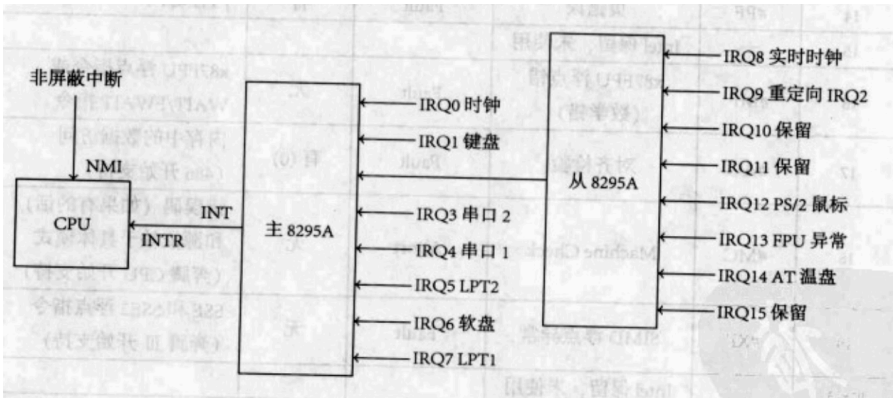
目 录

1 中断和异常的实现

1.1 设置 8259A

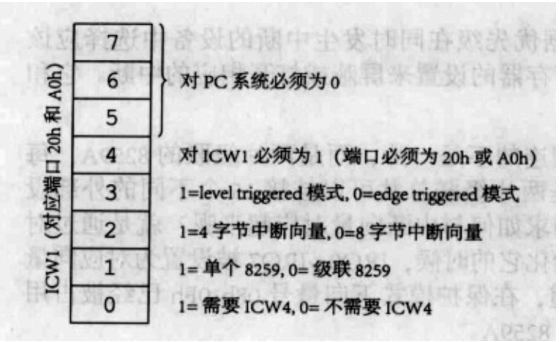
8259A 是中断机制中所有外围设备的一个代理，这个代理可以根据优先级在同时发生中断的设备中选择应该处理的请求。除此之外，还可以通过对 8259A 的寄存器的设置来屏蔽或打开相应的中断。可屏蔽外部中断与 CPU 是通过 8259A 连接起来的。8259A 与 CPU 的连接如下图所示：



由图可知，每一片 8259A 有 8 根中断信号线，两片级联的 8259A 可以挂接 15 个不同的外部设备。这些外部设备发出中断请求时，8259A 将其与相应的中断向量号对应起来。所以我们需要设置 8259A。

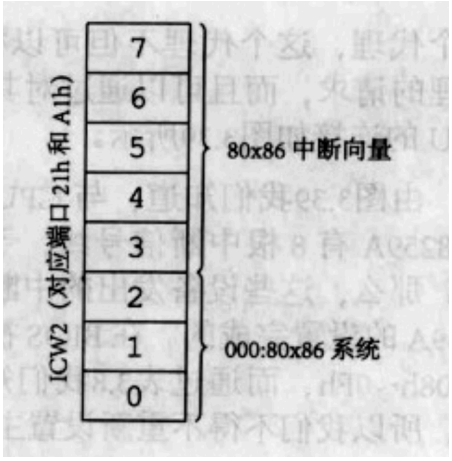
设置 8259A 的过程就是向其相应的端口写入特定的 ICW。主 8259A 的端口有 20h 和 21h，从 8259A 的端口有 A0h 和 A1h。ICW 全称是 Initialization Command Word，大小为一个字节。初始化 8259A 的过程如下：

- 首先往端口 20h 和 A0h 写入 ICW1。ICW1 的格式如图所示：

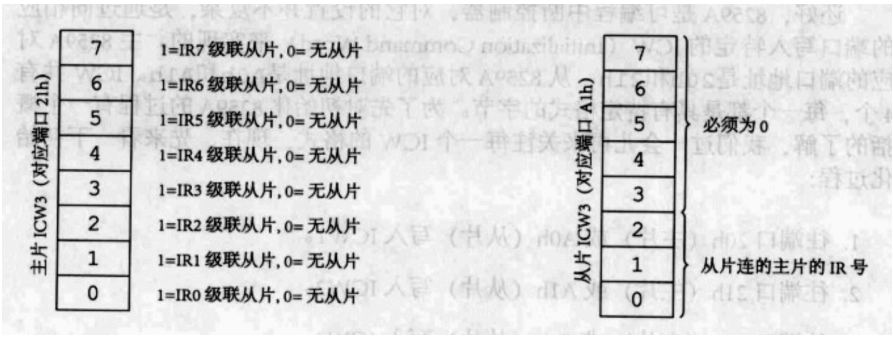


- 然后往端口 21h 和 A1h 写入 ICW2。主 8259A 和从 8259A 的 ICW2 内容可以不一样。写入 ICW2 时涉及与中断向量号的对应。比如，往主 8259A 写入 ICW2 时，如

果 ICW2 为 20h，那么 IRQ0 ～ IRQ7 就对应中断向量 20h ～ 27h。ICW2 的格式如图所示：



- 然后往端口 21h 和 A1h 写入 ICW3。主 8259A 的 ICW3 和从 8259A 的 ICW3 的格式不同。两个 ICW3 如图所示：



- 最后往端口 21h 和 A1h 写入 ICW4。ICW4 的格式如图所示：

实现代码如下所示：

```
; io_delay函数用于等待操作完成
io_delay:
    nop
    nop
    nop
    nop
    ret

Init8259A:
    ; 往端20h写入ICW1
    mov al, 011h
    out 020h, al
    call io_delay
    ; 往端A0h写入ICW1
    out 0A0h, al
    call io_delay
    ; 往端21h写入ICW2
    mov al, 020h
    out 021h, al
    call io_delay
    ; 往端A1h写入ICW2
    mov al ,028h
    out 0A1h, al
    call io_delay
    ; 往端21h写入ICW3
    mov al, 004h
    out 021h, al
    call io_delay
    ; 往端A1h写入ICW3
    mov al, 002h
    out 0A1h, al
    call io_delay
    ; 往端21h写入ICW4
    mov al, 001h
    out 021h, al
    call io_delay
    ; 往端A1h写入ICW4
    out 0A1h, al
    call io_delay
```

