h>：时间类型头文件，主要定义了 tm 结构和一些有关时间的函数原形。
<unistd.h>：Linux 标准头文件，定义了各种符号常数和类型，并声明了各种函数。如，定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。
<utime.h>：用户时间头文件，定义了访问和修改时间结构以及 utime()原型。

（1）体系结构相关头文件子目录 include/asm

这些头文件主要定义了一些与 CPU 体系结构密切相关的数据结构、宏函数和变量。共 4 个文件。

<asm/io.h>：I/O 头文件，以宏的嵌入汇编程序形式定义对 I/O 端口操作的函数。
<asm/memory.h>：内存拷贝头文件，含有 memcpy()嵌入式汇编宏函数。
<asm/segment.h>：段操作头文件，定义了有关段寄存器操作的嵌入式汇编函数。
<asm/system.h>：系统头文件，定义了设置或修改描述符/中断门等的嵌入式汇编宏。

（2）Linux 内核专用头文件子目录 include/linux

<linux/config.h>：内核配置头文件，定义键盘语言和硬盘类型（HD_TYPE）可选项。
<linux/fdreg.h>：软驱头文件，含有软盘控制器参数的一些定义。
<linux/fs.h>：文件系统头文件，定义文件表结构（file,buffer_head,m_inode等）。
<linux/hdreg.h>：硬盘参数头文件，定义访问硬盘寄存器端口、状态码和分区表等信息。
<linux/head.h>：head头文件，定义了段描述符的简单结构，和几个选择符常量。
<linux/kernel.h>：内核头文件，含有一些内核常用函数的原形定义。
<linux/mm.h>：内存管理头文件，含有页面大小定义和一些页面释放函数原型。
<linux/sched.h>： 调度程序头文件，定义了任务结构task_struct、初始任务0的数据，以及一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
<linux/sys.h>：系统调用头文件，含有72个系统调用C函数处理程序,以"sys_"开头。
<linux/tty.h>：tty头文件，定义了有关tty_io，串行通信方面的参数、常数。

（3）系统专用数据结构子目录 include/sys

<sys/stat.h>： 文件状态头文件，含有文件或文件系统状态结构stat{}和常量。
<sys/times.h>：定义了进程中运行时间结构tms以及times()函数原型。
<sys/types.h>：类型头文件，定义了基本的系统数据类型。
<sys/utsname.h>：系统名称结构头文件。
<sys/wait.h>：等待调用头文件，定义系统调用wait()和waitpid()及相关常数符号。

linux 内核中链表的实现

linux 内核中，数量巨大的数据是靠链表链接起来的，链表结构在内核中起着异常重要的作用。在 linux 内核中，链表的实现是以一个非常巧妙，非常有新意的方式来实现的，它脱离了传统数据结构课程上所教导的链表的实现方法，而是以一种非常有新意，而且也不缺乏适用性的方式来实现的，下面我就来分析一下 linux 内核中关于链表实现的方法。

```
struct list_head {
  struct list_head *next, *prev;
};
```

这个没什么太多可说的，定义一个双链表的结构

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
```

这个宏的作用是把参数为 name 的地址，赋值给结构里的头两个变量

```
#define LIST_HEAD(name) \
  struct list_head name = LIST_HEAD_INIT(name)
```

定义了一个链表类的变量，变量名字命名为所给的参数，然后把链表结构赋值，相当于环起来了。

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
  list->next = list;
  list->prev = list;
}
```

给定一个双链表类型的变量，使之初使化，把它环起来，根据以上的描述，那么就有这样的结论：

```
  LIST_HEAD(foo);
```

与

```
  struct list_head foo;
  INIT_LIST_HEAD(&foo);
```

等效。

```
#ifndef CONFIG_DEBUG_LIST
static inline void __list_add(struct list_head *new,
                struct list_head *prev,
                struct list_head *next)
{
  next->prev = new;
  new->next = next;
  new->prev = prev;
  prev->next = new;
```

```
}
#else
extern void __list_add(struct list_head *new,
                       struct list_head *prev,
                       struct list_head *next);
#endif
```

__list_add 这个函数是在已知的 prev 与 next 之间插一个 new 的元素。这个操作只是在内部使用，因为只有特定的人，才能确切知道 prev 与 next 之间，是不是在之前是相连的，没有其它元素。要是其间有其它元素，这个操作就会导致一些不确定的后果。另外，如果是要做 list debug，则这个操作就不在这里实现了，是在 list_debug.c 这个里面实现的了。

```
static inline void list_add(struct list_head *new, struct list_head *head)
{
  __list_add(new, head, head->next);
}
```

结合上一个函数的说明，这个操作的作用就很明了啦，就是在 head 这个元素后面，加一个 new 的元素。

```
static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
  __list_add(new, head->prev, head);
}
```

这个函数也很容易看懂，就是在 head 与 head->prev 这两个元素之间加一个元素。因为这个链表是双向的，所以 head 的头一个，其实就是尾了，所以这个函数的名字后面就有个 tail 了。

```
static inline void __list_del(struct list_head * prev, struct list_head * next)
{
  next->prev = prev;
  prev->next = next;
}
```

这个函数也是一个内部函数，用来删除 prev 与 next 这两个元素之间的一个元素。至于为什么，原因与__list_add 类似。

```
#ifndef CONFIG_DEBUG_LIST
static inline void list_del(struct list_head *entry)
{
  __list_del(entry->prev, entry->next);
  entry->next = LIST_POISON1;
  entry->prev = LIST_POISON2;
}
#else
extern void list_del(struct list_head *entry);
#endif
```

删除指定的元素的头一个元素与后一个元素之间的元素，所以很容易理解，就是删除指定元素的了。删除完后，给元素的两个指针赋值，赋一个特殊的值，用来表示此节点已经被删除。但这里为什么要使用这个特殊的值，而不使用 0 值呢？因为这是为了调试方便。如果所有的无效指针都是使用零，那么在访问的时候，报页错误的时候，都是提示地址 0 有页错误，这样没有使用一个特殊的值来进行特殊的定位这样方便，可见，Linux 内核中的代码，处处都是妙招啊。同样，如果是 debug 版的话，这个函数在 list_debug.c 里实现。

```
static inline void list_replace(struct list_head *old,
        struct list_head *new)
{
  new->next = old->next;
  new->next->prev = new;
  new->prev = old->prev;
  new->prev->next = new;
}
```

简单的两个元素的替换，容易理解。

```
static inline void list_replace_init(struct list_head *old,
          struct list_head *new)
{
  list_replace(old, new);
  INIT_LIST_HEAD(old);
}
```

把旧的链表替换了以后，给旧链接初始化，让它成为一个新的，只有一个元素的链表。

```
static inline void list_del_init(struct list_head *entry)
{
  __list_del(entry->prev, entry->next);
  INIT_LIST_HEAD(entry);
}
```

把元素从链表里删除以后，再给它初始化了。

```
static inline void list_move(struct list_head *list, struct list_head *head)
{
  __list_del(list->prev, list->next);
  list_add(list, head);
}
```

把 list 这个元素删掉，然后再把它加到 head 这个元素的后面。

```
static inline void list_move_tail(struct list_head *list,
           struct list_head *head)
{
  __list_del(list->prev, list->next);
  list_add_tail(list, head);
}
```

把 list 这个元素删掉，然后把它加到 head 这个链表之尾。

```
static inline int list_is_last(const struct list_head *list,
        const struct list_head *head)
{
  return list->next == head;
}
```

判断 list 这个元素是否为 head 这个链表的尾，因为是双向链表，所以这样判断很容易。

```
static inline int list_empty(const struct list_head *head)
{
  return head->next == head;
}
```

判断 head 这个链表，是否为空。如果 head 的下一个元素指向它自己，那么它就是空的了。

```
static inline int list_empty_careful(const struct list_head *head)
{
  struct list_head *next = head->next;
  return (next == head) && (next == head->prev);
}
```

用来判断这个链表是空的，并且不是被正在修改这样的状态之中。不过这个操作，得是在有同步技术支持的时候才起作用，或是只有单进程能执行相关代码的时候才能够起作用。

```
static inline int list_is_singular(const struct list_head *head)
{
  return !list_empty(head) && (head->next == head->prev);
}
```

用来判断这个链表是否只有单个元素。判断规则就是 head 不能为空，并且 head 的下一个元素与 head 的上一个元素相同。

```
static inline void __list_cut_position(struct list_head *list,
    struct list_head *head, struct list_head *entry)
{
  struct list_head *new_first = entry->next;
  list->next = head->next;
  list->next->prev = list;
  list->prev = entry;
  entry->next = list;
  head->next = new_first;
  new_first->prev = head;
}
```

