附录 A　Linux 内核 API

以下函数是 Linux 内核提供给用户进行内核级程序开发可以调用的主要函数。

1．驱动程序的基本函数

类别	函数名	功能	函数形成	参数	描述
驱动程序入口和出口点	module_init	驱动程序初始化入口点	module_init (x)	x为启动时或插入模块时要运行的函数	如果在启动时就确认把这个驱动程序插入内核或以静态形成链接，则 module_init 将其初始化例程加入到“__initcall.int”代码段，否则将用 init_module 封装其初始化例程，以便该驱动程序作为模块来使用
	module_exit	驱动程序退出出口点	module_exit (x)	x为驱动程序被卸载时要运行的函数	当驱动程序是一个模块，用 rmmod 卸载一个模块时 module_exit（）将用 cleanup_module（）封装 clean-up 代码。如果驱动程序是静态地链接进内核，则 module_exit（）函数不起任何作用
原子和指针操作	atomic_read	读取原子变量	atomic_read (v)	v 为指向 atom-ic_t 类型的指针	原子地读取 v 的值。注意要保证 atomic 的有用范围只有 24 位
	atomic_set	设置原子变量	atomic_set (v, i)	v 为指向 atomic _t 类型的指针，i 为待设置的值	原子地把 v 的值设置为 i。注意要保证 atomic 的有用范围只有 24 位
	atomic_add	把整数增加到原子变量	void atomic_add (int i, atomic_t * v)	i 为要增加的值，v 为指向 atomic_t 类型的指针	原子地把 i 增加到 v。注意要保证 atomic 的有用范围只有 24 位
	atomic_sub	减原子变量的值	void atomic_sub (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针。	原子地从 v 减取 i。注意要保证 atomic 的有用范围只有 24 位。
	atomic_sub_and_test	从变量中减去值，并测试结果	int atomic_sub_and_test (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针	原子地从 v 减取 i 的值，如果结果为 0，则返回真，其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位

	atomic_inc	增加原子变量的值	void atomic_inc (atomic_t * v)	v 为指向 atomic _t 类型的指针	原子地从 v 减取 1。注意要保证 atomic 的有用范围只有 24 位

续表

类别	函数名	功能	函数形成	参数	描述
原子和指针操作	atomic_dec	减取原子变量的值	void atomic_dec (atomic_t * v)	v 为指向 ato- mic_t 类型的指针	原子地给 v 增加 1。注意要保证 atomic 的有用范围只有 24 位
	atomic_dec _and_test	减少和测试	int atomic_dec_ and_test (atomic_ t * v)	v 为指向 ato- mic_t 类型的指针	原子地给 v 减取 1，如果结果为 0，则返回真，其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位
	atomic_inc _and_test	增加和测试	int atomic_ inc _and_test (atomic _t * v)	v 为指向 ato- mic_t 类型的指针	原子地给 v 增加 1，如果结果为 0，则返回真；其他所有情况都返回假。注意要保证 atomic 的有用范围只有 24 位
	atomic_add _negative	如果结果为负数，增加并测试	int atomic_add_ negative (int i, atomic_t * v)	i 为要减取的值，v 为指向 atomic_t 类型的指针	原子地给 v 增加 i，如果结果为负数，则返回真；如果结果大于等于 0，则返回假。注意要保证 atomic 的有用范围只有 24 位
	et_unalign ed	从非对齐位置获取值	get_unaligned (ptr)	ptr 指向获取的值	这个宏应该用来访问大于单个字节的值，该值所处的位置不在按字节对齐的位置，例如从非 u16 对齐的位置检索一个 u16 的值。注意，在某些体系结构中，非对齐访问要化费较高的代价
	put_unalig ned	把值放在一个非对齐位置	put_unaligned (val, ptr)	val 为要放置的值，ptr 指向要放置的位置	这个宏用来把大于单个字节的值放置在不按字节对齐的位置，例如把一个 u16 值写到一个非 u16 对齐的位置。注意事项同上

延时、调度及定时器例程	schedule_timeout	睡眠到定时时间到	signed long sche-dule_timeout (signed long timeout)	timeout 为以 jiffies 为单位的到期时间	使当前进程睡眠，直到所设定的时间到期。如果当前进程的状态没有进行专门的设置，则睡眠时间一到该例程就立即返回。如果当前进程的状态设置为： ① TASK_UNINTERRUPTIBLE：则睡眠到期该例程返回 0 ② TASK_INTERRUPTIBLE：如果当前进程接收到一个信号，则该例程就返回，返回值取决于剩余到期时间 当该例程返回时，要确保当前进程处于 TASK_RUNNING 状态

2．双向循环链表的操作

函数名	功能	函数形成	参数	描述
list_add	增加一个新元素	void list_add (struct list_head * new, struct list_head * head)	new为要增加的新元素,head为增加以后的链表头	在指定的头元素后插入一个新元素，用于栈的操作
list_add_tail	增加一个新元素	void list_add_tail (str- uct list_head * new, struct list_head * head);	new为要增加的新元素,head为增加以前的链表头	在指定的头元素之前插入一个新元素，用于队列的操作
list_del	从链表中删除一个元素	void list_del (struct list_head * entry);	entry 为要从链表中删除的元素	
list_del_init	从链表删除一个元素，并重新初始化链表	void list_del_init (struct list_head * entry)	entry 为要从链表中删除的元素	
list_empty	测试一个链表是否为空	int list_empty (struct list_head * head)	head 为要测试的链表	
list_splice	把两个链表合并在一起	void list_splice (struct list_head * list, struct list_head * head)	list 为新加入的链表，head 为第一个链表	
list_entry	获得链表中元素的	list_entry (ptr, type, member)	ptr 为指向 list_head 的指针，type 为一个结构体，而	

	结构		member 为结构 type 中的一个域，其类型为 list_head	
list_for_each	扫描链表	list_for_each (pos, head)	pos 为指向 list_head 的指针，用于循环计数，head 为链表头	

3．基本 C 库函数

当编写驱动程序时，一般情况下不能使用 C 标准库的函数。Linux 内核也提供了与标准库函数功能相同的一些函数，但二者还是稍有差别。

类别	函数名	功能	函数形成	参数	描述
字符串转换	simple_strtol	把一个字符串转换为一个有符号长整数	long simple_strtol (con- st char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数	
	simple_strtoll	把一个字符串转换为一个有符号长长整数	long long simple_strtoll (const char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数	

续表

类别	函数名	功能	函数形成	参数	描述
字符串转换	simple_strtoul	把一个字符串转换为一个无符号长整数	long long simple_strtoul (const char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数	
	simple_strtoull	把一个字符串转换为一个无符号长长整数	long long simple_ strt- oull (const char * cp, char ** endp, unsigned int base)	cp 指向字符串的开始，endp 为指向要分析的字符串末尾处的位置，base 为要用的基数	
	vsnprintf	格式化一个字符串，并把它放在缓存中	int vsnprintf (char * buf, size_t size, const char * fmt, va_list args)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为要使用的格式化字符串，args 为格式化字符串的参数	

	snprintf	格式化一个字符串，并把它放在缓存中	int snprintf (char * buf, size_t size, const char * fmt, ...)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为格式化字符串，使用@…来对格式化字符串进行格式化，…为可变参数	
	vsprintf	格式化一个字符串，并把它放在缓存中	int vsprintf (char * buf, const char * fmt, va_list args)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为要使用的格式化字符串，args 为格式化字符串的参数	
	sprintf	格式化一个字符串，并把它放在缓存中	int sprintf (char * buf, const char * fmt, ...)	buf 为存放结果的缓冲区，size 为缓冲区的大小，fmt 为格式化字符串，使用@…来对格式化字符串进行格式化，…为可变参数	
字符串操作	strcpy	拷贝一个以 NUL 结束的字符串	char * strcpy (char * dest, const char * src)	dest 为目的字符串的位置，src 为源字符串的位置	
	strncpy	拷贝一个定长的、以 NUL 结束的字符串	char * strncpy (char * dest, const char * src, size_t count)	dest 为目的字符串的位置，src 为源字符串的位置，count 为要拷贝的最大字节数	与用户空间的 strncpy 不同，这个函数并不用 NUL 填充缓冲区，如果与源串超过 count，则结果以非 NUL 结束