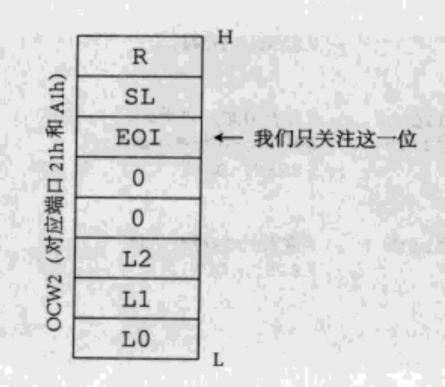
除了 ICW 字段，8259A 还接受 OCW 字段。OCW 全称为 operation control word。我们在两种情况下用到 OCW 字段。这两种情况是：

- 当屏蔽或打开外部中断时。此时端口 21h 或 A1h 写入 OCW1。OCW1 的格式如下图：

如果想屏蔽某一个外部中断，将对应的那一位设为 1 就可以了。实现代码如下：

```
        mov al, 11111111b
        out 0A1h, al
        call io_delay
```

- 发送 EOI 给 8259A 来通知它中断处理结束了，以便继续接收中断。此时往端口 20h 或 A0h 写 OCW2。OCW2 的格式如下图：



实现代码如下所示：

```
        mov al, 20h
        out 20h, al
```

1.2 建立 IDT

IDT 表和 GDT 表格式类似，它存放着中断门描述符、陷阱门描述符和任务门描述符。IDT 将每一个中断向量号和一个描述符对应起来。实现代码如下：

```
[SECTION .idt]
ALIGN 32
[BITS 32]
LABEL_IDT:
.01h: Gate SelectorCode32, SpuriousHandler, 0, DA_386IGate
; ...
IdtLen equ $ - LABEL_IDT
IdtPtr dw IdtLen - 1
       dd 0
```

加载 IDT 的代码如下：

```
xor eax, eax
mov ax, ds
shl eax, 4
add eax, LABEL_IDT
mov dword [IdtPtr + 2], eax

cli

lidt [IdtPtr]
```

1.3 实现一个中断

修改一下 IDT，将第 80h 号中断的中断处理函数改为 _UserIntHandler。实现代码如下所示：

```
[SECTION .idt]
ALIGN 32
[BITS 32]
LABEL_IDT:
; ...
.080h: Gate SelectorCode32, UserIntHandler, 0, DA_386IGate
; ...
IdtLen equ $ - LABEL_IDT
IdtPtr dw IdtLen - 1
       dd 0
; ...
[SECTION .s32]
[BITS 32]
_UserIntHandler:
UserIntHandler equ _UserIntHandler - $$
    mov ax, SelectorVideo
    mov gs, ax
    mov ah, 0Ch
    mov al, 'I'
    mov [gs:((80 * 0 + 70) * 2)], ax
    iretd
```

然后在 32 位代码段中添加如下代码，就可以实现一个中断。代码如下：

```
call Init8259A
int 080h
```

1.4 时钟中断试验

如果想打开时钟中断，一方面要打开外部中断，一方面要设计相应的中断处理程序。

首先打开外部中断，需要向 8259A 的 21h 或 A1h 写入相应的 OCW1，并且设置 IF 位为 1。时钟中断请求为 IRQ0。实现代码如下：

```
; 打开定时器中断
mov al, 11111110b
out 021h, al
call io_delay
; 屏蔽从8159A的所有中断
mov al, 11111111b
out 0A1h, al
call io_delay

ret
```

在初始化 8259A 中，我们将 IRQ0 的中断向量号设置为 20h，所以需要在 IDT 的第 20h 项中写相应的时钟中断处理程序。

```
[SECTION .idt]
ALIGN 32
[BITS 32]
LABEL_IDT:
; ...
.20h: Gate SelectorCode32, ClockHandler, 0, DA_386IGate
; ...

[SECTION .s32]
[BITS 32]
; 这个函数实现了将屏幕第0行、第70行的字符加一的功能
_ClockHandler:
ClockHandler equ _ClockHandler - $$
    inc byte [gs:((80 * 0 + 70) * 2)]
    ; 向端口20h写入OCW2，通知中断处理程序结束
    mov al, 20h
    out 20h, al
    iretd
```

下面是查看时钟中断效果的代码：

```
; 首先将屏幕第0行、第70行的字符设置为a
mov ax, SelectorVideo
mov gs, ax
mov ah, 0Ch
mov al, 'a'
mov [gs:((80 * 0 + 70) * 2)], ax
; 初始化8259A
```

```
8     call Init8259A
9     ; 打开时钟中断
10    mov al, 11111110b
11    mov 021h, al
12    call io_delay
13    ; 设置IF位为1，打开外部中断
14    sti
15    ; 让程序陷入循环，可以查看每次时钟中断后屏幕上字符的变化
16    jmp $
```

1.5 几点需要注意的事

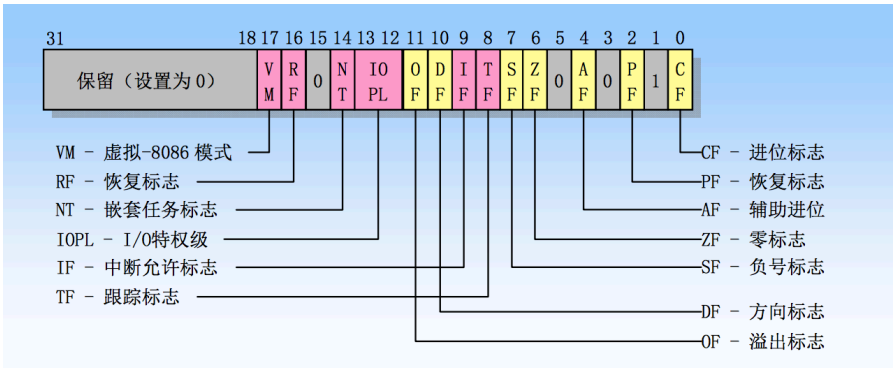
- 当中断产生时，大多会有特权级变换。通过中断门和陷阱门的中断相当于用 call 指令调用一个调用门，所以和第五次学习报告中特权级变换的内容一样。
- 中断或异常发生时，会和 call 指令一样进行压栈操作。需要注意的是，有些中断会产生出错码。如果有出错码，在 iretd 执行时，出错码是不会从堆栈中自动弹出的。所以在执行 iretd 之前，应该先将出错码从堆栈中清除掉。
- 中断门和陷阱门唯一的区别在于，中断门会对中断允许标志 IF 产生影响。由中断门向量引起的中断会复位 IF，在 iretd 指令执行后，会恢复 IF 位的原值。

2 保护模式下的 I/O

I/O 的控制权限是需要严格控制的，操作系统通过 IOPL 和 I/O 许可位图实现对 I/O 控制权限的限制。

2.1 IOPL

IOPL 字段位于 Eflags 寄存器的第 12、13 位。如下图所示：



操作系统将一些指令定义为 I/O 敏感指令，这些指令只有在 CPL≤IOPL 时才能执行，如果低特权级的任务试图执行这些指令将会引起一般性保护异常。I/O 敏感指令包括 in、ins、out、outs、cli 和 sti。