这个函数所做的操作，就是把 entry 放到 list 列表的左面，当做 list 链表的 prev 元素，list 链表的 next 元素指向原先 head 元素的 next 元素。entry 后面的元素都链到 head 链表右面，也即后面。我这里做的说明，引用了一个假设，那就是链表的元素是从左往右展开的，左边的元素在前，右边元素在后。起初看这个函数的时候，你肯定是很奇怪的，

为什么要这样操作呢？因为它是一个以__开头的函数，有这样的开头字符，就说明这个函数是一个内部函数，既然是内部函数，不是为外面的人调用而写，那么看起来奇怪就不算什么了，这点跟上面我提到的那些特殊的内部函数一样。这个函数主要是为下面的这个函数 list_cut_position 来服务的。

```
static inline void list_cut_position(struct list_head *list,
    struct list_head *head, struct list_head *entry)
{
  if (list_empty(head))
    return;
  if (list_is_singular(head) &&
    (head->next != entry && head != entry))
    return;
  if (entry == head)
    INIT_LIST_HEAD(list);
  else
    __list_cut_position(list, head, entry);
}
```

这个函数的功能，就是把 head 这个链表里，上至表头（不含），下至 entry 这个元素（包括），给移到 list 链表内，组成一个新的链表。然后 head 链表则与 entry 后面的元素形成一个新的链表。这里要求 entry 这个元素，一定要在 head 这个链表内。如果 head 里只有一个 entry 这个元素，则初始化 list 链表。

```
static inline void __list_splice(const struct list_head *list,
          struct list_head *prev,
          struct list_head *next)
{
  struct list_head *first = list->next;
  struct list_head *last = list->prev;
  first->prev = prev;
  prev->next = first;
  last->next = next;
  next->prev = last;
}
```

这个函数的功能是让原链表的最后一个元素的 next 指向 next 参数，让原链表的第一个元素的 prev 指向 prev 参数，list 这个表头就把架空了，通过它能访问链表，但链表不能访问它，已经链不起来了。

```
static inline void list_splice(const struct list_head *list,
        struct list_head *head)
{
  if (!list_empty(list))
    __list_splice(list, head, head->next);
}
```

这个函数其实就是把 head 链表加入到了 list 链表中来，从中可以可以看出，head 是加到了原 list 链表的头部。

```
static inline void list_splice_tail(struct list_head *list,
        struct list_head *head)
{
  if (!list_empty(list))
    __list_splice(list, head->prev, head);
}
```

这个函数与上面那个很类似，只不过是 head 加到了原 list 链表的尾部。

```
static inline void list_splice_init(struct list_head *list,
                struct list_head *head)
{
  if (!list_empty(list)) {
    __list_splice(list, head, head->next);
    INIT_LIST_HEAD(list);
  }
}
```

这个函数与前面的 list_splice 一样，只不过多了一步，就是把 list 这个节点重新进行了一下初始化，初始化成一个可以使用的全新的节点。

```
static inline void list_splice_tail_init(struct list_head *list,
            struct list_head *head)
{
  if (!list_empty(list)) {
    __list_splice(list, head->prev, head);
    INIT_LIST_HEAD(list);
  }
}
```

这个函数与前面的 list_splice_tail 功能一样，只不过多了一步，就是把 list 这个节点初始化了一下。

```
#define list_entry(ptr, type, member) \
  container_of(ptr, type, member)
```

这里有一个关于 container_of 的用法，我在之前的一篇博客里已经写了这个函数的实现方式与用法，不了解的可以去参考一下。这个函数的作用就是取得链表所在的结构的整个结构。

```
#define list_first_entry(ptr, type, member) \
  list_entry((ptr)->next, type, member)
```

获得 ptr 所在 entry 的下一个 entry。head 的下一个 entry 就是第一个 entry，所以在函数名里，有一个 first 字样。

```
#define list_for_each(pos, head) \
  for (pos = (head)->next; prefetch(pos->next), pos != (head); \
                pos = pos->next)
```

这个函数，定义了一个 for 循环，来不停地访问 pos 所指向的 entry，pos 的初始值设置成了 head->next，也即链表的头一个元素。注意，这个 for 循环后面没有跟分号，所以这个函数后面，还得接上每次 for 都要执行的语句，如果不止一行，就要加大括号。还有一个非常重要的地方，那就是这个函数的定义里，有一个 prefetch 指令，这个指令是把数据从内存或是磁盘上预先取到高速缓存中，这样能快速的遍历整个链表。关于这个函数可以参考 gcc 文档里的详细描述。与这个函数相对应的，还有一个类似的函数定义，为：

```
#define __list_for_each(pos, head) \
  for (pos = (head)->next; pos != (head); pos = pos->next)
```

这两个函数的唯一区别，就是这个函数定义里，没有使用 prefetch 这样的优化指令，那么这个函数就建议使用在短链表里，比如说大多数情况下，只有零个或是一个元素。因为链表长了，消耗的存储就多，那么所需要的数据被系统从高速缓存置换到内存或是磁盘上的机率就要大，所以长链表的遍历，就需要使用带 prefetch 指令的，短的就不需要了。

```
#define list_for_each_prev(pos, head) \
  for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
          pos = pos->prev)
```

这个就很容易理解啦，往前遍历，同时使用 prefetch 指令。

```
#define list_for_each_safe(pos, n, head) \
  for (pos = (head)->next, n = pos->next; pos != (head); \
    pos = n, n = pos->next)
#define list_for_each_safe(pos, n, head) \
  for (pos = (head)->next, n = pos->next; pos != (head); \
    pos = n, n = pos->next)
```

与下面这个函数

```
#define list_for_each_prev_safe(pos, n, head) \
  for (pos = (head)->prev, n = pos->prev; \
       prefetch(pos->prev), pos != (head); \
       pos = n, n = pos->prev)
```

类似，这两个函数里面，把当前元素的下一个元素给取出来了，这样的作用，就是保证，你在遍历这个链表的时候，把当前的这个链表里的元素删了，这个链表后面还能接得上，呵呵，其它方面的，我就不用多说了，看函数定义就可以啦。

```
#define list_for_each_entry(pos, head, member)        \
  for (pos = list_entry((head)->next, typeof(*pos), member);  \
       prefetch(pos->member.next), &pos->member != (head);    \
       pos = list_entry(pos->member.next, typeof(*pos), member))
```

与下面这个函数

```
#define list_for_each_entry_reverse(pos, head, member)      \
  for (pos = list_entry((head)->prev, typeof(*pos), member);  \
       prefetch(pos->member.prev), &pos->member != (head);    \
       pos = list_entry(pos->member.prev, typeof(*pos), member))
```

类似，这两个函数也是遍历链表里的元素的。一个往前，一个往后。这里也使用了 prefetch 指令来提前装载内存数据到高速缓存。从这个函数的实现就可以看出，member 这个参数，一定得是链表所在结构的链表的名字。

```
#define list_prepare_entry(pos, head, member) \
  ((pos) ? : list_entry(head, typeof(*pos), member))
```

判断 pos 这个类型是否为空，如果不是为空，则返回这个类型指针的，而且指针是指向 head 元素所在的结构的，这个并不是第一个元素，而是链表的头，链表的头是空的，并不放数据。

```
#define list_for_each_entry_continue(pos, head, member)      \
  for (pos = list_entry(pos->member.next, typeof(*pos), member);  \
           prefetch(pos->member.next), &pos->member != (head);  \
           pos = list_entry(pos->member.next, typeof(*pos), member))
```

与下面这个函数

```
#define list_for_each_entry_continue_reverse(pos, head, member)    \
  for (pos = list_entry(pos->member.prev, typeof(*pos), member);  \
           prefetch(pos->member.prev), &pos->member != (head);  \
           pos = list_entry(pos->member.prev, typeof(*pos), member))
```

类似，这两个函数其实与上面的一组函数几乎一样，只不过这里的遍历，是从 pos 的下一个元素开始的，并不是从 head 的下一个元素开始的，其它的都与上面的一样。

```
#define list_for_each_entry_from(pos, head, member)        \
  for (; prefetch(pos->member.next), &pos->member != (head);  \
           pos = list_entry(pos->member.next, typeof(*pos), member))
```

从 pos 当前的位置开始遍历

```
#define list_for_each_entry_safe(pos, n, head, member)      \
  for (pos = list_entry((head)->next, typeof(*pos), member),  \
    n = list_entry(pos->member.next, typeof(*pos), member);  \
           &pos->member != (head);            \
           pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

与

```
#define list_for_each_entry_safe_continue(pos, n, head, member)      \
  for (pos = list_entry(pos->member.next, typeof(*pos), member),      \
    n = list_entry(pos->member.next, typeof(*pos), member);    \
           &pos->member != (head);            \
           pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

与

```
#define list_for_each_entry_safe_from(pos, n, head, member)        \
  for (n = list_entry(pos->member.next, typeof(*pos), member);    \
           &pos->member != (head);            \
           pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

还有