续表

类别	函数名	功　能	函数形成	参　数	描　述
字符串操作	strcat	把一个以 NUL 结束的字符串添加到另一个串的末尾	char * strcat (char * dest, const char * src)	dest 为要添加的字符串，src 为源字符串	
	strncat	把一个定长的、以 NUL 结束的字符串添加到另一个串的末尾	char * strncat (char * dest, const char * src, size_t count)	dest 为要添加的字符串，src 为源字符串，count 为要拷贝的最大字节数	注意，与 strncpy，形成对照，strncat 正常结束
	strchr	在一个字符串中查找第一次出现的某个字符	char * strchr (const char * s, int c)	s 为被搜索的字符串，c 为待搜索的字符	

	strrchr	在一个字符串中查找最后一次出现的某个字符	char * strrchr (const char * s, int c)	s 为被搜索的字符串，c 为待搜索的字符	
	strlen	给出一个字符串的长度	size_t strlen (const char * s)	s 为给定的字符串	
	strnlen	给出给定长度字符串的长度	size_t strnlen (const char * s, size_t count)	s 为给定的字符串	
	strpbrk	在一个字符串中查找第一次出现的一组字符	char * strpbrk (const char * cs, const char * ct)	cs 为被搜索的字符串，ct 为待搜索的一组字符	
	strtok	把一个字符串分割为子串	char * strtok (char * s, const char * ct)	s 为被搜索的字符串，ct 为待搜索的子串	注意，一般不提倡用这个函数，而应当用 strsep
	memset	用给定的值填充内存区	void * memset (void * s, int c, size_t count)	s 为指向内存区起始的指针，c 为要填充的内容，count 为内存区的大小	I/O 空间的访问不能使用 memset，而应当使用 memset_io
	bcopy	把内存的一个区域拷贝到另一个区域	char * bcopy (const char * src, char * dest, int count)	src 为源字符串，dest 为目的字符串，而 count 为内存区的大小	注意，这个函数的功能与 memcpy 相同，这是从 BSD 遗留下来的，对 I/O 空间的访问应当用 memcpy_toio 或 memcpy_fromio

续表

类别	函数名	功能	函数形成	参数	描述
字符串操作	memcpy	把内存的一个区域拷贝到另一个区域	void * memcpy (void * dest, const void * src, size_t count)	dest 为目的字符串，src 为源字符串，而 count 为内存区的大小	对 I/O 空间的访问应当用 memcpy_toio 或 memcpy_fromio
	memmove	把内存的一个区域拷贝到另一个区域	void * memmove (void * dest, const void * src, size_t count)	dest 为目的字符串，src 为源字符串，而 count 为内存区的大小	memcpy 和 memmove 处理重叠的区域，而该函数不处理
	memcmp	比较内存的两个区域	int memcmp (const void * cs, const void * ct, size_t count)	cs 为一个内存区，ct 为另一个内存区，而 count 为内存区的大小	

	memscan	在一个内存区中查找一个字符	void * memscan (void * addr, int c, size_t size)	addr 为内存区，c 为要搜索的字符，而 size 为内存区的大小	返回 c 第一次出现的地址，如果没有找到 c，则向该内存区传递一个字节
	strstr	在以 NUL 结束的串中查找第一个出现的子串	char * strstr (const char * s1, const char * s2)	s1 为被搜索的串，s2 为待搜索的串	
	memchr	在一个内存区中查找一个字符	void * memchr (const void * s, int c, size_t n)	s 为内存区，为待搜索的字符，n 为内存的大小	返回 c 第 1 次出现的位置，如果没有找到 c，则返回空
位操作	set_bit	在位图中原子地设置某一位	void set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	这个函数是原子操作，如果不需要原子操作，则调用__set_bit 函数，nr 可以任意大，位图的大小不限于一个字
	__set_bit	在位图中设置某一位	void __set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	
	clear_bit	在位图中清某一位	void clear_bit (int nr, volatile void * addr)	nr 为要清的位，addr 为位图的起始地址	该函数是原子操作，但不具有加锁功能，如果要用于加锁目的，应当调用 smp_mb__before_clear_bit 或 smp_mb__after_clear_bit 函数，以确保任何改变在其他的处理器上是可见的
	__change_bit	在位图中改变某一位	void__change_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	与 change_bit 不同，该函数是非原子操作
	change_bit	在位图中改变某一位	void change_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	

续表

类别	函 数 名	功　能	函数形成	参　数	描　述
位操作	test_and_set_bit	设置某一位并返回该位原来的值	int test_and_set_bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	该函数是原子操作

	__test_and_set_bit	设置某一位并返回该位原来的值	int__test_and_set_ bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	该函数是非原子操作，如果这个操作的两个实例发生竞争，则一个成功而另一个失败，因此应当用一个锁来保护对某一位的多个访问
	test_and_clear_bit	清某一位，并返回原来的值	int test_and_clear_bit (int nr, volatile void * addr);	nr 为要设置的位，addr 为位图的起始地址	该函数是原子操作
	__test_and_clear_bit	清某一位，并返回原来的值	int __test_and_clear_bit (int nr, volatile void * addr);	nr 为要设置的位，addr 为位图的起始地址。	该函数为非原子操作
	test_and_change_bit	改变某一位并返回该位的新值	int test_and_change_ bit (int nr, volatile void * addr)	nr 为要设置的位，addr 为位图的起始地址	该函数为原子操作
	test_bit	确定某位是否被设置	int test_bit (int nr, const volatile void * addr)	nr 为要测试的第几位，addr 为位图的起始地址	
	find_first_zero_bit	在内存区中查找第一个值为 0 的位	int find_first_zero_ bit (void * addr, unsigned size)	addr 为内存区的起始地址，size 为要查找的最大长度	返回第一个位为 0 的位号
	find_next_zero_bit	在内存区中查找第一个值为 0 的位	int find_next_zero_ bit (void * addr, int size, int offset)	addr 为内存区的起始地址，size 为要查找的最大长度，offset 开始搜索的起始位号	
	ffz	在字中查找第一个 0	unsigned long ffz (un- signed long word);	word 为要搜索的字	
	ffs	查找第一个已设置的位	int ffs (int x)	x 为要搜索的字	这个函数的定义方式与 Libc 中的一样
	hweight32	返回一个 N 位字的加权平衡值	hweight32 (x)	x 为要加权的字	一个数的加权平衡是这个数所有位的总和

4．Linux 内存管理中 slab 缓冲区

函数名	功能	函数形成	参数	描述
kmem_cache_create	创建一个缓冲区	kmem_cache_t * kmem_cache_create (const char * name, size_t size, size_t offset, unsigned long flags, void (*ctor) (void*, kmem_cache _t *, unsigned long), void (*dtor) (void*, kmem_cache_t *, unsigned long));	Name 为在 /proc/slabinfo 中标识这个缓冲区的名字；size 为在这个缓冲区中创建对象的大小；offset 为页中的位移量；flags 为 slab 标志；ctor 和 dtor 分别为构造和析构对象的函数	成功则返回指向所创建缓冲区的指针，失败则返回空。不能在一个中断内调用该函数，但该函数的执行过程可以被中断。当通过该缓冲区分配新的页面时 ctor 运行，当页面被还回之前 dtor 运行
kmem_cache_shrink	缩小一个缓冲区	int kmem_cache_shrink (kmem_cache _t * cachep)	Cachep 为要缩小的缓冲区	为缓冲区释放尽可能多的 slab。为了有助于调试，返回 0 意味着释放所有的 slab
kmem_cache_destroy	删除一个缓冲区	int kmem_cache_destroy (kmem_cac- he_t * cachep);	cachep 为要删除的缓冲区	从 slab 缓冲区删除 kmem_cache_t 对象，成功则返回 0 这个函数应该在卸载模块时调用。调用者必须确保在 kmem_cache_destroy 执行期间没有其他对象再从该缓冲区分配内存
kmem_cache_alloc	分配一个对象	void * kmem_cache_alloc (kmem_cache_t * cachep, int flags);	cachep 为要删除的缓冲区，flags 请参见 kmalloc()	从这个缓冲区分配一个对象。只有当该缓冲区没有可用对象时，才用到标志 flags
kmalloc	分配内存	void * kmalloc (size_ t size, int flags)	size 为所请求内存的字节数，flags 为要分配的内存类型	kmalloc 是在内核中分配内存常用的一个函数。flags 参数的取值如下： GFP_USER – 代表用户分配内存，可以睡眠 GFP_KERNEL – 分配内核中的内存，可以睡眠 GFP_ATOMIC – 分配但不睡眠，在中断处理程序内部使用 另外，设置 GFP_DMA 标志表示所分配的内存必须适合 DMA，例如在 i386 平台上，就意味着必须从低 16MB 分配内存

kmem_cache_free	释放一个对象	void kmem_cache_free (kmem_cache_t * cachep, void * objp)	cachep 为曾分配的缓冲区，objp 为曾分配的对象	释放一个从这个缓冲区中曾分配的对象
kfree	释放以前分配的内存	void kfree (const void * objp)	objp 为由 kmalloc () 返回的指针	