IOPL 字段是可以修改的，程序可以通过 popf 和 iretd 指令修改 IOPL 字段。只有当任务特权级为 0 时，popf 和 iretd 才可以成功修改 IOPL 的值。否则即使执行了指令，IOPL 也不会改变，不过也不会引起异常。

popf 指令还可以用来改变 IF 标志，只有当 CPL≤IOPL 时，才能成功修改 IF 标志，否则 IF 将维持原值，不会产生任何异常。

2.2 I/O 许可位图

在第五次学习报告中，我有实现过 TSS。其中代码有一处是“I/O 位图基址”。I/O 位图基址指向的就是 I/O 许可位图。I/O 许可位图的每一位用于表示一个字节的端口地址是否可用。如果该位为 0，表示此位对应的端口号可用，为 1 则代表不可用。I/O 许可位图的使用使得即便在同一特权级下不同任务也可以有不同的 I/O 访问权限。

I/O 许可位图就位于 TSS 段中，而 I/O 位图基址实际上是以 TSS 的地址为基址的偏移。如果 I/O 位图基址大等于 TSS 段界限，就表示 TSS 段中没有 I/O 许可位图。由于每个任务都有单独的 TSS，所以每个任务都有自己单独的 I/O 许可位图。

下面是一个任务中 I/O 许可位图的实现代码：

```
1    [SECTION .tss3]
2    LABEL_TSS3:
3        DW $ - LABEL_TSS3 + 2 ; 指向I/O许可位图
4    times 12 DB 0FFh ; 端口00h~5fh都不可用
5        DB 11111101b ; 端口60h~67h，只有端口61h可以用
6        DB 0FFh ; I/O许可位图结束标志,I/O许可位图必须以0FFh结尾
7    TSS3Len equ $ - LABEL_TSS3
```

3 linux 下的内存管理

3.1 页

内核用 struct page 结构表示系统中的每个物理页，这个结构在 mm_types.h 中定义。定义代码如下：

```
// 这个代码省去了联合结构体
struct page
{
    unsigned long flags;
    atomic_t _count;
    atomic_t _mapcount;
    unsigned long private;
    struct address_space* mapping;
    pgoff_t index;
    struct list_head lru;
    void* virtual;
}
```

下面是对 page 结构体中各个域的介绍：

- flag 域用来存放页的状态，flag 的每一位单独表示一种状态，所以它至少可以同时表示出 32 中不同的状态。这些标志在 page-flags.h 文件中定义，代码如下：

```
enum pageflags
{
    PG_locked, // 该页被锁住
    PG_error, // 此页发生了一个I/O错误
    PG_referenced, // used for page reclaim for anonymous pagecache
    PG_uptodate, // 表示该页的内容是否有效
    PG_dirty, // 该页为脏页
    PG_lru,
    PG_active,
    PG_slab,
    PG_owner_priv_1,
    PG_arch_1,
    PG_reserved, // 表示页永远不会被换出，也有可能不存在
    PG_private, // 表示该页含有文件系统中特定的数据
    PG_private_2,
    PG_writeback,
    PG_head,
    PG_swapcache,
    PG_mappedtodisk,
    PG_reclaim,
    PG_swapbacked,
    PG_unevictable,
    PG_mlocked,
    PG_uncached,
    PG_hwpoison,
    PG_young,
    PG_idle,
```

```
28          __NR_PAGEFLAGS,
29          PG_checked = PG_owner_priv_1,
30          PG_fscache = PG_private_2,
31          PG_pinned = PG_owner_priv_1,
32          PG_savepinned = PG_dirty,
33          PG_foreign = PG_owner_priv_1,
34          PG_slob_free = PG_private,
35          PG_double_map = PG_private_2,
36          PG_isolated = PG_reclaim
37      };
```

- _count 域存放在页的引用计数。
- virtual 域是页的虚拟地址，它就是页在虚拟内存中的地址。

需要知道的是，page 结构只是用于描述当前时刻在相关物理页中存放的东西。这个数据结构的目的在于描述物理内存本身，而并没有描述包含在其中的数据。内核使用这个数据结构来管理系统中所有的页。

3.2 区

系统中存在两种因为硬件缺陷而引起的内存寻址问题：

- 一些硬件只能用特定的内存地址来执行 DMA。
- 一些体系结构的内存的物理寻址范围比虚拟寻址范围大得多，导致有一些内存不能永久地映射到内核空间中。

为了解决这两个问题，内核把页分为了六个区：

- ZONE_DMA，这个区用于执行 DMA 操作。
- ZONE_DMA32，这个区同样用于执行 DMA 操作，只是这些页面只能被 32 位设备访问。
- ZONE_NORMAL，这个区包含的是能正常映射的页。
- ZONE_HIGHEM，这个区包含的页不能永久地映射到内核空间中。
- ZONE_MOVABLE
- ZONE_DEVICE

需要注意的是，区的划分是没有任何物理意义的，这只是内核为了管理页而采取的一种逻辑上的分组。linux 把系统的页划分为区，形成不同的内存池，然后根据用途进行分配。例如，当需要内存用于执行 DMA 操作，就可以从 ZONE_DMA 中按照请求的数目取出页。

下面是区的数据结构的定义，内核用它来管理系统中所有的区：

```
struct zone
{
    unsigned long watermark[NR_WMARK];
    unsigned long lowmem_reserve[MAX_NR_ZONES];
    struct per_cpu_pageset pageset[NR_CPUS];
    spinlock_t lock;
    struct free_area free_area[MAX_ORDER];
    spinlock_t lru_lock;
    struct zone_lru
    {
        struct list_head list;
        unsigned long nr_saved_scan;
    }lru[NR_LRU_LISTS];
    struct zone_reclaim_stat reclaim_stat;
    unsigned long pages_scanned;
    unsigned long flags;
    atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS];
    int prev_priority;
    unsigned int inactive_ratio;
    wait_queue_head_t *wait_table;
    unsigned long wait_table_hash_nr_entries;
    unsigned long wait_table_bits;
    struct palist_data *zone_pgdat;
    unsigned long zone_start_pfn;
    unsigned long spanned_pages;
    unsigned long present_pages;
    const char* name;
};
```