```
#define list_for_each_entry_safe_reverse(pos, n, head, member)    \
  for (pos = list_entry((head)->prev, typeof(*pos), member),  \
    n = list_entry(pos->member.prev, typeof(*pos), member);  \
          &pos->member != (head);            \
          pos = n, n = list_entry(n->member.prev, typeof(*n), member))
```

都与各自的非 safe 版本的功能几乎是一样的，只不过，由于 safe 版本的，增加了对当前元素的后面一个元素的保存，所以如果涉及到要删除当时元素的操作时，后面的链表也可以接上，这个在之前的函数实现里也已经分析过了。

这个 list 实现的后面有一个单链表的实现，叫 hlist，按照程序中的注释，这个单链表一般是用在哈希表里的，不过这个链表没有哈希链表里 O(1)的访问效率。

我一开始看关于 hlist 的实现代码后，有一个非常不解的疑问，那就是为什么是单链表，还有一个 pprev 域呢，而且还是一个 struct hlist_node **类型的。其实经过后面的代码阅读，我才知道了，原来这里的 pprev 并不是指向单链表中当前元素的上一个元素，而是指向它自己。函数通过对 pprev 这个指针的判断，来判断这个节点是否是哈希的，具体的可以看看 hlist_unhashed 这个函数的实现。

而且在 hlist 的实现中，在删除节点以后，旧节点的 next 与 pprev 域都被置成了 LIST_POISON1 与 LIST_POISON2，并不是简单的设置成 NULL，这样做可以达到一个目的，就是当程序访问已经被删除的节点以后，通过程序打印出来的 OOPS 错误信息，可以快速地定位出错的位置，这比直接设成 NULL 要有效得多。这个技术点，我在之前的某篇文章里有讲过，这里就不再多说了，呵呵。

关于 hlist 的函数，我不想再做分析了，因为把关键的 pprev 的作用，还有前面的双链表的知识都掌握了的话，弄懂单链表的实现方法，弄懂它的用法，还是很容易的。如果对单链表有兴趣的朋友，可以自己试着分析一下，不难的。

对 linux 内核中 compiler.h 文件的分析

所有的内核代码，基本都包含了 linux\compile.h 这个文件，所以它是基础，打算先分析这个文件里的代码看看，有空再分析分析其它的代码。

首先印入眼帘的是对__ASSEMBLY__这个宏的判断，这个变量实际是在编译汇编代码的时候，由编译器使用-D 这样的参数加进去的，AFLAGS 这个变量也定义了这个变量，gcc 会把这个宏定义为 1。用在这里，是因为汇编代码里，不会用到类似于__user 这样的属性（关于__user 这样的属性是怎么回子事，本文后面会提到），因为这样的属性是在定义函数的时候加的，这样避免不必要的在编译汇编代码时候的引用。

接下来是一个对__CHECKER__这个宏的判断，这里需要讲的东西比较多。

当编译内核代码的时候，使用 make C=1 或 C=2 的时候，会调用一个叫 Sparse 的工具，这个工具对内核代码进行检查，怎么检查呢，就是靠对那些声明过 Sparse 这个工具所能识别的特性的内核函数或是变量进行检查。在调用 Sparse 这个工具的同时，在 Sparse 代码里，会加上#define __CHECKER__ 1 的字样。换句话说，就是，如果使用 Sparse 对代码进行检查，那么内核代码就会定义__CHECKER__宏，否则就不定义。

所以这里就能看出来，类似于__attribute__((noderef, address_space(1)))这样的属性就是 Sparse 这个工具所能识别的了。

那么这些个属性是干什么用的呢，我一个个做介绍。

这样的属性说明，有一部分在 gcc 的文档里还没有加进去，至少我在 gcc 4.3.2 的特性里没有看到，网上有哥们问类似的问题，Greg 对他进行了解答，然后他对 Greg 抱怨文档的事，Greg 对他说，他有时间抱怨的话，还不如自己来更新文档。他不能对一个免费工具的文档有如此之高的要求，除非他付费。

```
# define __user    __attribute__((noderef, address_space(1)))
```

__user 这个特性，即__attribute__((noderef, address_space(1)))，是用来修饰一个变量的，这个变量必须是非解除参考（no dereference）的，即这个变量地址必须是有效的，而且变量所在的地址空间必须是 1，即用户程序空间的。

这里把程序空间分成了 3 个部分，0 表示 normal space，即普通地址空间，对内核代码来说，当然就是内核空间地址了。1 表示用户地址空间，这个不用多讲，还有一个 2，表示是设备地址映射空间，例如硬件设备的寄存器在内核里所映射的地址空间。

所以在内核函数里，有一个 copy_to_user 的函数，函数的参数定义就使用了这种方式。当然，这种特性检查，只有当机器上安装了 Sparse 这个工具，而且进行了编译的时候调用，才能起作用的。

```
# define __kernel  /* default address space */
```

根据定义，就是默认的地址空间，即 0，我想定义成__attribute__((noderef, address_space(0)))也是没有问题的。

```
# define __safe    __attribute__((safe))
```

这个定义在 sparse 里也有，内核代码是在 2.6.6-rc1 版本变到 2.6.6-rc2 的时候被 Linus 加入的，经过我的艰苦的查找，终于查找到原因了，知道了为什么 Linus 要加入这个定义，原因是这样的：

有人发现在代码编译的时候，编译器对变量的检查有些苛刻，导致代码在编译的时候老是出问题（我这里没有去检查是编译不通过还是有警告信息，因为现在的编译器已经不是当年的编译器了，代码也不是当年的代码）。比如说这样一个例子，

```
int test( struct a * a, struct b * b, struct c * c ) {
  return a->a + b->b + c->c;
}
```

这个编译的时候会有问题，因为没有检查参数是否为空，就直接进行调用。但是呢，在内核里，有好多函数，当它们被调用的时候，这些个参数必定不为空，所以根本用不着去对这些个参数进行非空的检查，所以呢，就增加了一个__safe 的属性，如果这样声明变量，

```
int test( struct a * __safe a, struct b * __safe b, struct c * __safe c ) {
  return a->a + b->b + c->c;
}
```

编译就没有问题了。

不过我在现在的代码里没有发现有使用__safe 这个定义的地方，不知道是不是编译器现在已经支持这种特殊的情况了，所以就不用再加这样的代码了。

```
# define __force  __attribute__((force))
```

表示所定义的变量类型是可以做强制类型转换的，在进行 Sparse 分析的时候，是不用报告警信息的。

```
# define __nocast  __attribute__((nocast))
```

这里表示这个变量的参数类型与实际参数类型一定得对得上才行，要不就在 Sparse 的时候生产告警信息。

```
# define __iomem  __attribute__((noderef, address_space(2)))
```

这个定义与__user, __user 是一样的，只不过这里的变量地址是需要在设备地址映射空间的。

```
# define __acquires(x)  __attribute__((context(x,0,1)))
# define __releases(x)  __attribute__((context(x,1,0)))
```

这是一对相互关联的函数定义，第一句表示参数 x 在执行之前，引用计数必须为 0，执行后，引用计数必须为 1，第二句则正好相反，这个定义是用在修饰函数定义的变量的。

```
# define __acquire(x)  __context__(x,1)
# define __release(x)  __context__(x,-1)
```

这是一对相互关联的函数定义，第一句表示要增加变量 x 的计数，增加量为 1，第二句则正好相反，这个是用来函数执行的过程中。

以上四句如果在代码中出现了不平衡的状况，那么在 Sparse 的检测中就会报警。当然，Sparse 的检测只是一个手段，而且是静态检查代码的手段，所以它的帮助有限，有可能把正确的认为是错误的而发出告警。要是对以上四句的意思还是不太了解的话，请在源代码里搜一下相关符号的用法就能知道了。这第一组与第二组，在本质上，是没什么区别的，只是使用的位置上，有所区别罢了。

```
# define __cond_lock(x,c)  ((c) ? ({ __acquire(x); 1; }) : 0)
```

这句话的意思就是条件锁。当 c 这个值不为 0 时，则让计数值加 1，并返回值为 1。不过这里我有一个疑问，就是在这里，有一个__cond_lock 定义，但没有定义相应的__cond_unlock，那么在变量的释放上，就没办法做到一致。而且我查了一下关于 spin_trylock()这个函数的定义，它就用了__cond_lock，而且里面又用了_spin_trylock 函数，在_spin_trylock 函数里，再经过几次调用，就会使用到__acquire 函数，这样的话，相当于一个操作，就进行了两次计算，会导致 Sparse 的检测出现告警信息，经过我写代码进行实验，验证了我的判断，确实是会出现告警信息，如果我写两遍 unlock 指令，就没有告警信息了，但这是与程序的运行是不一致的。

```
extern void __chk_user_ptr(const volatile void __user *);
extern void __chk_io_ptr(const volatile void __iomem *);
```

这两句比较有意思。这里只是定义了函数，但是代码里没有函数的实现。这样做的目的，就是在进行 Sparse 的时候，让 Sparse 给代码做必要的参数类型检查，在实际的编译过程中，并不需要这两个函数的实现。