5．Linux 中的 VFS

类别	函数名	功能	函数形成	参数	描述
目录项缓存	d_invalidate	使一个目录项无效	int d_invalidate (str- uct dentry * dentry)	dentry 为要无效的目录项	如果通过这个目录项能够到达其他的目录项，就不能删除这个目录项，并返回-EBUSY。如果该函数操作成功，则返回 0
	d_find_alias	找到索引节点一个散列的别名	struct dentry * d_find_alias (struct inode *	inode 为要讨论的索引节点	如果 inode 有一个散列的别名，就获取对这个别名，并返回它，否则返回空。注意，如果 inode 是一个目录，就只能有一个别名，如果它没有子目录，就不能进行散列
	prune_dcache	裁减目录项缓存	void prune_dcache (int count)	count 为要释放的目录项的一个域	缩小目录项缓存。当需要更多的内存，或者仅仅需要卸载某个安装点（在这个安装点上所有的目录项都不使用），则调用该函数 如果所有的目录项都在使用，则该函数可能失败
	shrink_dcache_sb	为一个超级块而缩小目录项缓存	void shrink_dcache _sb (struct super_ block * sb)	sb 为超级块	为一个指定的超级块缩小目录缓存。在卸载一个文件系统是调用该函数释放目录缓存
	have_submounts	检查父目录或子目录是否包含安装点	int have_submou-nts (struct dentry * parent)	parent 为要检查的目录项	如果 parent 或它的子目录包含一个安装点，则该函数返回真
	shrink_dcache_parent	裁减目录项缓存	void shrink_dcache _parent (struct dentry * parent)	parent 为要裁减目录项的父目录项	裁减目录项缓存以删除父目录项不用的子目录项
	d_alloc	分配一个目录项	struct dentry * d_ alloc (struct dentry * parent, const struct qstr * name)	parent 为要分配目录项的父目录项，name 为指向 qstr 结构的指针	分配一个目录项。如果没有足够可用的内存，则返回 NULL；成功则返回目录项

	d_instantiate	为一个目录项填充索引节点信息	void d_instantiate (struct dentry * entry, struct inode * inode)	entry为要完成的目录项，inode为这个目录项的 inode	在目录项中填充索引节点的信息。注意，这假定 inode 的 count 域已由调用者增加，以表示 inode 正在由该目录项缓存使用
	d_alloc_root	分配根目录项	struct dentry * d_ alloc_root (struct inode * root_inode)	root_inode 为要给根分配的 inode	为给定的 inode 分配一个根("/") 目录项，该 inode 被实例化并返回。如果没有足够的内存或传递的 inode 参数为空，则返回空

续表

类别	函数名	功能	函数形成	参数	描述
目录项缓存	d_lookup	查找一个目录项	struct dentry * d_ lookup (struct dentry * parent, struct qstr * name)	parent 为父目录项，name 为要查找的目录项名字的 qstr 结构。	为 name 搜索父目录项的子目录项。如果该目录项找到，则它的引用计数加 1，并返回所找到的目录项。调用者在完成了对该目录项的使用后，必须调用 d_put 释放它
	d_validate	验证由不安全源所提供的目录项	int d_validate (struct dentry * dentry, struct dentry * dparent)	dentry 是 dparent 有效的子目录项，dparent 是父目录项（已知有效）	一个非安全源向我们发送了一个 dentry，在这里，我们要验证它并调用 dget。该函数由 ncpfs 用在 readdir 的实现。如果 dentry 无效，则返回 0
	d_delete	删除一个目录项	void d_delete (struct dentry * dentry)	dentry 为要删除的目录项	如果可能，把该目录项转换为一个负的目录项，否则从哈希队列中移走它以便以后的删除
	d_rehash	给哈希表增加一个目录项	void d_rehash (struct dentry * entry)	dentry 为要增加的目录项	根据目录项的名字向哈希表增加一个目录项
	d_move	移动一个目录项	void d_move (struct dentry * dentry, struct dentry * target)	dentry 为要移动的目录项，target 为新目录项	更新目录项缓存以反映一个文件名的移动。目录项缓存中负的目录项不应当以这种方式移动

	__d_path	返回一个目录项的路径	char * __d_path (struct dentry * dentry, struct vfsmount * vfsmnt, struct dentry * root, struct vfsmount * rootmnt, char * buffer, int buflen)	dentry 为要处理的目录项，vfsmnt 为目录项所属的安装点，root 为根目录项，rootmnt为根目录项所属的安装点，buffer为返回值所在处，buflen 为 buffer 的长度	把一个目录项转化为一个字符串路径名。如果一个目录项已被删除，串“(deleted)”被追加到路径名，注意这有点含糊不清。返回值放在 buffer 中 “buflen”应该为页大小的整数倍。调用者应该保持 dcache_lock 锁
	is_subdir	新目录项是否是父目录项的子目录	int is_subdir (struct dentry * new_den- try, struct dentry * old_dentry)	new_dentry 为新目录项，old_dentry 为旧目录项	如果新目录项是父目录的子目录项（任何路径上），就返回 1，否则返回 0
	find_inode _number	检查给定名字的目录项是否存在	ino_t find_inode_ number (struct dentry * dir, struct qstr * name)	dir为要检查的目录，name 为要查找的名字	对于给定的名字，检查这个目录项是否存在，如果该目录项有一个 inode，则返回其索引节点号，否则返回 0

续表

类别	函数名	功能	函数形成	参数	描述
目录项缓存	d_drop	删除一个目录项	void d_drop (struct dentry * dentry)	dentry 为要删除的目录项	d_drop 从父目录项哈希表中解除目录项的哈希连接，以便通过 VFS 的查找再也找不到它。注意这个函数与 d_delete 的区别，d_delete 尽可能地把目录项表记为负的，查找时会得到一个负的目录项，而 d_drop 会使查找失败
	d_add	向哈希队列增加目录项	void d_add (struct dentry * entry, struct inode * inode)	dentry 为要增加的目录项，inode为与目录项对应的索引节点	该函数将把目录项加到哈希队列，并初始化 inode。这个目录项实际上已在 d_alloc () 函中得到填充
	dget	获得目录项的一个引用	struct dentry * dget (struct dentry * dentry)	dentry 为要获得引用的目录项	给定一个目录项或空指针，如果合适就增加引用 count 的值。当一个目录项有引用时（count 不为 0），就不能删除这个目录项。引用计数为 0 的目录项永远也不会调用 dget

	d_unhashed	检查目录项是否被散列	int d_unhashed (struct dentry * dentry)	dentry 为要检查的目录项	如果通过参数传递过来的目录项没有用哈希函数散列过，则返回真
索引节点处理	__mark_inode_dirty	使索引节点“脏”	void __mark_inode _dirty (struct inode * inode, int flags)	inode为要标记的索引节点，flags为标志，应当为 I_DIRTY_SYNC	这是一个内部函数，调用者应当调用 mark_inode_dirty 或 mark_inode_dirty_sync
	write_inode_now	向磁盘写一个索引节点	void write_inode_ now (struct inode * inode, int sync)	inode为要写到磁盘的索引节点，sync 表示是否需要同步。	如果索引节点为脏，该函数立即把它写到给磁盘。主要由 knfsd 来使用
	clear_inode	清除一个索引节点	void clear_inode (struct inode * inode)	inode为要写清除的索引节点	由文件系统来调用该函数，告诉我们该索引节点不再有用
	invalidate_inodes	丢弃一个设备上的索引节点	int invalidate_ inodes (struct super_block * sb);	sb为超级块	对于给定的超级块，丢弃所有的索引节点。如果丢弃失败，说明还有索引节点处于忙状态，则返回一个非 0 值。如果丢弃成功，则超级块中所有的节点都被丢弃。