区的数据结构有 3 个重要的域，如下所示：

- watermark 数组。内核使用水位为每个内存区设置合适的内存消耗基准，watermark 数组持有该区水位所能达到的最小值、最低和最高水位值。
- lock 域。lock 是一个自旋锁，用于防止这个结构被并发访问。
- name 域。name 用于表示这个区的名字，内核启动期间将初始化这个值，三个区的名字分别为”DMA”、”Normal” 和”HighMem”。

3.3 gfp_mask 标志

内核中定义了三类分配器标志：

- 行为修饰符，用于表示内核应当如何分配所需的内存。
- 区修饰符，用于表示从哪里分配内存。
- 类型修饰符，组合了行为修饰符和区修饰符。

以上三个描述符都定义在 gfp.h 文件中，一般只使用类型修饰符。下面是各个类型修饰符及其相应描述：

标　志	描　述
GFP_ATOMIC	这个标志用在中断处理程序、下半部、持有自旋锁以及其他不能睡眠的地方
GFP_NOWAIT	与 GFP_ATOMIC 类似，不同之处在于，调用不会退给紧急内存池。这就增加了内存分配失败的可能性。
GFP_NOIO	这种分配可以阻塞，但不会启动磁盘 I/O。这个标志在不能引发更多磁盘 I/O 时能阻塞 I/O 代码，这可能导致令人不愉快的递归
GFP_NOFS	这种分配在必要时可能阻塞，也可能启动磁盘 I/O，但是不会启动文件系统操作。这个标志在你不能再启动另一个文件系统的操作时，用在文件系统部分的代码中
GFP_KERNEL	这是一种常规分配方式，可能会阻塞。这个标志在睡眠安全时用在进程上下文代码中。为了获得调用者所需的内存，内核会尽力而为。这个标志应当是首选标志
GFP_USER	这是一种常规分配方式，可能会阻塞。这个标志用于为用户空间进程分配内存时
GFP_HIGHUSER	这是从 ZONE_HIGHMEM 进行分配，可能会阻塞。这个标志用于为用户空间进程分配内存
GFP_DMA	这是从 ZONE_DAM 进行分配。需要获取能供 DMA 使用的内存的设备驱动程序使用这个标志，通常与以上的某个标志组合在一起使用

下面介绍了各个类型修饰符使用的情形：

情　形	相应标志
进程上下文，可以睡眠	使用 GFP_KERNEL
进程上下文，不可以睡眠	使用 GFP_ATOMIC，在你睡眠之前或之后以 GFP_KERNEL 执行内存分配
中断处理程序	使用 GFP_ATOMIC
软中断	使用 GFP_ATOMIC
tasklet	使用 GFP_ATOMIC
需要用于 DMA 的内存，可以睡眠	使用（GFP_DMA | GFP_KERNEL）
需要用于 DMA 的内存，不可以睡眠	使用（GFP_DMA | GFP_ATOMIC），或在你睡眠之前执行内存分配

3.4 获得页

内核提供了一种请求内存的底层机制，并提供了相应的接口函数，这些接口函数以页为单位分配内存。其中最核心的函数为：

```
1    struct page* alloc_pages(gfp_t gfp_mask, unsigned int order);
```

这个函数位于 gfp.h 文件中，用于分配 2^{order} 个连续的物理页。函数还返回了一个指针，用于指向第一个页的 page 结构体。

内核使用下面的函数把给定的页转换成相应的逻辑地址，这个函数返回的是指向物理页当前逻辑地址的指针：

```
1    void* page_address(struct page* page);
```

3.5 释放页

当内核不再需要页的时候，它使用下面的函数释放页：

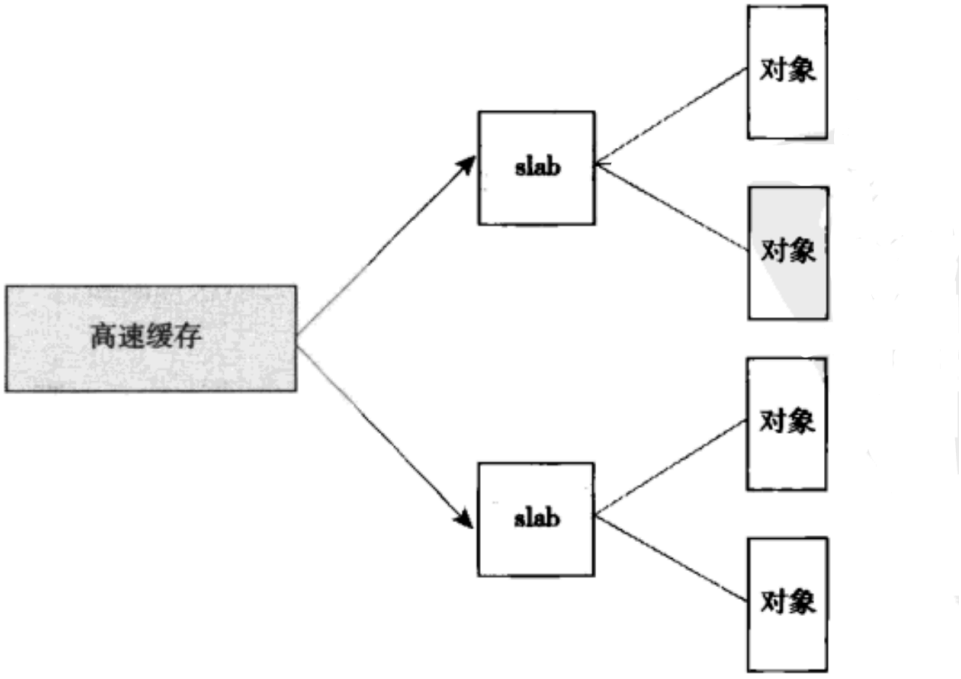
```
1  void __free_pages(struct page* page, unsigned int order);
2  void free_pages(unsigned long addr, unsigned int order);
3  void free_page(unsigned long addr);
```