```
#define notrace __attribute__((no_instrument_function))
```

这一句，是定义了一个属性，这个属性可以用来修饰一个函数，指定这个函数不被跟踪。那么这个属性到底是怎么回子事呢？原来，在 gcc 编译器里面，实现了一个非常强大的功能，如果在编译的时候把一个相应的选择项打开，那么就可以在执行完程序的时候，用一些工具来显示整个函数被调用的过程，这样就不需要让程序员手动在所有的函数里一点点添加能显示函数被调用过程的语句，这样耗时耗力，还容易出错。那么对应在应用程序方面，可以使用 Graphviz 这个工具来进行显示，至于使用说明与软件实现的原理可以自己在网上查一查，很容易查到。对应于内核，因为内核一直是在运行阶段，所以就不能使用这套东西了，内核是在自己的内部实现了一个 ftrace 的机制，编译内核的时候，如果打开这个选项，那么通过挂载一个 debugfs 的文件系统来进行相应内容的显示，具体的操作步骤，可以参看内核源码所带的文档。那上面说了这么多，与 notrace 这个属性有什么关系呢？因为在进行函数调用流程的显示过程中，是使用了两个特殊的函数的，当函数被调用

与函数被执行完返回之前，都会分别调用这两个特别的函数。如果不把这两个函数的函数指定为不被跟踪的属性，那么整个跟踪的过程就会陷入一个无限循环当中。

```
#define likely(x)  __builtin_expect(!!(x), 1)
#define unlikely(x)  __builtin_expect(!!(x), 0)
```

这两句是一对对应关系。__builtin_expect(expr, c)这个函数是新版 gcc 支持的，它是用来作代码优化的，用来告诉编译器，expr 的期，非常有可能是 c，这样在 gcc 生成对应的汇编代码的时候，会把相应的可能执行的代码都放在一起，这样能少执行代码的跳转。为什么这样能提高 CPU 的执行效率呢？因为 CPU 在执行的时候，都是有预先取指令的机制的，把将要执行的指令取出一部分出来准备执行。CPU 不知道程序的逻辑，所以都是从可程序程序里挨着取的，如果这个时候，能不做跳转，则 CPU 预先取出的指令都可以接着使用，反之，则预先取出来的指令都是没有用的。还有个问题是需要注意的，在__builtin_expect 的定义中，以前的版本是没有!!这个符号的，这个符号的作用其实就是负负得正，为什么要这样做呢？就是为了保证非零的 x 的值，后来都为 1，如果为零的 0 值，后来都为 0，仅此而已。

```
#ifndef barrier
# define barrier() __memory_barrier()
#endif
```

这里表示如果没有定义 barrier 函数，则定义 barrier()函数为__memory_barrier()。但在内核代码里，是会包含 compiler-gcc.h 这个文件的，所以在这个文件里，定义 barrier()为__asm__ __volatile__(""：：:"memory")。barrier 翻译成中文就是屏障的意思，在这里，为什么要一个屏障呢？这是因为 CPU 在执行的过程中，为了优化指令，可能会对部分指令以它自己认为最优的方式进行执行，这个执行的顺序并不一定是按照程序在源码内写的顺序。编译器也有可能在生成二进制指令的时候，也进行一些优化。这样就有可能在多 CPU，多线程或是互斥锁的执行中遇到问题。那么这个内存屏障可以看作是一条线，内存屏障用在这里，就是为了保证屏障以上的操作，不会影响到屏障以下的操作。然后再看看这个屏障怎么实现的。__asm__表示后面的东西都是汇编指令，当然，这是一种在 C 语言中嵌入汇编的方法，语法有其特殊性，我在这里只讲跟这条指令有关的。__volatile__表示不对此处的汇编指令做优化，这样就会保证这里代码的正确性。""表示这里是个空指令，那么既然是空指令，则所对应的指令所需要的输入与输出都没有。在 gcc 中规定，如果以这种方式嵌入汇编，如果输出没有，则需要两个冒号来代替输出操作数的位置，所以需要加两个::，这时的指令就为""：:。然后再加上为分隔输入而加入的冒号，再加上空的输入，即为""：：:。后面的 memory 是 gcc 中的一个特殊的语法，加上它，gcc 编译器则会产生一个动作，这个动作使 gcc 不保留在寄存器内内存的值，并且对相应的内存不会做存储与加载的优化处理，这个动作不产生额外的代码，这个行为是由 gcc 编译器来保证完成的。如果对这部分有更大的兴趣，可以考察 gcc 的帮助文档与内核中一篇名为 memory-barriers.txt 的文章。

```
#ifndef RELOC_HIDE
# define RELOC_HIDE(ptr, off)          \
```

```
  ({ unsigned long __ptr;          \
     __ptr = (unsigned long) (ptr);       \
     (typeof(ptr)) (__ptr + (off)); })
#endif
```

这个没有什么太多值得讲的，也能看明白，虽然不知道具体用在哪里，所以留做以后遇到了再说吧。

接下来好多定义都没有实现，可以看一看注释就知道了，所以这里就不多说了。唉，不过再插一句，__deprecated 属性的实现是为 deprecated。

```
#define noinline_for_stack noinline
#ifndef __always_inline
#define __always_inline inline
#endif
```

这里 noinline 与 inline 属性是两个对立的属性，从词面的意思就非常好理解了。

```
#ifndef __cold
#define __cold
#endif
```

从注释中就可以看出来，如果一个函数的属性为__cold，那么编译器就会认为这个函数几乎是不可能被调用的，在进行代码优化的时候，就会考虑到这一点。不过我没有看到在 gcc 里支持这个属性的说明。

```
#ifndef __section
# define __section(S) __attribute__ ((__section__(#S)))
#endif
```

这个比较容易理解了，用来修饰一个函数是放在哪个区域里的，不使用编译器默认的方式。这个区域的名字由定义者自己取，格式就是__section__加上用户输入的参数。

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
```

这个函数的定义很有意思，它就是访问这个 x 参数所对应的东西一次，它是这样做的：先取得这个 x 的地址，然后把这个地址进行变换，转换成一个指向这个地址类型的指针，然后再取得这个指针所指向的内容。这样就达到了访问一次的目的，哈哈。

真不容易，终于把这个东西写完了，仅仅几十行的代码，里面所包含的知识真的是异常丰富，通过分析这个头文件，我自己学得了不少东西，不敢独享，拿出来给与兴趣的朋友一同分享。

linux 中关于函数__stringify(x)

在 linux 中，有一个很有意思的函数，叫__stringify，这个函数的功能叫做 stringification，没有查到它合适的中文翻译，我把它叫做参数“字符串化”。

它的功能就是把参数 x 转换成一个字符串，这个函数的实现是由两部分组成的，如下：

```
#define __stringify_1(x)  #x
#define __stringify(x)    __stringify_1(x)
```

这样写有什么好处呢？为什么不直接写成

```
#define __stringify(x)  #x
```

这个形式呢？

打个比方，如果有这样一个定义

#define FOO bar

那么如果是情况二的话，则会这样翻译

```
__stringify(FOO)
  ->FOO
```

如果是情况一呢，就会这样翻译

```
__stringify(FOO)
  ->__stringify_1(bar)
  ->bar
```

所以，这样定义的话，这个函数的使用，就可以使用表达式了，符合一般函数的调用习惯。

GCC __attribute__ 选项

2012 年 7 月 6 日 星期五
10:52

__attribute__ 可以用来设置 函数属性, 变量属性 和 类型属性, 只与声明一起使用. 即使函数是定义在同一文件内, 也需要额外提供声明才能生效. 写法为 __attribute__((arg1, arg2, arg3…)), 这里列举一些用比较常用的属性:

1. format

format 被用来描述函数的参数形式, 支持的参数形式
有 `printf`, `scanf`, `strftime` 和 `strfmon`, 比较常用的是 printf 形式, format 的语法为:
__attribute__((format(archetype, string-index, first-to-check)))
archetype 为参数形式, string-index 是指函数的格式化字符串在所有参数中所处的位置, first-to-check 是指第一个格式化参数的位置, 例如

```
1 void myprint(const char *fmt, ...) __attribute__((format(printf,
  1, 2)));
```

以上声明表示 myprint() 函数的第一个参数为格式化字符串, 第二个及往后的都是格式化参数, 这样以下调用在 gcc -Wformat 选项中可能会产生警告:

```
1 myprint("%s", 6);   // warning: format '%s' expects type 'char
2 *', but argument 2 has type 'int'
3 myprint("%d", 7);   // ok
4 myprint("%d%s%d", 7, 3); // test.c:10: warning: format '%s'
  expects type 'char *', but argument 3 has type 'int'
                          // test.c:10: warning: too few arguments
  for format
```

2. const

const 属性告诉编译器该函数调用一次后缓存返回结果,以后的调用直接使用缓存的值

3. noreturn

例如以下代码使用-Wall 选项编译时会产生警告: warning: control reaches end of non-void function

```
1 extern void myexit{exit();}
2 int test(int n) {
3     if ( n > 0 ) {
4         myexit();
5     }  // 控制逻辑分支没有返回值
6     else
7         return 0;
8 }
```

而将 myexit()的声明改为:

```
1 extern void myexit(exit()) __attribute__((noreturn));
```

则不会产生上面的警告了.