续表

类别	函数名	功能	函数形成	参数	描述
索引节点处理	get_empty_inode	获得一个索引节点	struct inode * get_ empty_inode (void)	无	这个函数的调用发生在诸如网络层想获得一个无索引节点号的索引节点，或者文件系统分配一个新的、无填充信息的索引节点 成功则返回一个指向 inode 的指针，失败则返回一个 NULL 指针。返回的索引节点不在任何超级块链表中
	iunique	获得一个唯一的索引节点号	ino_t iunique (struct super_block * sb, ino_t max_reserved)	sb 为超级块，max_reserved 为最大保留索引节点号	对于给定的超级块，获得该系统上一个唯一的索引节点号。这一般用在索引节点编号不固定的文件系统中。返回的节点号大于保留的界限但是唯一。注意，如果一个文件系统有大量的索引节点，则这个函数会很慢
	insert_inode_hash	把索引节点插入到哈希表	void insert_inode_ hash (struct inode * inode)	inode为要插入的索引节点	把一个索引节点插入到索引节点的哈希表中，如果该节点没有超级块，则把它加到一个单独匿名的链中

	remove_inode_hash	从哈希表中删除一个索引节点	void remove_inode_hash (struct inode * inode)	inode为要删除的索引节点	从超级块或匿名哈希表中删除一个索引节点
	iput	释放一个索引节点	void iput (struct inode * inode)	inode为要释放的索引节点	如果索引节点的引用计数变为0，则释放该索引节点，并且可以撤销它
	bmap	在一个文件中找到一个块号	int bmap (struct inode * inode, int block)	inode为文件的索引节点，block为要找的块	返回设备上的块号，例如，寻找索引节点1的块4，则该函数将返回相对于磁盘起始位置的盘块号
	update_atime	更新访问时间	void update_atime (struct inode * inode)	inode为要访问的索引节点	更新索引节点的访问时间，并把该节点标记为写回。这个函数自动处理只读文件系统、介质、“noatime”标志以及具有“noatime”标记者的索引节点
	make_bad_inode	由于I/O错误把一个索引节点标记为坏	void make_bad_inode (struct inode * inode)	inode为要标记为坏的索引节点	由于介质或远程网络失败而造成不能读一个索引节点时，该函数把该节点标记为“坏”，并引起从这点开始的 I/O 操作失败
	is_bad_inode	是否是一个错误的inode	int is_bad_inode (struct inode * inode)	inode为要测试的索引节点	如果要测试的节点已标记为坏，则返回真

续表

类别	函数名	功能	函数形成	参数	描述
注册以及超级块	register_filesystem	注册一个新的文件系统	int register_filesys- tem (struct file_sys-tem_type * fs)	fs 为指向文件系统结构的指针	把参数传递过来的文件系统加到文件系统的链表中。成功则返回0，失败则返回一个负的错误码
	unregister_filesystem	注销一个文件系统	int unregister_files- ystem (struct file_ system_type * fs)	fs 为指向文件系统结构的指针	把曾经注册到内核中的文件系统删除。如果没有找到个文件系统，则返回一个错误码，成功则返回0 这个函数所返回的 file_system_type 结构被释放或重用
	get_super	获得一个设备的超级块	struct super_block * get_super (kdev_t dev)	dev为要获得超级块的设备	扫描超级块链表，查找在给定设备上安装的文件系统的超级块。如果没有找到，则返回空

6．Linux 的连网

套	函数名	功能	函数形成	参数	描述

接字缓冲区函数	skb_queue_empty	检查队列是否为空	int skb_queue_empty (struct sk_buff_head * list)	list 为队列头	如果队列为空返回真，否则返回假
	skb_get	引用缓冲区	struct sk_buff * skb_get (struct sk_buff * skb)	skb 为要引用的缓冲区	对套接字缓冲区再引用一次，返回指向缓冲区的指针
	kfree_skb	释放一个 sk_buff	void kfree_skb (struct sk_buff * skb)	sk 为要释放的缓冲区	删除对一个缓冲区的引用，如果其引用计数变为 0，则释放它
	skb_cloned	缓冲区是否是克隆的	int skb_cloned (struct sk_buff * skb)	skb 为要检查的缓冲区	如果以 skb_clone 标志来产生缓冲区，并且是缓冲区多个共享拷贝中的一个，则返回真。克隆的缓冲区具有共享数据，因此在正常情况下不必对其进行写
	skb_shared	缓冲区是否是共享的	int skb_shared (struct sk_buff * skb)	skb 为要检查的缓冲区	如果有多于一个的人引用这个缓冲区就返回真
	skb_share_check	检查缓冲区是否共享的,如果是就克隆它	struct sk_buff * skb_share_check (struct sk_buff * skb, int pri)	skb 为要检查的缓冲区,pri 为内存分配的优先级	如果缓冲区是共享的，就克隆这个缓冲区，并把原来缓冲区的引用计数减 1，返回新克隆的缓冲区。如果不是共享的，则返回原来的缓冲区。当从中断状态或全局锁调用该函数时，pri 必须是 GFP_ATOMIC 内存分配失败则返回 NULL

续表

套接字缓冲区函数	函数名	功能	函数形成	参数	描述
	skb_queue_len	获得队列的长度	__u32 skb_queue_len (struct sk_buff_ head * list_)	list_为测量的链表	返回&sk_buff 队列的指针
	__skb_queue_head	在链表首部对一个缓冲区排队	void __skb_queue_ head (struct sk_buff _head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区	在链表首部对一个缓冲区进行排队。这个函数没有锁，因此在调用它之前必须持有必要的锁。一个缓冲区不能同时放在两个链表中
	skb_queue_head	在链表首部对一个缓冲区排队	void skb_queue_ head (struct sk_buff _head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区	在链表首部对一个缓冲区进行排队。这个函数此可以安全地使用。一个缓冲区不能同时放在两个链表中

	__skb_queue_tail	在链表尾部对一个缓冲区排队	void __skb_queue_tail (struct sk_buff_ head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区	在链表尾部对一个缓冲区进行排队。这个函数没有锁，因此在调用它之前必须持有必要的锁。一个缓冲区不能同时放在两个链表中
	skb_queue_tail	在链表尾部对一个缓冲区排队	void skb_queue_tail (struct sk_buff_ head * list, struct sk_buff * newsk)	list 为要使用的链表，newsk 为要排队的缓冲区	在链表尾部对一个缓冲区进行排队。这个函数有锁，因此可以安全地使用。一个缓冲区不能同时放在两个链表中
	__skb_dequeue	从队列的首部删除一个缓冲区	struct sk_buff * __skb_dequeue (struct sk_buff_ head * list)	list 为要操作的队列	删除链表首部。这个函数不持有任何锁，因此使用时应当持有适当的锁。如果队链表为空则返回 NULL，成功则返回首部元素
	skb_dequeue	从队列的首部删除一个缓冲区	struct sk_buff * skb_dequeue (struct sk_buff_head * list)	list 为要操作的队列	删除链表首部，这个函数持有锁，因此可以安全地使用。如果队链表为空则返回 NULL，成功则返回首部元素。
	skb_insert	插入一个缓冲区	void skb_insert (struct sk_buff * old, struct sk_buff * newsk)	old 为插入之前的缓冲区，newsk 为要插入的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作。一个缓冲区不能同时放在两个链表中
	skb_append	追加一个缓冲区	void skb_append (struct sk_buff * old, struct sk_buff * newsk)	old 为插入之前的缓冲区，newsk 为要插入的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作。一个缓冲区不能同时放在两个链表中
	skb_unlink	从链表删除一个缓冲区	void **skb_unlink** (struct sk_buff * *skb*);	Skb 为要删除的缓冲区	把一个数据包放在链表中给定的包之前。该函数持有链表锁，并且是原子操作