3.6 slab 层

slab 层在 linux 内核中扮演着通用数据结构缓存层的角色。

3.6.1 slab 层的设计

slab 层管理着多个高速缓存，每个高速缓存由一个或多个 slab 组成，slab 用于存放对象。高速缓存、slab 和对象的关系如下图所示：



每个高速缓存使用 kmem_cache 结构来表示，这个结构包含三个链表：slabs_full、slabs_partial、slabs_empty。这三个链表存放在 kmem_list3 结构中，用于表示满、部分满或空的 slab。这三个链表包含了高速缓存中所有的 slab。定义代码如下：

```
1  struct kmem_cache
```

```
    {
        struct array_cache* array[NR_CPUS];
        unsigned int batchcount;
        unsigned int limit;
        unsigned int shared;
        unsigned int buffer_size;
        u32 reciprocal_buffer_size;
        unsigned int flags;
        unsigned int num;
        unsigned int gfporder;
        gfp_t gfpflags;
        size_t colour;
        unsigned int colour_off;
        struct kmem_cache* slabp_cache;
        unsigned int slab_size;
        unsigned int dflags;
        void (*ctor)(void *obj);
        struct list_head next;
        struct kmem_list3* nodelists[MAX_NUMNODES];
    };
```

kmem_list3 的定义如下：

```
    struct list_head
    {
        struct list_head *next, *prev;
    };
    struct kmem_list3
    {
        struct list_head slabs_partial;
        struct list_head slabs_full;
        struct list_head slabs_free;
        unsigned long free_objects;
        unsigned int free_limit;
        unsigned int colour_next;
        spinlock_t list_lock;
        struct array_cache* shared;
        struct array_cache** alien;
        unsigned long next_reap;
        int free_touched;
    };
```

slab 结构体定义如下：

```
    struct slab
    {
        struct list_head list;
        unsigned long colour_off;
        void* s_mem;
        unsigned int inuse;
        kmem_bufctl_t free;
        unsigned short nodeid;
    };
```

slab 层可以创建新的 slab。slab 由一个或几个页组成，所以 slab 层可以通过 __get_free_pages()

函数新建 slab。

slab 层通过下面的函数新建高速缓存：

```
struct kmem_cache* kmem_cache_create(
    const char* name, // 存放高速缓存的名字
    size_t size, // 缓存中每个元素的大小
    size_t align, // slab中第一个对象的偏移
    unsigned long flags, // 用于控制高速缓存的行为
    void (*ctor)(void*) // 高速缓存的构造函数
);
```

slab 层通过下面的函数销毁高速缓存：

```
int kmem_cache_destroy(struct kmem_cache *cachep);
```

3.6.2 分配对象

slab 层通过下面的函数获得对象，这个函数返回指向对象的指针。如果高速缓存中的所有 slab 都没有空闲的对象，那么就新建 slab。函数如下：

```
void* kmem_cache_alloc(struct kmem_cache* cachep, gfp_t flags);
```

slab 层使用下面的函数释放对象：

```
void kmem_cache_free(struct kmem_cache* cachep, void* objp);
```

3.6.3 使用 slab 层的例子

fork.c 里就使用 slab 层进行内存的分配。首先，内核定义了一个全局变量，这个高速缓存用于存放 task_struct 结构。

```
struct kmem_cache* task_struct_cachep;
```

这个高速缓存在 fork_init() 中被初始化：

```
task_struct_cachep = kmem_cache_create(
    "task_struct",
    sizeof(struct task_struct),
    ARCH_MIN_TASKALIGN,
    SLAB_PANIC | SLAB_NOTRACK,
    NULL
);
```

当调用 fork() 函数时，系统创建新进程，调用 do_fork() 函数。在复制父进程的过程中，调用 copy_process() 函数。该函数又调用 dup_task_struct()。在这个函数中，将创建一个新的进程描述符，这时就用 slab 层中分配对象的函数分配空间给新进程描述符，代码如下：

```
static struct task_struct* dup_task_struct(struct task_struct* orig)
{
    struct task_struct* tsk;
    tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
    if(!tsk)
        return NULL;
    // ...
}

static struct task_struct* copy_process(
        unsigned long clone_flags,
        unsigned long stack_start,
        struct pt_regs* regs,
        unsigned long stack_size,
        int __user *child_tidptr,
        struct pid* pid,
        int trace
                            )
{
    struct task_struct* p;
    // ...
    p = dup_task_struct(current);
    // ...
}

long do_fork(
            unsigned long clone_flags,
            unsigned long stack_start,
            struct pt_regs* regs,
            unsigned long stack_size,
            int __user *parent_tidptr,
            int __user *child_tidptr
            )
{
    struct task_struct* p;
    // ...
    p = copy_process(clone_flags, stack_start, regs, stack_size, child_tidptr,
        NULL, trace);
    // ...
}
```

当进程执行完后，如果没有子进程在等待，那么它的进程描述符将被释放，并返回给 task_struct_cachep 这个高速缓存，实现代码如下：

```
kmem_cache_free(task_struct_cachep, tsk);
```

4 初始化与清理

4.1 初始化

类似于 C++ 的方式，Java 也采用了构造器进行初始化。例子如下：

```
class Rock
{
    Rock()
    {
        System.out.println("Hello world");
    }
}
```

和 C++ 一样，Java 的构造器也可以进行重载。例子如下：

```
class Tree
{
    int height;
    Tree()
    {
        System.out.println("Planting a seeding");
        height = 0;
    }
    Tree(int inititalHeight)
    {
        height = inititalHeight;
        System.out.println("Creating new Tree that is " + height + " feer tall"
            );
    }
}
```

关于默认构造器，这里需要注意：如果程序员没有提供任何构造器，编译器将自动生成一个默认构造器；如果程序员写了一个构造器，编译器就不会自动生成一个默认构造器。

4.2 this 关键字

Java 的 this 关键字和 C++ 的 this 关键字有区别。C++ 的 this 关键字是该对象的地址，而 Java 中是没有指针的。Java 的 this 关键字表示该对象的引用。例子如下：

```
public class Leaf
{
    int i = 0;
    Leaf increment()
    {
        ++i;
        return this; // 相当于返回当前对象
    }
```

```
9       }
```

4.2.1 利用 this 调用构造器

在构造器中，如果为 this 添加了参数列表，将产生对符合此参数列表的某个构造器的明确调用。例子如下：

```
1   public class Flower
2   {
3       int petalCount = 0;
4       String s = "initial value";
5       Flower(int petals)
6       {
7           petalCount = petals;
8       }
9       Flower(String ss)
10      {
11          s = ss;
12      }
13      Flower(String s, int petals)
14      {
15          this(petals);
16          // 不能再写this(s)，否则会覆盖原有的内容
17          this.s = s;
18      }
19      void print()
20      {
21          // 在非构造函数中不能使用this(s)
22          System.out.println("Hello world!");
23      }
24  }
```

4.3 成员初始化

即使程序员没有对类的每个基本类型数据成员初始化，它们都会有一个初始值。在类中定义一个对象引用时，如果不将其初始化，该对象引用的初值就是 null。

与 C++ 不同的是，如果想为类中某个变量赋初值，只要在定义类成员变量的地方将其赋值即可，而 C++ 不允许这样的行为。

4.3.1 使用构造器进行初始化

可以使用构造器进行初始化，但是这无法阻止自动初始化的进行，它将在构造器被调用前发生。

在类的内部，变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于函数定义之间，它们也将在任何函数 (包括构造器) 被调用之前得到初始化。例子如下：

```
class Window
{
    Window(int order)
    {
        System.out.println("order is " + order);
    }
}

class House
{
    Window w1 = new Window(1);
    House()
    {
        w3 = new Window(4);
    }
    Window w2 = new Window(2);
    Window w3 = new Window(3);
}

public class OrderOfInitialization
{
    public static void main(String[] args)
    {
        House h = new House();
    }
}
```