4. aligned

设置类型所使用的对齐方式,

```
1 #include
2
```

```
 3 struct a {
 4     char b;
 5     short c;
 6 };
 7
 8 struct a4 {
 9     char b;
10     short c;
11 } __attribute__((aligned(4)));
12
13 struct a8 {
14     char b;
15     short c;
16 } __attribute__((aligned(8)));
17
18 struct ap {
19     char b;
20     short c;
21 } __attribute__((packed));
22
23 int main() {
24     printf("sizeof(char) = %dn",sizeof(char));
25     printf("sizeof(short) = %dn",sizeof(short));
26     printf("sizeof(a) = %dn",sizeof(struct a));
27     printf("sizeof(a4) = %dn",sizeof(struct a4));
28     printf("sizeof(a8) = %dn",sizeof(struct a8));
29     printf("sizeof(ap) = %dn",sizeof(struct ap));
30 }
```

运行结果为

```
sizeof(char) = 1
sizeof(short) = 2
sizeof(a) = 4
sizeof(a4) = 4
sizeof(a8) = 8
sizeof(ap) = 3
```

5. no_instrument_function

关于这个参数的使用首先要解释一下 gcc 的 -finstrument-functions 选项，当 GCC 使用这个选项编译代码的时候会在每一个用户自定义函数中添加两个函数调用:

```
void __cyg_profile_func_enter(void *this, void *callsite);
void __cyg_profile_func_exit(void *this, void *callsite);
```

这两个函数是在 glibc 内部声明的，可以由用户自己定义实现，__cyg_profile_func_enter() 在进入函数的时候调用，void __cyg_profile_func_exit() 在函数退出的时候调用. 第一个参数 this 指向当前函数地址，第二个参数 callsite 指向上一级函数地址,举个例子来说明一下:

```
 1 #include
 2
 3 #define debug_print(fmt, args...)
 4     do {
 5         fprintf(stderr, fmt, ##args);
 6     }while(0)
 7
 8 extern "C" {
 9     void __cyg_profile_func_enter(void* callee, void* callsite)
10         __attribute__((no_instrument_function));
```

```
11      void __cyg_profile_func_enter(void* callee, void* callsite)
12  {
13          debug_print("Entering %p in %pn", callee, callsite);
14      }
15      void __cyg_profile_func_exit(void* callee, void* callsite)
16          __attribute__((no_instrument_function));
17      void __cyg_profile_func_exit(void* callee, void* callsite) {
18          debug_print("Exiting %p in %pn", callee, callsite);
19      }
20
21      void foo() {
22          printf("foo()n");
23      }
24  }
25
26  main() {
27      foo();
28      return 0;
    }
```

编译运行:

```
1  $ g++ t.cc  -finstrument-functions -g
2  $ ./a.out
3  Entering 0x8048690 in 0xb764ec76
4  Entering 0x804862a in 0x80486b3
5  foo()
6  Exiting 0x804862a in 0x80486b3
7  Exiting 0x8048690 in 0xb764ec76
```

这里输出的都是函数地址信息,如果想定位到函数代码,可以借助 addr2line 工具:

```
1  $ addr2line -e a.out -f -s 0x804862a 0x80486b3
2  foo
3  t.cc:22
4  main
5  t.cc:29
```

最后回到 __attribute__((no_instrument_function)), 这个选项的作用就是用来禁止编译器向指定的函数内部添加__cyg_profile_func_enter() 和 __cyg_profile_func_exit()调用代码, 例如用户可能会在__cyg_profile_func_enter 函数中调用自定义的函数, 这些被调用的自定义函数声明中都要加上__attribute__((no_instrument_function))属性, 避免产生无限递归的调用. 另外也可以通过 GCC 的以下两个选项来实现同样的作用:

`-finstrument-functions-exclude-file-list=file1,file2,... #屏蔽 file1,file2 中的函数`

`-finstrument-functions-exclude-function-list=func1,func2,... #屏蔽 func1,func2 函数`

另外可以参考一篇 developerWorks 的文章，使用 -finstrument-functions 选项结合 addr2line、graphviz 实现函数调用可视化

6. `deprecated`

设置了这一属性的函数/变量/类型在代码中被使用的时候会使编译器产生一条警告, 例如

```
1  void foo() __attribute__((deprecated));
2
3  void foo(){}
4
5  int main() {
6      foo();
```

```
7     return 0;
8 }
```

```
1 $ gcc test.c
2 test.c: In function ‘main’:
3 test.c:6: warning: ‘foo’ is deprecated (declared at test.c:3)
```

这个选项可以用来在检验代码升级的时候旧版本的代码是否都已经被移除.

linux ELF 文件格式

ELF 文件类型

ELF (Executable Linkable Format) 是 linux 下的可执行文件格式，与 windows 下的 PE (Portable Executable) 格式一样，都是 COFF (Common File Format)文件格式的变种。在 linux 下除了可执行文件，编译过程中产生的目标文件（.o 文件），动态链接文件(.so 文件)，静态链接库文件（.a 文件），核心转储文件（Core Dump File）都按照 ELF 格式存储。查看 ELF 文件类型可以用 file 命令

```
1 $ file /lib/libc-2.11.2.so
2 /lib/libc-2.11.2.so: ELF 32-bit LSB shared object, Intel
  80386, version 1 (SYSV), dynamically linked (uses shared
  libs), for GNU/Linux 2.6.18, stripped
```

ELF 文件结构

ELF 文件结构在 /usr/include/elf.h 中有完整定义

一些有用的命令：

```
file elf_ile 输出 ELF 各个段的 size
readelf [options] elf_file
       -S 输出 section header
       -t 输出符号表 symbol table
       -h 输出 ELF 文件头信息
objdump [options] elf_file
       -h 输出文件基本 ELF 信息
       -d 反汇编代码段
       -s 输出完整内容
       -r 查看重定位表
       -D 反汇编所有段内容
       -S 输出内容包含源代码（需要 gcc -g 参数支持）
```


ELF 结构中比较重要的几个段：

```
.data 数据段，存放已经初始化的全局/静态变量
.bss(Block Started by Symbol) 存放未初始化的全局变量和静态变量，因为这些变量
    在程序加载的时候都会被初始化为零，所以不需要存放实际的数据，只需要预留位置
    就可以了。
.text 代码段，存放源代码编译后的机器指令
.rodata 只读数据段，存放只读变量
.symtab(Symbol Table) 符号表
.strtab(String Table) 字符串表
```