续表

套接字缓冲区	函数名	功能	函数形成	参数	描述
	skb_dequeue_tail	从队头删除	struct sk_buff * **skb_dequeue_tail** (struct sk_buff_ head * *list*)	List 为要操作的链表	删除链表尾部，这个函数持有锁，因此可以安全地使用。如果队链表为空则返回 NULL，成功则返回首部元素

区函数	skb_put	把数据加到缓冲区	unsigned char * **skb_put** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要使用的缓冲区，len 为要增加的数据长度	这个函数扩充缓冲区所使用的数据区。如果扩充后超过缓冲区总长度，内核会产生警告。函数返回的指针指向所扩充数据的第 1 字节
	skb_push	把数据加到缓冲区的开始	unsigned char * **skb_push** (struct sk_buff * *skb*, unsigned int *len*);	skb 为要使用的缓冲区，len 为要增加的数据长度	这个函数扩充在缓冲区的开始处缓冲区所使用的数据区。如果扩充后超过缓冲区首部空间的总长度，内核会产生警告。函数返回的指针指向所扩充数据的第一个字节
	skb_pull	从缓冲区的开始删除数据	unsigned char * **skb_pull** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要使用的缓冲区，len 为要删除的数据长度	这个函数从链表开始处删除数据，把腾出的内存归还给首部空间。把指向下一个缓冲区的指针返回
	skb_headroom	缓冲区首部空闲空间的字节数	int **skb_headroom** (const struct sk_buff * *skb*)	skb 为要检查的缓冲区	返回&sk_buff 首部空闲空间的字节数
	skb_tailroom	缓冲区尾部的空闲字节数	int **skb_tailroom** (const struct sk_buff * *skb*)	skb 为要检查的缓冲区	返回&sk_buff 尾部空闲空间的字节数
	skb_reserve	调整头部的空间	void **skb_reserve** (struct sk_buff * *skb*, unsigned int *len*)	skb 为要改变的缓冲区，len 为要删除的字节数	通过减少尾部空间，增加一个空&sk_buff 的首部空间。这仅仅适用于空缓冲区
	skb_trim	从缓冲区删除尾部	void **skb_trim** (struct sk_buff * *skb*, unsigned int *len*);	skb 为要改变的缓冲区，len 为新的长度	通过从尾部删除数据，剪切缓冲区的长度。如果缓冲区已经处于指定的长度，则不用改变
	skb_orphan	使一个缓冲区成为孤儿	void **skb_orphan** (struct sk_buff * *skb*);	skb 是要成为孤儿的缓冲区	如果一个缓冲区当前有一个拥有者，我们就调用拥有者的析构函数，使 skb 没有拥有者。该缓冲区继续存在，但以前的拥有者不再对其“负责”
	skb_queue_purge	使一个链表空	void **skb_queue_purge** (struct sk_buff_head * *list*)	list 为要腾空的链表	删除在&sk_buff 链表上的所有缓冲区。这个函数持有链表锁，并且是原子的

续表

套	函数名	功能	函数形成	参数	描述

接字缓冲区函数	dev_alloc_skb	为发送分配一个skbuff	struct sk_buff * **dev_alloc_skb** (unsigned int *length*)	length 为要分配的长度	分配一个新的&sk_buff，并赋予它一个引用计数。这个缓冲区有未确定的头空间。用户应该分配自己需要的头空间。 如果没有空闲内存，则返回NULL。尽管这个函数是分配内存，但也可以从中断来调用
	skb_cow	当需要时拷贝 skb 的首部	struct sk_buff * **skb_cow** (struct sk_buff * *skb*, unsigned int *headroom*)	skb 为要拷贝的缓冲区，headroom 为需要的头空间	如果传递过来的缓冲区缺乏足够的头空间或是克隆的，则该缓冲区被拷贝，并且附加的头空间变为可用。如果没有空闲的内存，则返回空。如果缓冲区拷贝成功，则返回新的缓冲区，否则返回已存在的缓冲区
	skb_over_panic	私有函数	void **skb_over_panic** (struct sk_buff * *skb*, int *sz*, void * *here*)	skb 为缓冲区，sz 为大小，here 为地址	用户不可调用
	skb_under_panic	私有函数	void **skb_under_panic** (struct sk_buff * *skb*, int *sz*, void * *here*)	skb 为缓冲区，sz 为大小，here 为地址	用户不可调用
	alloc_skb	分配一个网络缓冲区	struct sk_buff * **alloc_skb** (unsigned int *size*, int *gfp_mask*)	size 为要分配的大小，gfp_mask 为分配掩码	分配一个新的&sk_buff。返回的缓冲区没有 size 大小的头空间和尾空间。新缓冲区的引用计数为 1。返回值为一缓冲区，如果失败则返回空。从中断分配缓冲区，掩码只能使用 GFP_ATOMIC 的 gfp_mask
	__kfree_skb	私有函数	void **__kfree_skb** (struct sk_buff * *skb*)	skb 为缓冲区	释放一个 sk_buff。释放与该缓冲区相关的所有事情，清除状态。这是一个内部使用的函数，用户应当调用 kfree_skb
	skb_clone	复制一个 sk_buff	struct sk_buff * **skb_clone** (struct sk_buff * *skb*, int *gfp_mask*)	skb 为要克隆的缓冲区，gfp_mask 为分配掩码	复制一个&sk_buff。新缓冲区不是由套接字拥有。两个拷贝共享相同的数据包而不是结构。新缓冲区的引用计数为 1。如果分配失败，函数返回 NULL，否则返回新的缓冲区。如果从中断调用这个函数，掩码只能使用 GFP_ATOMIC 的 gfp_mask

续表

	函数名	功能	函数形成	参数	描述
套接字缓冲区函数	skb_copy_expand	拷贝并扩展 sk_buff	struct sk_buff * **skb_copy_expand** (const struct sk_buff * *skb*, int *newheadroom*, int *newtailroom*, int *gfp_mask*);	skb 为要拷贝的缓冲区，newhead- room 为头部的新空闲字节数，newtailroom 为尾部的新空闲字节数	既拷贝&sk_buff 也拷贝其数据，同时分配额外的空间。当调用者希望修改数据并需要对私有数据进行改变，以及给新的域更多的空间时调用该函数。失败返回 NULL，成功返回指向缓冲区的指针 返回的缓冲区其引用计数为 1。如果从中断调用，则必须传递的优先级为 GFP_ATO- MIC

7．网络设备支持

	函数名	功能	函数形成	参数	描述
驱动程序的支持	init_etherdev	注册以太网设备	truct net_device * **init_etherdev** (struct net_device * *dev*, int *sizeof_priv*)	dev 为要填充的以太网设备结构，或者要分配一个新的结构时为 NULL，sizeof_priv 是为这个以太网设备要分配的额外私有结构的大小	用以太网的通用值填充这个结构的域。如果传递过来的 dev 为 NULL，则构造一个新的结构，包括大小为 sizeof_priv 的私有数据区。强制将这个私有数据区在 32 字节（不是位）上对齐
	dev_add_pack	增加数据包处理程序	void **dev_add_pack** (struct packet_type * *pt*)	pt 为数据包类型	把一个协议处理程序加到网络栈，把参数传递来的&packet_type 链接到内核链表中
	dev_remove_pack	删除数据包处理程序	void **dev_remove_pack** (struct packet_type * *pt*)	pt 为数据包类型	删除由 dev_add_pack 曾加到内核的协议处理程序。把&packet_type 从内核链表中删除，一旦该函数返回，这个结构还能再用
	__dev_get_by_name	根据名字找设备	struct net_device * **__dev_get_by_name** (const char * *name*);	name 为要查找的名字	根据名字找到一个接口。必须在 RTNL 信号量或 dev_base_lock 锁的支持下调用。如果找到这个名字，则返回指向设备的指针，如果没有找到，则返回 NULL。引用计数器并没有增加，因此调用者必须小心地持有锁