Java 中 static 关键字不能应用于局部变量，只能作用于域。当类中的静态基本类型域没有初始化时，它的值会是基本类型的标准初值。如果它是一个对象引用，那么它的默认初始化值是 null。如果想在定义处进行初始化静态数据，采取的方法与非晶态数据没有什么不同。

静态变量只有在必要时刻才会初始化。只有类被第一次访问或使用时，类中静态成员才会被初始化，而且是类中所有静态成员变量一起被初始化。此后，静态对象不会再次被初始化。需要注意的是，静态成员变量在非静态成员变量之前被初始化。例子如下:

```
class Bowl
{
    Bowl(int marker)
    {
        System.out.println("order is " + marker);
    }
    void f1()
    {
        System.out.println("f1: Hello world");
    }
}

class Table
{
    static Bowl bowl1 = new Bowl(1);
    Table()
    {
```

```
18              System.out.println("Table()");
19              bowl2.foo();
20          }
21          void foo2()
22          {
23              System.out.println("foo2: Hello world");
24          }
25          static Bowl bowl2 = new Bowl(2);
26      }
27
28      class CupBoard
29      {
30          Bowl bowl3 = new Bowl(3);
31          static Bowl bowl4 = new Bowl(4);
32          CupBoard()
33          {
34              System.out.println("CupBoard()");
35              bowl4.foo1();
36          }
37          void foo3()
38          {
39              System.out.println("foo3: Hello world");
40          }
41          static Bowl bowl5 = new Bowl(5);
42      }
43
44      public class StaticInitialization
45      {
46          public static void main(String[] args)
47          {
48              System.out.println("Creating new CupBoard() in main");
49              new CupBoard();
50              System.out.println("Creating new CupBoard() in main");
51              new CupBoard();
52              table.foo2();
53              cupboard.foo3();
54          }
55          static Table table = new Table();
56          static CupBoard cupboard = new CupBoard();
57      }
```

假设有一个名为 Dog 的类，Dog 对象创建的步骤如下：

- 当首次创建类型为 Dog 的对象，或者 Dog 类的静态成员变量或静态成员函数 (构造器本身也是静态函数) 被访问到时，Java 解释器必须查找类路径，用于定位 Dog.class 文件，然后载入 Dog.class。
- 载入 Dog.class 之后，将执行静态初始化的所有动作。也就是说，静态初始化在类对象首次加载时进行。
- 使用 new 创建对象时，会在堆上为 Dog 对象分配足够的存储空间。然后将这块存储空间清零，这样一来，Dog 对象中所有基本类型都被设置成了默认值，而引用被设置为 null。

- 随后执行所有出现于字段定义处的初始化动作。
- 执行构造器。

4.3.2 显式的静态初始化

Java 允许将多个静态初始化动作组织成一个特殊的“静态子句”。例子如下：

```
public class Spoon
{
    static int i;
    static
    {
        i = 47;
    }
}
```

静态子句只会执行一次。当首次生成这个类的一个对象或者首次访问属于这个类的静态成员变量或静态成员函数时，该代码段将执行。

4.3.3 数组初始化

Java 不允许指定数组的大小。例子如下：

```
// 现在拥有的是对数组的一个引用
// 我们并没有给数组对象本身分配任何空间
// 我们只是为该引用分配了空间
int a[];
```

4.4 清理

Java 拥有一个垃圾回收器，用于释放由 new 分配的内存。在其他情况下，如果该内存区域不是由 new 分配，那么垃圾回收器就不知道该如何释放这块特殊内存。

为了解决这种情况，Java 允许在类中定义一个名为 finalize() 的函数。一旦垃圾回收器准备释放对象占用的存储空间，将首先调用其 finalize() 函数，并在垃圾回收动作发生时，真正地回收对象占用的内存。

finalize() 函数与 C++ 中的析构函数有区别，因为 C++ 中的对象一定会被销毁，而 Java 中的对象不一定总是被垃圾回收器回收，而且垃圾回收并不等于析构。需要知道的是，只要程序没有濒临存储空间用完的时刻，垃圾回收器就不会释放程序所创建的任何对象的存储空间。

4.4.1 finalize 函数的用途

finalize 函数不会负责释放对象所占有的内存。无论对象是如何创建的，垃圾回收器都会负责释放对象占据的所有内存。只有当 Java 程序中调用了本地方法分配空间时，finalize 函数才会派上用场。本地方法是一种在 Java 中调用非 Java 代码的方式。在非 Java 代码中，可能会调用 C 的 malloc 函数分配存储空间，此时只有使用了 free 函数，该内存空间才会释放。所以在对象释放时，需要在 finalize 函数中调用 free 函数来释放这块特殊的内存，否则将引起内存泄漏。

下面是使用 finalize() 函数的例子：

```
class Book
{
    boolean checkedOut = false;
    Book(boolean checkOut)
    {
        checkedOut = checkOut;
    }
    void checkIn()
    {
        checkedOut = false;
    }
    protected void finalize()
    {
        if(checkedOut)
            System.out.println("Error: checked out");
    }
}

public class TerminationCondition
{
    public static void main(String[] args)
    {
        Book novel = new Book(true);
        novel.checkIn();
        new Book(true);
        System.gc();
    }
}
```

以上程序通过 finalize() 函数确保所有 Book 对象在被当作垃圾回收钱都应该被 check in。当有一本书没有 check in 时，程序将输出错误情况。类似的，如果一个对象代表了一个打开的文件，在对象被回收前程序员应该关闭这个文件。finalize() 函数可以用来检查文件是否关闭。finalize 函数更多地被用来发现程序中隐晦的缺陷。

4.4.2 垃圾回收器的工作机制

首先需要意识到，Java 中除了基本类型，所有对象都在堆上分配空间。然而 Java 从堆分配空间的速度，可以和其他语言从堆栈上分配空间的速度相媲美。

对比一下 Java 与 C++ 的堆空间分配机制。C++ 的堆像一个院子，里面的每个对象都

负责管理自己的地盘。如果对象被销毁了，这个地盘必须加以重新利用。在 Java 中，堆更像一个堆栈，每分配一个新对象，它就往前移动一格。

Java 这样的实现方式会导致频繁的内存页面调度。为了避免这种情况的出现，Java 使用了垃圾回收器。垃圾回收器工作的时候，一边回收空间，一边讲堆中的对象排列紧凑。通过垃圾回收器对对象重新排列，实现了一种告诉的、有无限空间可供分配的堆模型。