.plt(Procedure Linkage Table) 动态链接跳转表
.got(Global Offset Table) 动态链接全局入口表

动态链接和静态链接

在静态链接过程中，编译器将需要重定位的符号写在 ELF 结构中的重定位表(Relocation Table)内，之后链接器(Linker)分析重定位表，从全局符号表中找到相应符号的地址，完成重定位
如果希望某个代码文件生成的对象文件可以被动态的链接，需要在编译时给 GCC 指定 -fPIC 参数，PIC（Position Independent Code）即位置无关代码

```
1 /***** echo.c *******/
2 #include <stdio.h>
3 extern int global_variable;
4 int echo(){
5     printf("%dn",global_variable);
6     return 0;
7 }
```

```
1 /******* main.c *******/
2 #include <stdio.h>
3 extern int echo();
4 int global_variable = 0;
5 int main() {
6     echo();
7     return 0;
8 }
```



```
1  $ g++ -fPIC -shared echo.cc -o libecho.so
2  $ g++ -c main.cc -o main.o
3  $ g++ main.o -o dynamic -lecho -L.
4  $ g++ main.cc echo.cc -o static
5
6  $ objdump -d main.o
7  00000000 <main>:
8  0:  55                      push   %ebp
9  1:  89 e5                   mov    %esp,%ebp
10 3:  83 e4 f0                and    $0xfffffff0,%esp
11 6:  e8 fc ff ff ff          call   7 <main+0x7>
12 b:  b8 00 00 00 00          mov    $0x0,%eax
13 10: 89 ec                   mov    %ebp,%esp
14 12: 5d                      pop    %ebp
15 13: c3                      ret
16
17 $ objdump -d dynamic
18 08048584 <main>:
19 8048584: 55                      push   %ebp
20 8048585: 89 e5                   mov    %esp,%ebp
21 8048587: 83 e4 f0                and    $0xfffffff0,%esp
22 804858a: e8 11 ff ff ff          call   80484a0 <echo@plt>
23 804858f: b8 00 00 00 00          mov    $0x0,%eax
24 8048594: 89 ec                   mov    %ebp,%esp
25 8048596: 5d                      pop    %ebp
26 8048597: c3                      ret
27 8048598: 90                      nop
28
```

```
$ objdump -d static
080484c8 <main>:
80484c8: 55                      push   %ebp
80484c9: 89 e5                   mov    %esp,%ebp
80484cb: 83 e4 f0                and    $0xfffffff0,%esp
80484ce: e8 d1 ff ff ff          call   80484a4 <echo>
80484d3: b8 00 00 00 00          mov    $0x0,%eax
80484d8: 89 ec                   mov    %ebp,%esp
80484da: 5d                      pop    %ebp
80484db: c3                      ret
80484dc: 90                      nop

PIC sample

$ g++  echo.cc -shared -o libecho.so
0000051c <echo>:
 51c:   55                      push   %ebp
 51d:   89 e5                   mov    %esp,%ebp
 51f:   83 ec 18                sub    $0x18,%esp
 522:   a1 00 00 00 00          mov    0x0,%eax
 527:   89 44 24 04             mov    %eax,0x4(%esp)
 52b:   c7 04 24 94 05 00 00    movl   $0x594,(%esp)
 532:   e8 fc ff ff ff          call   533 <echo+0x17>
 537:   b8 00 00 00 00          mov    $0x0,%eax
 53c:   c9                      leave
 53d:   c3                      ret
 53e:   90                      nop
 53f:   90                      nop

$ g++  echo.cc -shared -fPIC -o libecho.so
0000052c <echo>:
 52c:   55                      push   %ebp
 52d:   89 e5                   mov    %esp,%ebp
 52f:   53                      push   %ebx
 530:   83 ec 14                sub    $0x14,%esp
 533:   e8 ef ff ff ff          call   527
<__i686.get_pc_thunk.bx>
 538:   81 c3 e4 11 00 00       add    $0x11e4,%ebx
 53e:   8b 83 fc ff ff ff       mov    -0x4(%ebx),%eax
 544:   8b 00                   mov    (%eax),%eax
 546:   89 44 24 04             mov    %eax,0x4(%esp)
 54a:   8d 83 a8 ee ff ff       lea    -0x1158(%ebx),%eax
 550:   89 04 24                mov    %eax,(%esp)
 553:   e8 f0 fe ff ff          call   448 <printf@plt>
 558:   b8 00 00 00 00          mov    $0x0,%eax
 55d:   83 c4 14                add    $0x14,%esp
 560:   5b                      pop    %ebx
 561:   5d                      pop    %ebp
```

GOT 和 PLT

```
#include

void foo() {
    printf("test 0x%06xn", 10);
    return;
```

```
 6 }
 7
 8 int main () {
 9     foo();
10     return 0;
11 }
```

编译后查看 ELF 的信息

```
1 gcc -g test.c -o test
2 objdump -S test
```

以下为部分段的内容:

```
1  080482cc :
2   80482cc:   ff 35 c8 95 04 08       pushl  0x80495c8
3   80482d2:   ff 25 cc 95 04 08       jmp    *0x80495cc
4   80482d8:   00 00                   add    %al,(%eax)
5
6  080482fc :
7   80482fc:   ff 25 d8 95 04 08       jmp    *0x80495d8
8   8048302:   68 10 00 00 00          push   $0x10
9   8048307:   e9 c0 ff ff ff          jmp    80482cc
10
11 int main () {
12  80483f0:   55                      push   %ebp
13  80483f1:   89 e5                   mov    %esp,%ebp
14  80483f3:   83 ec 08                sub    $0x8,%esp
15  80483f6:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
16     foo();
17  80483fd:   e8 ce ff ff ff          call   80483d0
18  8048402:   b8 00 00 00 00          mov    $0x0,%eax
19     return 0;
20  8048407:   83 c4 08                add    $0x8,%esp
21  804840a:   5d                      pop    %ebp
22  804840b:   c3                      ret
23
24 void foo() {
25  80483d0:   55                      push   %ebp
26  80483d1:   89 e5                   mov    %esp,%ebp
27  80483d3:   83 ec 08                sub    $0x8,%esp
28  80483d6:   8d 05 d0 84 04 08       lea    0x80484d0,%eax
29     printf("testn");
30  80483dc:   89 04 24                mov    %eax,(%esp)
31  80483df:   e8 18 ff ff ff          call   80482fc
32     return;
33  80483e4:   89 45 fc                mov    %eax,-0x4(%ebp)
34  80483e7:   83 c4 08                add    $0x8,%esp
35  80483ea:   5d                      pop    %ebp
36  80483eb:   c3                      ret
37  80483ec:   0f 1f 40 00             nopl   0x0(%eax)
```

上拉、下拉电阻

上、下拉电阻：

1、当 TTL 电路驱动 CMOS 电路时，如果电路输出的高电平低于 CMOS 电路的最低高电平（一般为 3.5V）， 这时就需要在 TTL 的输出端接上拉电阻，以提高输出高电平的值。

2、OC 门电路必须使用上拉电阻，以提高输出的高电平值。
3、为增强输出引脚的驱动能力，有的单片机管脚上也常使用上拉电阻。
4、在 CMOS 芯片上，为了防止静电造成损坏，不用的管脚不能悬空，一般接上拉电阻以降低输入阻抗， 提供泄荷通路。
5、芯片的管脚加上拉电阻来提高输出电平，从而提高芯片输入信号的噪声容限，增强抗干扰能力。
6、提高总线的抗电磁干扰能力，管脚悬空就比较容易接受外界的电磁干扰。
7、长线传输中电阻不匹配容易引起反射波干扰，加上、下拉电阻是电阻匹配，有效的抑制反射波干扰。

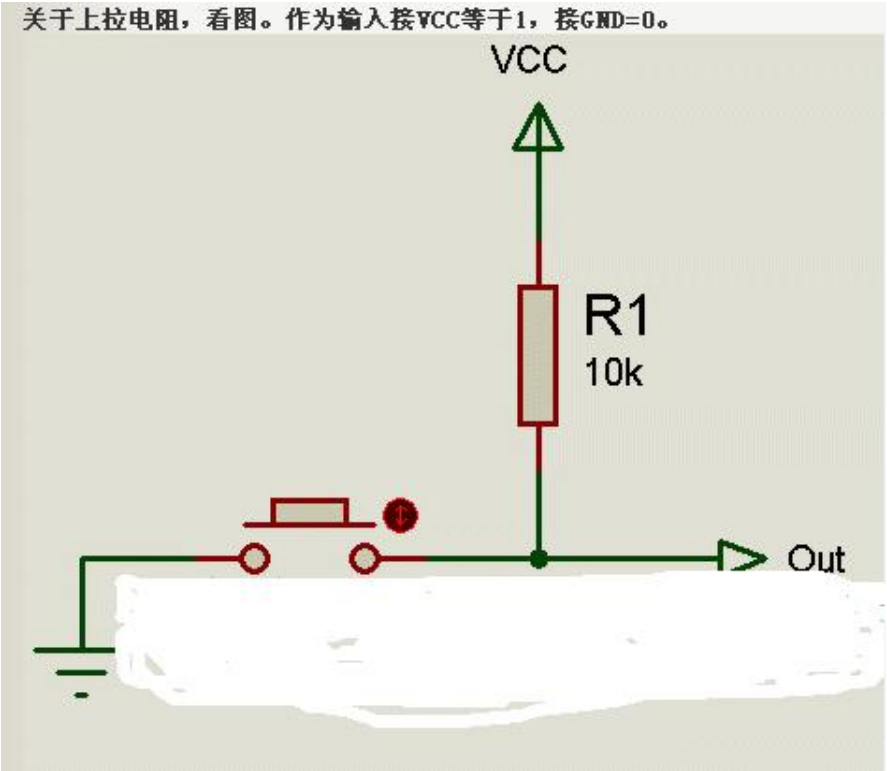
上拉电阻是用来解决总线驱动能力不足时提供电流的。一般说法是上拉增大电流，下拉电阻是用来吸收电流。

原理

在上拉电阻所连接的导线上，如果外部组件未启用，上拉电阻将“微弱地”将输入电压信号“拉高”。当外部组件未连接时，对输入端来说，外部“看上去”就是高阻抗的。这时，通过上拉电阻可以将输入端口处的电压拉高到高电平。如果外部组件启用，它将取消上拉电阻所设置的高电平。通过这样，上拉电阻可以使引脚即使在未连接外部组件的时候也能保持确定的逻辑电平。[3]