续表

	函数名	功能	函数形成	参数	描述
驱动程序的支持	dev_get	测试设备是否存在	int **dev_get** (const char *name)	name 为要测试的名字	测试名字是否存在。如果找到则返回真。为了确保在测试期间名字不被分配或删除，调用者必须持有 rtnl 信号量。这个函数主要用来与原来的驱动程序保持兼容
	__dev_get_by_index	根据索引找设备	struct net_device * **__dev_get_by_index** (int *ifindex*)	ifindex 为设备的索引	根据索引搜索一个接口。如果没有找到设备，则返回 NULL，找到则返回指向设备的指针。该设备的引用计数没有增加，因此调用者必须小心地关注加锁，调用者必须持有 RTNL 信号量或 dev_base _lock 锁
	dev_get_by_index	根据名字找设备	struct net_device * **dev_get_by_index** (int *ifindex*)	ifindex 为设备的索引	根据索引搜索一个接口。如果没有找到设备，则返回 NULL，找到则返回指向设备的指针。所返回设备的引用计数加 1，因此，在用户调用 dev_put 释放设备之前，返回指针是安全的
	dev_alloc_name	为设备分配一个名字	int **dev_alloc_name** (struct net_device * *dev*, const char *name)	dev 为设备，name 为格式化字符串。	传递过来一个格式化字符串，例如 ltd，该函数试图找到一个合适的 id。设备较多时这是很低效的。调用者必须在分配名字和增加设备时持有 dev_base 或 rtnl 锁，以避免重复。返回所分配的单元号或出错返回一个复数
	dev_alloc	分配一个网络设备和名字	struct net_device * **dev_alloc** (const char *name, int * *err*)	name 为格式化字符串，err 为指向错误的指针	传递过来一个格式化字符串，例如 ltd，函数给该名字分配一个网络设备和空间。如果没有可用内存，则返回 NULL。如果分配成功，则名字被分配，指向设备的指针被返回。如果名字分配失败，则返回 NULL，错误的原因放在 err 指向的变量中返回。调用者必须在做这一切时持有 dev_base 或 RTNL 锁，以避免重复分配名字
	netdev_state_change	设备改变状态	void **netdev_state_change** (struct net_device * *dev*)	name 为引起通告的设备	当一个设备状态改变时调用该函数

续表

	函数名	功能	函数形成	参数	描述
驱动程序的支持	dev_open	为使用而准备一个接口	int **dev_open** (str- uct net_device * *dev*)	device 为要打开的设备	以从低层到上层的过程获得一个设备。设备的私有打开函数被调用，然后多点传送链表被装入，最后设备被移到上层，并把 NETDEV_UP 信号发送给网络设备的 notifier chain 在一个活动的接口调用该函数只能是个空操作。失败则返回一个负的错误代码
	dev_close	关闭一个接口	int **dev_close** (struct net_device * *dev*)	dev 为要关闭的设备	这个函数把活动的设备移到关闭状态。向网络设备的 notifier chain 发送一个 NETDEV_GOING_DOWN。然后把设备变为不活动状态，并最终向 notifier chain 发 NETDEV_DOWN 信号
	register_netdevice_notifier	注册一个网络通告程序块	int **register_netdevice_notifier** (struct notifier_ block * *nb*)	nb 为通告程序	当网络设备的事件发生时，注册一个要调用的通告程序。作为参数传递来的通告程序被连接到内核结构，在其被注销前不能重新使用它。失败则返回一个负的错误码
	unregister_netdevice_notifier	注销一个网络通告块	int **unregister_netdevice_notifier** (struct notifier _block * *nb*)	nb 为通告程序	取消由 register_netdevice_notifer 曾注册的一个通告程序。把这个通告程序从内核结构中解除，然后还可以重新使用它。失败则返回一个负的错误码
	dev_queue_xmit	传送一个缓冲区	int **dev_queue_xmit** (struct sk_ buff * *skb*)	skb 为要传送的缓冲区	为了把缓冲区传送到一个网络设备，对缓冲区进行排队。调用者必须在调用这个函数前设置设备和优先级，并建立缓冲区。该函数也可以从中断中调用。失败返回一个负的错误码。成功并不保证帧被传送，因为也可能由于拥塞或流量调整而撤销这个帧
	netif_rx	把缓冲区传递到网络协议层	void **netif_rx** (struct sk_buff * *skb*)	skb 为要传送的缓冲区	这个函数从设备驱动程序接收一个数据包，并为上层协议的处理对其进行排队。该函数总能执行成功。在处理期间，可能因为拥塞控制而取消这个缓冲区。

续表

	函数名	功能	函数形成	参数	描述
驱动程序的支持	register_gifconf	注册一个 SIOCGIF 处理程序	int **register_gifconf** (unsigned int *family*, gifconf_func_t * *gifconf*)	family 为地址族，gifconf 为处理程序	注册由地址转储例程决定的协议。当另一个处理程序替代了由参数传递过来的处理程序时，才能释放或重用后者
	netdev_set_master	建立主／从对	int **netdev_set_master** (struct net_device * *slave*, struct net_ device *master)	slave 为从设备，master 为主设备。	改变从设备的主设备。传递 NULL 以中断连接。调用者必须持有 RTNL 信号量。失败返回一个负错误码。成功则调整引用计数，RTM_NEWLINK 发送给路由套接字，并且返回 0
	dev_set_allmulti	更新设备上多个计数	void **dev_set_all-multi** (struct net_device * *dev*, int *inc*)	dev 为设备，inc 为修改者	把接收的所有多点传送帧增加到设备或从设备删除。当设备上的引用计数依然大于 0 时，接口保持着对所有接口的监听。一旦引用计数变为 0，设备回转到正常的过滤操作。负的 inc 值用来在释放所有多点传送需要的某个资源时减少其引用计数
	dev_ioctl	网络设备的 ioctl	int **dev_ioctl** (unsigned int *cmd*, void * *arg*)	cmd 为要发出的命令，arg 为用户空间指向 ifreq 结构的指针	向设备发布 ioctl 函数。这通常由用户空间的系统调用接口调用，但有时也用作其他目的。返回值为一个正数，则表示从系统调用返回，为负数，则表示出错
	dev_new_index	分配一个索引	int **dev_new_index** (*void*)	无	为新的设备号返回一个合适而唯一的值。调用者必须持有 rtnl 信号量以确保它返回唯一的值。
	netdev_finish_unregister	完成注册	int **netdev_finish_unregister** (struct net_device * *dev*)	dev 为设备	撤销或释放一个僵死的设备。成功返回 0
	unregister_netdevice	从内核删除设备	int **unregister_netdevice** (struct net_device * *dev*)	dev 为设备。	这个函数关闭设备接口并将其从内核表删除。成功返回 0，失败则返回一个负数。

8390网卡	ei_open	打开／初始化网板	int **ei_open** (str- uct net_device * *dev*)	dev 为要初始化的网络设备	尽管很多注册的设备在每次启动时仅仅需要设置一次，但这个函数在每次打开设备时还彻底重新设置每件事。
	ei_close	关闭网络设备	int **ei_close** (str- uct net_device * *dev*)	dev 为要关闭的网络设备	ei_open 的相反操作，在仅仅在完成“ ifconfig<devname> down ”时使用

续表

	函数名	功能	函数形成	参数	描述
8390网卡	ethdev_init	初始化 8390 设备结构的其余部分	int **ethdev_init** (struct net_device * *dev*)	dev 为要初始化的网络设备结构	初始化 8390 设备结构的其余部分。不要用__init ()，因为这也由基于 8390 的模块驱动程序使用
	NS8390_init	初始化 8390 硬件	void **NS8390_init** (struct net_device * *dev*, int *start*p)	dev 为要初始化的设备，startp 为布尔值，非 0 启动芯片处理。	必须持以锁才能调用该函数