Java 垃圾回收器依据的思想是：对任何活的对象，一定能追溯到其存活在堆栈或静态存储区之中的引用。那么，只要从堆栈或静态存储区之中的引用开始遍历，就能找到所有活的对象。

Java 垃圾回收器处理存活对象的方式为：先暂停程序的运行，将所有存活的对象从当前堆复制到另一个堆，没有复制的全都是垃圾。当对象被复制到新堆，它们是紧凑排列的。这就避免了内存页面调度频繁的发生。这种做法称为“停止-复制”。

如果 Java 虚拟机发现程序很少产生垃圾甚至不产生垃圾时，Java 会切换到另一种工作模式，叫做“标记-清扫”。“标记-清扫”的思路是：从堆栈和静态存储区出发，遍历所有引用，进而找出所有存活的对象。每当它找到一个存活对象，就会给对象设一个标记。当标记完所有存活的对象后，清理工作才开始进行。在清理过程中，没有标记的对象会被释放。这样一来，就不会有复制动作的发生，但是剩下的堆空间也会不连续。垃圾回收器要是希望得到连续空间，就要重新整理剩下的对象。“标记-清扫”工作必须在程序暂停的情况下才能进行。

5 访问权限控制

Java 有四个访问权限：public、protected、包访问权限 (没有关键词) 和 private。

5.1 包: 库单元

包内包含一组类，它们在单一的名字空间之下被组织在一起。使用 package 语句指定类的名字空间。例子如下：

```
package psd.mypackage;

public class MyClass
{
    // ...
}
```

当同一目录下其他文件想调用这个类时，需要使用 import 语句。例子如下：

```
import psd.mypackage.MyClass;

public class ImportedMyClass
{
    public static void main(String[] args)
    {
        MyClass m = new MyClass();
    }
}
```

如果两个文件不在同一目录下，就需要设置 CLASSPATH。Java 中的包名相当于路径名。当一个文件声明 package psd.mypackage 的时候，这个文件应该处于 CLASSPATH/psd/mypackage 的目录下。如果其他目录下的文件想要调用时，只有写 import psd.mypackage.*，那么这个文件就会搜寻 CLASSPATH/psd/mypackage 的目录下的所有文件。

5.2 Java 访问权限修饰词

5.2.1 包访问权限

当类或成员没有关键词时，对于同一包下的文件，对这个类或成员都有访问权限。而对于其他包的所有类，这个类或成员就是 private 的。

5.2.2 public 访问权限

当一个类被 public 修饰，那么就表明自己对其他人都是可用的。例子如下：

```
package access.dessert;
```

```
2
3   public class Cookie
4   {
5       public Cookie()
6       {
7           System.out.println("Hello world");
8       }
9       void bite()
10      {
11          System.out.println("bite");
12      }
13  }
```

Cookie 类是 public 的，所有其他文件可以使用它创建对象。需要注意的是，bite() 成员函数是具有包访问权限的，所以只有同一包的文件才能访问它。例子如下：

```
1   import access.dessert.*;
2
3   public class Dinner
4   {
5       public static void main(String[] args)
6       {
7           Cookie x = new Cookie();
8           // x.bite(); error!!!
9       }
10  }
```

5.2.3 private 访问权限

当一个成员被 private 修饰，那么这个成员只能在类中使用，和 C++ 的用法是一样的。例子如下：

```
1   class Sundae
2   {
3       private Sundae(){}
4       static Sundae makeASundae()
5       {
6           return new Sundae();
7       }
8   }
9
10  public class IceCream
11  {
12      public static void main(String[] args)
13      {
14          // Sundae x = new Sundae(); error!!!
15          Sundae x = Sundae.makeASundae();
16      }
17  }
```

5.2.4 protected 访问权限

protected 访问权限就是继承访问权限，和 C++ 的用法一样。例子如下：

```
package access.cookie2.*;

public class Cookie
{
    public Cookie()
    {
        System.out.println("Cookie constructor");
    }
    protected void bite()
    {
        System.out.println("bite");
    }
}
```

处于其他类继承这个类时，就可以访问 bite() 函数。例子如下：

```
import access.cookie2.Cookie;

public class Chocolate extends Cookie
{
    public Chocolate()
    {
        System.out.println("Chocolate constructor");
    }
    public void chomp()
    {
        bite();
    }
}
```

5.3 public 类

一个类只有两种访问权限：包访问权限和 public。只有一个类被修饰为 public，其他包下的类才可以创建该类的对象。与 public 类有关的限制如下：

- 每个文件中只能有一个 public 类。
- public 类的名称必须和文件名完全一样。

6 复用类

Java 中复用类的两种方式：

- 在新的类中产生现有类的对象，这种方法称为组合。
- 按照现有类的类型来创建新类，这种方法称为继承。

6.1 类的组合

组合技术很直观，只要将对象引用置于新类中即可。例子如下：

```
class WaterSource
{
    private String s;
    WaterSource()
    {
        System.out.println("WaterSource");
        s = "Constructed";
    }
}
```

6.2 类的继承

Java 中继承的语法和 C++ 类似，不过 Java 中使用关键字 extends 声明。如果继承基类，新类就会得到基类中所有非私有的域和成员函数。例子如下：

```
class Cleanser
{
    private String s = "Cleanser";
    public void append(String a)
    {
        s += a;
    }
    public static void main(String[] args)
    {
        Cleanser x = new Cleanser();
        x.append(" hello world");
    }
}

public class Detergent extends Cleanser
{
    public static void main(String[] args)
    {
        Detergent x = new Detergent();
        x.append(" hello world");
        Cleanser.main(args);
    }
```

```
23      }
```

6.2.1 基类的初始化

如果没有特别声明，将调用基类默认的构造器或者无参数构造器。如果想调用一个带参数的基类构造器，就必须使用 super 显式地调用基类构造器。例子如下：

```
1   class Game
2   {
3       Game(int i)
4       {
5           System.out.println("Hello World");
6       }
7   }
8
9   public class Chess extends Game
10  {
11      Chess()
12      {
13          super(1);
14          System.out.println("Chess constructor");
15      }
16      public static void main(String[] args)
17      {
18          Chess c = new Chess();
19      }
20  }
```