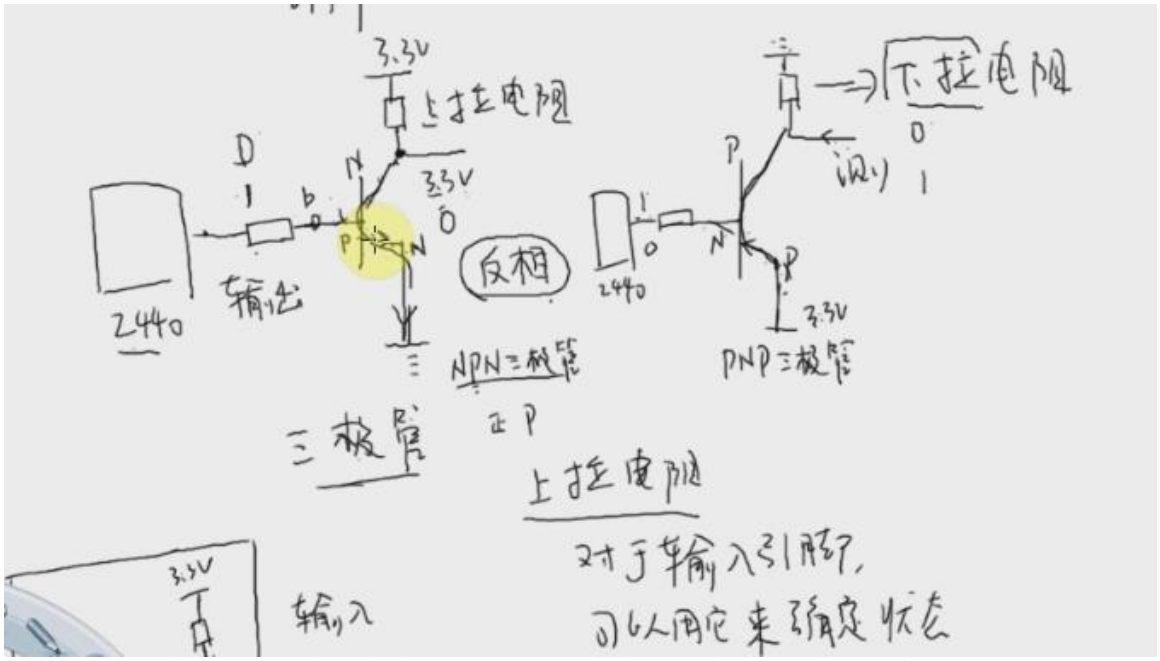
下拉电阻

同样的，一个下拉电阻（Pull-down resistor）以类似的方式工作，不过是与地（GND）连接。它可以使逻辑信号保持在接近 0 伏特的状态，即使没有活动的设备连接在其所在的引脚上。



如果按键短路（按下）电阻为零，按键按下，Out=0，当按键断开，Out=？显然当 Out 悬空输出 VCC，这可以用仪表测量，
这个 VCC 就是靠 R1“上拉”产生的，顾名思义，R1 就是上拉电阻。上拉电阻的大小，取决于输出接负载的需要，通常逻辑电路对高电平输出阻抗很大，要求输出电流很小，在上拉电阻上压降可以忽略，当然上拉电阻不能太大，否则就不能忽略了。

1，上拉电阻就是把端口连接到电源的电阻，下拉电阻就是把端口连接到地的电阻。
2，简单的说，上拉电阻的主要作用在于提高输出信号的驱动能力、确定输入信号的电平（防止干扰）等，具体的表现为：
当 TTL 电路驱动 COMS 电路时，如果 TTL 电路输出的高电平低于 COMS 电路的最低高电平（一般为 3.5V）， 这时就需要在 TTL 的输出端接上拉电阻，以提高输出高电平的值。
OC 门电路必须加上拉电阻，以提高输出的搞电平值。
为加大输出引脚的驱动能力，有的单片机管脚上也常使用上拉电阻。
在 COMS 芯片上，为了防止静电造成损坏，不用的管脚不能悬空，一般接上拉电阻产生降低输入阻抗，提供泄荷通路。



l 芯片的管脚加上拉电阻来提高输出电平，从而提高芯片输入信号的噪声容限增强抗干扰能力。

l 提高总线的抗电磁干扰能力。管脚悬空就比较容易接受外界的电磁干扰。

l 长线传输中电阻不匹配容易引起反射波干扰，加上下拉电阻是电阻匹配，有效的抑制反射波干扰。

上拉电阻阻值的选择原则包括:

l 从节约功耗及芯片的灌电流能力考虑应当足够大；电阻大，电流小。

l 从确保足够的驱动电流考虑应当足够小；电阻小，电流大。

l 对于高速电路，过大的上拉电阻可能边沿变平缓。

综合考虑以上三点，通常在 1K 到 10K 之间选取。对下拉电阻也有类似道理。

驱动中相关结构与函数的含义：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsi
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
} ? end file_operations ? ;
```


struct file_operations {
struct module *owner;
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *, fl_owner_t id);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
unsigned long);
int (*check_flags)(int);
int (*dir_notify)(struct file *filp, unsigned long arg);

```
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
};
```

1，模块所有者指针，一般初始化为 THIS_MODULES 。

```
struct module *owner;
```

2，用来修改文件当前的读写位置，返回新位置。Loff_t 为一个“长偏移量”。

3，同步读取函数：读取成功返回读取的字节数。设置为 NULL，调用时返回 -EINVAL 。

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)
```

参 1：要读的文件。
参 2：是将文件读到哪里去。
参 3：要读文件多少内容。Size_t 是 unsigned int .
参 4：用来修改文件当前的读写位置，返回新位置。Loff_t 为一个“长偏移量”。（是指这次对文件进行操作的起
始位置。）
struct file 中的 f_pos 是最后一次文件操作以后的当前读写位置.

4，异步读取操作，为 NULL 时全部通过 read 处理。

5，同步写入函数。

6，异步写入操作。

7，仅用于读取目录，对于设备文件，该字段为 NULL 。

8，判断目前是否可以对设备进行读写操作。字段为空时，设备会被认为既可读也可写。

9，向设备发送 IO 控制命令的函数，不设置入口点。返回 -ENOTTY。

10，不使用 BLK 的文件系统，将使用此种函数指针代替 ioctl 。

11，在 64 位系统上，32 位的 ioctl 调用，将使用此函数指针代替。

12，用于请求将设备内存映射到进程地址空间。如果无此方法，将访问 -ENODEV

注册 file_operations 结构体到内核：

```
int register_chrdev(unsigned int major, const char *name,
                    const struct file_operations *fops)
```

参 1：设备文件的主设备号，为“0”时让系统自动分配。这时“register_chrdev()”返回一个 int 类型的主设备号。
这个返回值要接收，因为在“出口函数”中，要注销这个注册到内核中的设备的操作结构时，要提供两个参数：一个
主设备号，一个为设备名。

参 2：设备名字，可以在“/dev”目录下看到这个设备节点名。

参 3：要注册到内核的 file_operations 结构体。

注销注册到内核中的设备：

```
int unregister_chrdev(unsigned int major, const char *name)
```

参 1：这个设备的主设备号。（主设备号代表着这类设备。）
参 2：这个设备在系统中的名字。如 /dev/ttyUSB* .

struct class
设备类 struct class 是一个设备的高级视图，它抽象出低级的实现细节。例如，驱动可以见到一个 SCSI 磁盘或者一个 ATA 磁盘，在类的级别，他们都是磁盘，类允许用户空间基于它们作什么来使用设备，而不是它们如何被连接或者它们如何工作。

```
struct class{
        const char *name;                   //类名称
        struct module *owner;               //对应模块
        struct subsystem subsys;          //对应的 subsystem;
        struct list_head children;      //class_device 链表
        struct list_head    interfaces;     //class_interface 链表
        struct semaphore    sem;                //用于同步的信号锁
        struct class_attribute *class_attrs;     //类属性
        int (*uevent)(struct class_device *dev,char **envp,int num_envp,
                        char *buffer,int buffer_size);     //事件
        void (*release)(struct class_device *dev);       //释放类设备
        void (*class_release)(struct class *class);      //释放类
}
```

```
/* class可以看成一个容器，容器包含了很多class_device，每个都对应一个具体的逻辑设备，
 * 并通过成员变量 dev 关联一个物理设备。
 */
struct class_device {
    struct list_head    node;

    struct kobject      kobj;       /* 内嵌的 kobject，用于计数。 */
    struct class        * class;    /* required 所属的类 */
    dev_t           devt;           /* dev_t, creates the sysfs "dev"  设备号 */
    struct class_device_attribute *devt_attr;
    struct class_device_attribute uevent_attr;
    struct device       * dev;      /* not necessary, but nice to have。若存在，创建到/sys/devices 相应入口的
    void            * class_data;   /* class-specific data 私有数据*/
    struct class_device *parent;    /* parent of this child device, if there is one  父设备 */
    struct attribute_group  ** groups;  /* optional groups */

    void    (*release)(struct class_device *dev);     /* 释放对应类实际设备的方法 */
    int (*uevent)(struct class_device *dev, char **envp,
                int num_envp, char *buffer, int buffer_size);
    char    class_id[BUS_ID_SIZE];  /* unique to this class 类标志*/
};
```