8．模块支持

	函数名	功能	函数形成	参数	描述
模块装入	request_module	试图装入一个内核模块	int **request_mod- ule** (const char * *module_name*)	module_name 为模块名	使用用户态模块装入程序装入一个模块。成功返回 0，失败返回一个负数。注意，一个成功的装入并不意味着这个模块在自己出错时就能卸载和退出。调用者必须检查他们所提出的请求是可用的，而不是盲目地调用。 如果自动装入模块的功能被启用，那么这个函数就不起作用。
	call_usermode helper	启动一个用户态的应用程序	int call_usermod- ehelper (char * path, char ** argv, char ** envp);	path 为应用程序的路径名，argv 为以空字符结束的参数列表，envp 为以空字符结束的环境列表	运行用户空间的一个应用程序。该应用程序被异步启动。它作为keventd的子进程来运行，并具有 root 的全部权能。Keventd 在退出时默默地获得子进程 必须从进程的上下文中调用该函数，成功返回 0，失败返回一个负数。

内部模块支持	inter_module_register	注册一组新的内部模块数据	void inter_module_register (const char * im_name, struct module * owner, const void * userdata)	im_name 为确定数据的任意字符串，必须唯一，owner 为正在注册数据的模块，通常用 THIS_MODULE，userdata 指向要注册的任意用户数据	检查 im_name 还没有被注册，如果已注册就发出“抱怨”。对新数据，则把它追加到 inter_module_entry 链表。

续表

	函数名	功能	函数形成	参数	描述
内部模块支持	inter_module_get	从另一模块返回任意的用户数据	const void * inter_module_get (const char * im_name)	im_name 为确定数据的任意字符串，必须唯一	如果 im_name 还没有注册，则返回 NULL。增加模块拥有者的引用计数，如果失败则返回 NULL，否则返回用户数据
	inter_module_get_request	内部模块自动调用 request_module	const void * inter_module_get_request (const char * im_name, const char * modname)	im__name 为确定数据的任意字符串，必须唯一；modname 为期望注册 m_name 的模块	如果 inter_module_get 失败，调用 request_module，然后重试
	inter_module_put	释放来自另一个模块的数据	void inter_module_put (const char * im_name)	im__name 为确定数据的任意字符串，必须唯一	如果 im_name 还没有被注册，则“抱怨”，否则减少模块拥有者的引用计数

9．硬件接口

硬件处理	函 数 名	功 能	函数形成	参 数	描 述
	Disable_irq_nosync	不用等待使一个 irq 无效	void inline **disable_irq_nosync** (unsigned int *irq*)	irq 为中断号	使所选择的中断线无效。使一个中断栈无效。与 disable_ irq 不同，这个函数并不确保 IRQ 处理程序的现有实例在退出前已经完成。可以从 IRQ 的上下文中调用该函数。

	Disable_irq	等待完成使一个 irq 无效	void **disable_irq** (unsigned int *irq*)	irq 为中断号	使所选择的中断线无效。使一个中断栈无效 这个函数要等待任何挂起的处理程序在退出之前已经完成。如果你在使用这个函数，同时还持有 IRQ 处理程序可能需要的一个资源，那么，你就可能死锁。要小心地从 IRQ 的上下文中调用这个函数
	Enable_irq	启用 irq 的	void **enable_irq** (unsigned int *irq*)	irq 为中断号	重新启用这条 IRQ 线上的中断处理。在 IRQ 的上下文中调用这个函数
	Probe_irq_mask	扫描中断线的位图	unsigned int **probe_irq_mask**(unsigned long *val*)	val 为要考虑的中断掩码	扫描 ISA 总线的中断线，并返回活跃中断的位图。然后把中断探测的逻辑状态返回给它以前的值

续表

	函数名	功能	函数形成	参数	描述
MTRR 处理	Mtrr_del	删除一个内存区类型	int **mtrr_del** (int *reg*, unsigned long *base*, unsigned lon- g *size*);	reg 为由 mtrr_add 返回的寄存器，base 为物理基地址，size 为内存区大小	如果提供了寄存器 reg，则 base 和 size 都可忽略。这就是驱动程序如何调用寄存器。如果引用计数降到 0，则释放该寄存器，该内存区退回到缺省状态。成功则返回寄存器，失败则返回一个负数
PCI 支持库	pci_find_slot	从一个给定的 PCI 插槽定位 PCI	struct pci_dev * **pci_find_slot** (unsigned int *bus*, unsigned int *devfn*)	bus为所找PCI设备所驻留的 PCI 总线的成员，devfn为PCI插槽的成员	给定一个 PCI 总线和插槽号，所找的 PCI 设备位于 PCI 设备的系统全局链表中。如果设备被找到，则返回一个指向它的数据结构，否则返回空

	pci_find_device	根据 PCI 标识号开始或继续搜索一个设备	struct pci_dev * **pci_find_device** (unsigned int *vendor*, unsigned int *device*, const struct pci_dev * *from*)	vendor 为要匹配的 PCI 商家 id，或要与所有商家 id 匹配的 PCI_ANY _ID，device 为要匹配的 PCI 设备 id，或要与所有商家 id 匹配的 PCI_ ANY_ ID，from 为以前搜索中找到的 PCI 设备，或对于一个新的搜索来说为空	循环搜索已知 PCI 设备的链表。如果找到与 vendor 和 device 匹配的 PCI 设备，则返回指向设备结构的指针，否则返回 NULL 给 from 参数传递 NULL 参数则开始一个新的搜索，否则，如果 from 不为空，则从那个点开始继续搜索
	pci_find_class	根据类别开始或继续搜索一个设备	struct pci_dev * **pci_find_class** (unsigned int *class*, const struct pci_dev * *from*)	class：根据类别名称搜索 PCI 设备 Previous：在搜索着找到的 PCI 设备，对于新的搜索则为 NULL	循环搜索已知 PCI 设备的链表。如果找到与 class 匹配的 PCI 设备，则返回指向设备结构的指针，否则返回 NULL 给 from 参数传递 NULL 参数则开始一个新的搜索，否则，如果 from 不为空，则从那个点开始继续搜索
	pci_find_capability	查询设备的权能	int pci_find_ capability (struct pci_dev * dev, int cap)	dev 为要查询的 PCI 设备，cap 为权能取值	断定一个设备是否支持给定 PCI 权能。返回在设备 PCI 配置空间内所请求权能结构的地址，如果设备不支持这种权能，则返回 0

续表

PCI 支持库	函数名	功能	函数形成	参数	描述
	pci_set_power_state	设置一个设备电源管理的状态	int **pci_set_ power_state** (struct pci_dev * *dev*, int *new_state*)	dev 为 PCI 设备，new_state 为新的电源管理声明(0 == D0, 3 == D3 等)	设置设备的电源管理状态。对于从状态 D3 的转换，并不像想象的那么简单，因为很多设备在唤醒期间忘了它们的配置空间。返回原先的电源状态
	pci_save_state	保存设备在挂起之前 PCI 的配置空间	int pci_save_ state (struct pci_dev * dev, u32 * buffer)	dev 为我们正在处理的 PCI 设备，buffer 为持有配置空间的上下文	缓冲区必须足够大，以保持整个 PCI 2.2 的配置空间(>= 64 bytes)。

	pci_restore_state	恢复PCI设备保存的状态	int pci_restore_state (struct pci_dev * dev, u32 * buffer)	dev为我们正在处理的 PCI 设备，buffer 为保存的配置空间	
	pci_enable_device	驱动程序使用设备前进行初始化	int pci_enable_device (struct pci_dev * dev)	dev为要初始化的PCI设备	驱动程序使用设备前对设备进行初始化。请求低级代码启用 I/O 和内存。如果设备被挂起，则唤醒它。小心，这个函数可能失败
	pci_disable_device	使用PCI设备之后使其无效	void pci_disable_device (struct pci_dev * dev)	dev 为使无效的PCI设备	向系统发送信号，以表明系统不再使用 PCI 设备。这仅仅包括使 PCI 总线控制（如果激活）无效
	pci_enable_wake	当设备被挂起时启用设备产生PME#	int pci_enable_wake (struct pci_dev * dev, u32 state, int enable)	dev为对其实施操作的 PCI 设备，state为设备的当前状态，enable为启用或禁用“产生”的标志	当系统被挂起时，在设备的 PM 能力中设置位以产生PME#。如果设备没有PM能力，则返回-EIO。如果设备支持它，则返回-EINVAL，但不能产生唤醒事件。如果操作成功，则返回0
	pci_release_regions	释放保留的PCI I/O和内存资源	void pci_release_regions (struct pci_dev * pdev)	pdev为PCI设备，其资源以前曾由pci_request_regions保留。	释放所有的 PCI I/O 和以前对 pci_request_regions 成功调用而使用的内存。只有在 PCI 区的所有使用都停止后才调用这个函数
	pci_request_regions	保留 PCI I/O和内存资源	int pci_request_regions (struct pci_dev * pdev, char * res_ name)	pdev为PCI设备，它的资源要被保留，res_name 为与资源相关的名字	把所有与PCI设备pdev相关联的 PCI 区进行标记，设备pdev是由属主res_name保留的。除非这次调用成功返回，否则不要访问 PCI 内的任何地址 成功返回0,出错返回EBUSY，失败时也打印警告信息