6.2.2 名称屏蔽

与 C++ 不同的是，Java 中导出类如果重载基类中的函数，并不会屏蔽其在基类中该函数的任何版本。例子如下：

```
1   class Homer
2   {
3       char doh(char c)
4       {
5           return c;
6       }
7       float doh(float c)
8       {
9           return c;
10      }
11  }
12
13  class Bart extends Homer
14  {
15      String doh(String s)
16      {
17          return s;
```

```
        }
    }
```

需要注意的是，因为这个语法特点，Java 中其实是没有名称屏蔽的。那么当我们要覆写基类中的一个函数时，很可能将其重载而非覆写。为了防止这个错误的发生，Java 提供了 @Override 注解相应的函数。如果这个函数是重载而非覆写时，编译器就会产生错误：

```
class Lisa extends Homer
{
    @Override
    String doh(String s)
    {
        return s; // 将产生错误
    }
}
```

6.2.3 向上转型

继承技术最重要的不是为新的类提供函数，而是用于表现新类和基类之间的关系。新类是现有类的一种类型。例子如下：

```
class Instrument
{
    public void play(){}
    static void tune(Instrument i)
    {
        i.play();
    }
}

public class Wind extends Instrument
{
    public static void main(String[] args)
    {
        Wind flute = new Wind();
        Instrument.tune(flute);
        // tune函数接受的是Instrument对象
        // 这里它也可以接受Wind对象
        // 因为Wind是Instrument的一种类型
    }
}
```

将导出类引用转换为基类引用的动作称为向上转型。在实现上看，导出类是基类的一个超集。在向上转型的过程中，导出类引用转换为基类引用，并且只保留基类拥有的方法。

导出类无法继承 private 函数。即使在导出类中以相同的名称声明一个函数，也不会覆盖基类中相应的 private 函数，而是生成了一个新的函数。当向上转型时，这个函数将会被丢弃。

6.2.4 继承技术的用途

相对于组合技术，继承技术不常用。只有需要从新类向基类进行向上转型，继承才是必要的。

6.3 final 关键字

final 关键字可以修饰数据、函数和类。

6.3.1 final 数据

Java 中使用 final 告知一块数据是恒定不变的，相当于 C 中的 const 关键字。需要知道的是，Java 中常量必须是基本数据类型。

一个既是 static 又是 final 的域只占据一段不能改变的存储空间。

当用 final 修饰对象引用时，这个引用将恒定不变。也就是说，引用一旦被初始化指向一个对象，就无法再把它改为指向另一个对象，而被引用的对象本身是可以被修改的。Java 没有提供使任何对象恒定不变的途径。

Java 允许生成空白 final。也就是这个域被 final 修饰但又没有赋初值。final 域在使用前必须被初始化。

在函数参数列表中将参数指明为 final，那么在函数中就无法修改参数引用所指向的对象。

6.3.2 final 函数

使用 final 函数的原因如下:

- 将函数锁定。以防任何继承类修改它的实现。
- 追求效率。当一个函数指明为 final，编译器就将该函数的所有调用都转为内嵌调用。这和 C++ 的 inline 关键字的作用一样。

类中 private 方法都隐式地指定为 final。

6.3.3 final 类

当将某个类的整体定义为 final，那么这个类就无法被继承。final 类中的域不一定是 final 的。

7 多态

多态又称为动态绑定，和 C++ 的多态类似。在讨论多态之前，先感受一下多态的特性。例子如下：

```
class Instrument
{
    public void play()
    {
        System.out.println("Instrument.play()");
    }
}

class Wind extends Instrument
{
    public void play()
    {
        System.out.println("Wind.play()");
    }
}

public class Music
{
    public static void tune(Instrument i)
    {
        i.play();
    }
    public static void main(String[] args)
    {
        Wind flute = new Wind();
        // tune接受Instrument类型
        // 将Wind转为Instrument类型
        // 输出的是: Wind.play()
        tune(flute);
    }
}
```

从这个例子可以看出一个多态的现象：虽然 tune 函数接受一个 Instrument 引用，但是它知道这个 Instrument 引用指向的是 Wind 对象。正是动态绑定实现了这项特性。

7.1 绑定

将一个函数调用和一个函数主体关联起来称为绑定。绑定有两种类型，如下：

- 前期绑定。在程序执行前就将一个函数调用和一个函数主体关联起来。
- 后期绑定，又称为动态绑定。在程序运行时根据对象的类型进行绑定。

Java 中除了 static 方法和 final 方法，其他所有方法都是后期绑定的。

7.2 多态的缺陷

7.2.1 private 函数无法动态绑定

程序不能对 private 函数进行动态绑定。这是因为 private 函数是 final 函数，而且导出类无法覆盖基类中的 private 函数。例子如下：

```
class PrivateOvrride
{
    private void f()
    {
        System.out.println("private f()");
    }

    public static void main(String[] args)
    {
        PrivateOvrride po = new Derived();
        po.f(); // 不会指向Derived类中的f()，而是指向PrivateOvrride类中的f()
    }
}

public class Derived extends PrivateOvrride
{
    public void f()
    {
        System.out.println("public f()");
    }
}
```

为了避免造成混乱的代码，导出类中的函数名不要和基类中的 private 函数名相同。

7.2.2 域无法动态绑定

和 C++ 一样，Java 中域是无法动态绑定的。也就是说，程序无法根据对象的类型选择相应的域。例子如下：

```
class Super
{
    public int field = 0;
}

Sub extends Super
{
    public int field = 1;
}

public class FieldAccess
{
    public static void main(String[] args)
    {
        Super sup = new Sub();
        System.out.println(sup.field); // 输出0
```

```
17              Sub sub = new Sub();
18              System.out.println(sub.field); // 输出1
19          }
20      }
```

为了避免造成混乱的代码，不要把基类中的域和导出类的域赋予相同的名字。

7.3 协变返回类型

协变返回类型表明，子类覆写基类方法时，返回的类型可以是基类方法返回类型的子类。

```
1   class Grain
2   {
3       public String toString()
4       {
5           return "Grain";
6       }
7   }
8
9   class Wheat extends Grain
10  {
11      public String toString()
12      {
13          return "Wheat";
14      }
15  }
16
17  class Mill
18  {
19      Grain process()
20      {
21          return new Grain();
22      }
23  }
24
25  class WheatMill extends Mill
26  {
27      Wheat process()
28      {
29          return new Wheat();
30      }
31  }
32
33  public class CovariantReturn
34  {
35      public static void main(String[] args)
36      {
37          Mill m = new Mill();
38          Grain g = m.process();
39          System.out.println(g); // 输出Grain
40          m = new WheatMill();
41          g = m.process();
```

```
42                 System.out.println(g); // 输出Wheat
43             }
44         }
```