在/sys/class/下创建类目录：

```
struct class *class_create(struct module *owner, const char *name)
```

参 1：
参 2：

在设备类下创建具体的设备：

```
struct class_device *class_device_create(struct class *cls,
                    struct class_device *parent,
                    dev_t devt,
                    struct device *device,
                    const char *fmt, ...)
```

参 1：设备类。
参 2：父设备。没有时，直接为 NULL。
参 3：主设备号。
参 4：LINUX 中的任一设备在设备模型中都由一个 device 对象描述。可以直接用 NULL。
参 5：在/dev 目录中的设备节点名字。Mdev 应用程序会根据 class 和 类下的设备创建 /dev 下相应的设备节点。

```
int __must_check class_register(struct class *); /* 注册类 */
```

参 1：要注册的设备类。
返回值：

```
void class_unregister(struct class *);          /* 注销类 */
```

参 1：要注销的设备类。
返回值：无


```
static int release(struct inode * inode, struct file * file)
```

用“request_irq()”注册了中断后，当设备关闭时，要释放这些中断，则要在 file_operations 结构中定义一个注销的操作函数。

GCC 参数详解

GCC 参数详解
[版本]-0.13[声明]
这篇文档是我的关于 gcc 参数的笔记,我很怀念 dos 年代我用小本子,纪录所有的 dos
命令的参数.哈哈,下面的东西可能也不是很全面,我参考了很多的书,和 gcc 的帮助.
不全的原因是,有可能我还没有看到这个参数,另一种原因是,我可能还不会用它
不过,我会慢慢的补齐的.哈如果你要转在本文章请保留我的 email(pianopan@beeship.com)和文章的全面性.

[介绍]

gcc and g++分别是 gnu 的 c & c++编译器
gcc/g++在执行编译工作的时候，总共需要 4 步
1.预处理,生成.i 的文件[预处理器 cpp]
2.将预处理后的文件不转换成汇编语言,生成文件.s[编译器 egcs]
3.有汇编变为目标代码(机器代码)生成.o 的文件[汇编器 as]
4.连接目标代码,生成可执行程序[链接器 ld]

[参数详解]

```
-x language filename
```

设定文件所使用的语言,使后缀名无效,对以后的多个有效.也就是根据约定 C 语言的后缀名称是.c 的，而 C++的后缀名是.C 或者.cpp,如果你很个性，决定你的 C 代码文件的后缀名是.pig 哈哈，那你就要用这个参数,这个参数对他后面的文件名都起作用，除非到了下一个参数的使用。

可以使用的参数吗有下面的这些

```
`c', `objective-c', `c-header', `c++', `cpp-output',
`assembler', and `assembler-with-cpp'.
```

看到英文，应该可以理解的。

例子用法: gcc -x c hello.pig

-x none filename

关掉上一个选项，也就是让 gcc 根据文件名后缀，自动识别文件类型
例子用法:

```
gcc -x c hello.pig -x none hello2.c -c
```

只激活预处理,编译,和汇编,也就是他只把程序做成 obj 文件
例子用法:

```
gcc -c hello.c
```

他将生成.o 的 obj 文件

-S

只激活预处理和编译，就是指把文件编译成为汇编代码。
例子用法

```
gcc -S hello.c
```

他将生成.s 的汇编代码，你可以用文本编辑器察看

-E

只激活预处理,这个不生成文件,你需要把它重定向到一个输出文件里面.
例子用法:

```
gcc -E hello.c > pianoapan.txt
gcc -E hello.c | more
```

慢慢看吧,一个 hello word 也要与处理成 800 行的代码

-o

制定目标名称,缺省的时候,gcc 编译出来的文件是 a.out,很难听,如果你和我有同感，改掉它,哈哈
例子用法

```
gcc -o hello.exe hello.c (哦,windows用习惯了)
gcc -o hello.asm -S hello.c
```

-pipe

使用管道代替编译中临时文件,在使用非 gnu 汇编工具的时候,可能有些问题

```
gcc -pipe -o hello.exe hello.c
```


-ansi

关闭 gnu c 中与 ansi c 不兼容的特性,激活 ansi c 的专有特性(包括禁止一些 asm inline typeof 关键字,以及 UNIX,vax 等预处理宏,

-fno-asm

此选项实现 ansi 选项的功能的一部分，它禁止将 asm,inline 和 typeof 用作关键字。

```
-fno-strict-prototype
```

只对 g++起作用,使用这个选项,g++将对不带参数的函数,都认为是没有显式的对参数的个数和类型说明,而不是没有参数.
而 gcc 无论是否使用这个参数,都将对没有带参数的函数,认为城没有显式说明的类型

-fthis-is-varialble

就是向传统 c++看齐,可以使用 this 当一般变量使用.

-fcond-mismatch

允许条件表达式的第二和第三参数类型不匹配,表达式的值将为 void 类型

```
-funsigned-char
-fno-signed-char
-fsigned-char
-fno-unsigned-char
```

这四个参数是对 char 类型进行设置,决定将 char 类型设置成 unsigned char(前两个参数)或者 signed char(后两个参数)

-include file

包含某个代码,简单来说,就是便以某个文件,需要另一个文件的时候,就可以用它设定,功能就相当于在代码中使用#include<filename>
例子用法:

```
gcc hello.c -include /root/pianopan.
```

-imacros file

将 file 文件的宏,扩展到 gcc/g++的输入文件,宏定义本身并不出现在输入文件中

-Dmacro

相当于 C 语言中的#define macro

-Dmacro=defn

相当于 C 语言中的#define macro=defn

-Umacro

相当于 C 语言中的#undef macro

-undef

取消对任何非标准宏的定义

-Idir

在你是用#include"file"的时候,gcc/g++会先在当前目录查找你所制定的头文件,如果没有找到,他回到缺省的头文件目录找,如果使用-I 制定了目录,他回先在你所制定的目录查找,然后再按常规的顺序去找.
对于#include<file>,gcc/g++会到-I 制定的目录查找,查找不到,然后将到系统的缺省的头文件目录查找

-I-

就是取消前一个参数的功能,所以一般在-Idir 之后使用

-idirafter dir

在-I 的目录里面查找失败,讲到这个目录里面查找.

-iprefix prefix

-iwithprefix dir

一般一起使用,当-I 的目录查找失败,会到 prefix+dir 下查找

-nostdinc

使编译器不再系统缺省的头文件目录里面找头文件,一般和-I 联合使用,明确限定头文件的位置

-nostdin C++

规定不在 g++指定的标准路经中搜索,但仍在其他路径中搜索,.此选项在创建

libg++库使用

-C

在预处理的时候,不删除注释信息,一般和-E 使用,有时候分析程序，用这个很方便的

-M

生成文件关联的信息。包含目标文件所依赖的所有源代码
你可以用 gcc -M hello.c 来测试一下，很简单。

-MM

和上面的那个一样，但是它将忽略由#include<file>造成的依赖关系。

-MD

和-M 相同，但是输出将导入到.d 的文件里面

-MMD

和-MM 相同，但是输出将导入到.d 的文件里面

-Wa,option

此选项传递 option 给汇编程序;如果 option 中间有逗号,就将 option 分成多个选项,然后传递给会汇编程序

-Wl.option

此选项传递 option 给连接程序;如果 option 中间有逗号,就将 option 分成多个选项,然后传递给会连接程序.

-llibrary

制定编译的时候使用的库
例子用法

```
gcc -lcurses hello.c
```

使用 ncurses 库编译程序

-Ldir

制定编译的时候，搜索库的路径。比如你自己的库，可以用它制定目录，不然编译器将只在标准库的目录找。这个 dir 就是目录的名称。

```
-O0
-O1
-O2
-O3
```

编译器的优化选项的 4 个级别，-O0 表示没有优化,-O1 为缺省值，-O3 优化级别最高

-g

只是编译器，在编译的时候，产生条是信息。

-gstabs

此选项以 stabs 格式声称调试信息,但是不包括 gdb 调试信息.

-gstabs+

此选项以 stabs 格式声称调试信息,并且包含仅供 gdb 使用的额外调试信息.

-ggdb

此选项将尽可能的生成 gdb 的可以使用的调试信息.

-static

此选项将禁止使用动态库，所以，编译出来的东西，一般都很大，也不需要什么动态连接库，就可以运行.

-share

此选项将尽量使用动态库，所以生成文件比较小，但是需要系统由动态库.

-traditional

试图让编译器支持传统的 C 语言特性

[参考资料]

-Linux/UNIX 高级编程
中科红旗软件技术有限公司编著.清华大学出版社出版
-Gcc man page