续表

PCI支持库	函数名	功能	函数形成	参数	描述
	pci_unregister_driver	注销一个PCI设备	void pci_unregi-ster_driver (struct pci_ driver * drv)	drv为要注销的驱动程序结构	从已注册的 PCI 驱动程序链表中删除驱动程序结构，对每个驱动程序所驱动的设备，通过调用驱动程序的删除函数，给它一个清理的机会，把这些设备标记为无驱动程序的

	pci_insert_device	插入一个热插拔设备	void pci_insert_device (struct pci_dev * dev, struct pci_bus * bus)	dev 为要插入的设备，bus 为 PCI 总线，设备就插入到该总线	把一个新设备插入到设备列表，并向用户空间（/sbin/hotplug）发出通知
	pci_remove_device	删除一个热插拔设备	void pci_remove_device (struct pci_dev * dev)	dev 为要删除的设备	把一个新设备从设备列表删除，并向用户空间（/sbin/hotplug）发出通知。
	pci_dev_driver	获得一个设备的 pci_driver	struct pci_driver * pci_dev_driver (const struct pci_dev * dev)	dev 为要查询的设备	返回合适的 pci_driver 结构，如果一个设备没有注册的驱动程序，则返回 NULL
	pci_set_master	为设备 dev 启用总线控制	void pci_set_master (struct pci_dev * dev)	dev 为要启用的设备	启用设备上的总线控制，并调用 pcibios_set_master 对特定的体系结构进行设置
	pci_setup_device	填充一个设备的类和映射信息	int pci_setup_device (struct pci_dev * dev)	dev 为要填充的设备结构	用有关设备的商家、类型、内存及 IO 空间地址，IRO 线等初始化设备结构。在 PCI 子系统初始化时调用该函数。成功返回 0，设备类型未知返回-1

10．块设备

函数名	功能	函数形式	参数	描述	其他
blk_cleanup_queue	当不再需要一个请求队列时，释放一个 request_queue_t	void blk_cleanup_queue (request_queue_t * q);	q 为要释放的请求队列	blk_cleanup_queue 与 blk_init_ queue 是成对出现的。应该在释放请求队列时调用该函数；典型的情况是块设备正被注销时调用。该函数目前的主要任务是释放分配到队列中所有的 struct request 结构	低级驱动程序有希望首先完成任何重要的请求。

续表

函数名	功能	函数形式	参数	描述	其他

blk_queue_headactive	指明请求队列的头是否可以是活跃的	void blk_queue_headactive (request_queue_t * q, int active)	q 为这次申请的队列，active 为一个标志，表示队列头在哪儿是活跃的	块设备驱动程序可以选定把当前活动请求留在请求队列，只有在请求完成时才移走它。队列处理例程为安全起见把这种情况假定为缺省值，并在请求被撤销时，将不再在合并或重新组织请求时包括请求队列的头 如果驱动程序在处理请求之前从队列移走请求，它就可以在合并和重新安排中包含队列头。这可以通过以 active 标志为 0 来调用 blk_queue_headactive 如果一个驱动程序一次处理多个请求，它必须从请求队列移走他们（或至少一个） 当一个队列被插入，则假定该队列头为不活跃的	
blk_queue_make_request	为设备定义一个交替的 make_request 函数	void blk_queue_make_request (request_queue_t * q, make_request_fn * mfn)	q 为受影响设备的请求队列，mfn 为交替函数	把 buffer_heads 结构传递到设备驱动程序的常用方式为让驱动程序把请求收集到请求队列，然后让驱动程序准备就绪时把请求从那个队列移走。这种方式对很多块设备驱动程序很有效。但是，有些块设备（如虚拟设备 md 或 lvm）并不是这样，而是把请求直接传递给驱动程序，这可以通过调用 blk_queue_make_request () 函数来达到	按以上方式操作的驱动程序必须能够恰当地处理在“高内存”的缓冲区，这是通过调用 bh_kmap 获得一个内核映射，或通过调用 create_bounce 在常规内存创建一个缓冲区
blk_init_queue	为块设备的使用准备一个请求队列	void blk_init_queue (request_queue_t * q, request_fn_proc * rfn)	q 为要初始化的请求队列，rfn 为处理请求所调用的函数	如果一个块设备希望使用标准的请求处理例程，就调用该函数。当请求队列上有待处理的请求时，调用 rfn 函数	blk_init_queue 的反操作函数为 blk_cleanup_queue，当撤销块设备时调用后者（例如在模块卸载时）

续表

函数名	功能	函数形式	参数	描述	其他
generic_make_request	形成块设备的 I/O 请求	void generic_make_request (int rw, struct buffer_head * bh)	rw为I/O操作的类型，即READ、WRITE或READA，bh是内存和磁盘上的缓冲区首部	READ和WRITE的含义很明确，READA为预读。该函数不返回任何状态。请求的成功与失败，以及操作的完成是由bh->b_end_io递送的	
submit_bh	类似于上一个函数	void submit_bh (int rw, struct buffer_head * bh)	rw为I/O操作的类型，即READ、WRITE或READA，bh为描述I/O的buffer_head	该函数与与generic_make_request的目的非常类似，但submit_bh做更多的事情。	
ll_rw_block	对块设备的低级访问	void ll_rw_block (int rw, int nr, struct buffer_head * * bhs)	rw为READ、WRITE或READA，nr为数组中buffer_heads的个数，bhs为指向buffer_heads的数组	对普通文件的读／写和对块设备的读／写，都是通过调用该函数完成的	所有的缓冲区必须是针对同一设备的

11. USB 设备

函数名	功能	函数形成	参数	描述
usb_register	注册一个USB设备	Int usb_register (struct usb_driver * new_driver)	new_driver为驱动程序的USB操作	注册一个具有USB核心的USB驱动程序。只要增加一个新的驱动程序，就要扫描一系列独立的接口，并允许把新的驱动程序与任何可识别的设备相关联，成功则返回0，失败则返回一个负数
usb_scan_devices	扫描所有未申明的USB接口	Usb_scan_devices (void)	无	扫描所有未申明的USB接口，并通过“probe”函数向它们提供所有已注册的USB驱动程序。这个函数将在usb_register () 调用后自动地被调用
usb_deregister	注销一个USB驱动程序	Usb_deregister (struct usb_driver * driver)	Driver为要注销的驱动程序的USB操作	从USB内部的驱动程序链表中取消指定的驱动程序
usb_alloc_bus	创建一个新的USB宿主控制器结构	Struct usb_bus * usb_alloc_bus (struct	op为指向struct usb_operations的指针，这是	创建一个USB宿主控制器总线结构，并初始化所有必要的内部对象（仅仅由USB宿主控制器使用）。如果没有可用内存，则返回NULL

		usb_operations * op)	一个总线结构	

续表

函数名	功能	函数形成	参数	描述
usb_free_bus	释放由总线结构所使用的内存	Void usb_free_bus (struct usb_bus * bus)	无	（仅仅由 USB 宿主控制器驱动程序使用）
usb_register_bus	注册具有 usb 核心的 USB 宿主控制器	Void usb_register_bus (struct usb_bus * bus);	bus 指向要注册的总线	仅仅由 USB 宿主控制器驱动程序使用