内存之ioremap内存映射

# 1、概述

Linux内核驱动模块中常常使用ioremap对Soc/外设寄存器进行映射，以获取内核态虚拟地址。ioremap只有两个入参，一个是起始地址，一个是所要映射长度；相信很多驱动开发者对此都比较熟悉。接口使用上大家应该都不存在问题，但接口选择和接口实现上是否也比较清楚呢？若是对上面的问题存有疑虑，请往下看。

Note：本文是基于arm64架构平台进行的展开。

# 2、接口类型

在内核源码arch/arm64/include/asm/io.h中有如下几种ioremap接口

|  |
| --- |
| ioremap(addr, size) |
| ioremap\_nocache(addr, size) |
| ioremap\_wc(addr, size) |
| ioremap\_wt(addr, size) |
| ioremap\_cache(addr, size) |

从接口命名中大致可以区分开各自的功能，但为了真正明白还是要看下具体属性定义。

ioremap:

\_\_ioremap((addr), (size), \_\_pgprot(PROT\_DEVICE\_nGnRE))

ioremap\_nocache：

\_\_ioremap((addr), (size), \_\_pgprot(PROT\_DEVICE\_nGnRE))

ioremap\_wc：

\_\_ioremap((addr), (size), \_\_pgprot(PROT\_NORMAL\_NC))

ioremap\_wt：

\_\_ioremap((addr), (size), \_\_pgprot(PROT\_DEVICE\_nGnRE))

ioremap\_cache:

从上面的接口定义看，除了addr和size外还有一个\_\_pgprot(X)参数，上面接口中X有PROT\_DEVICE\_nGnRE和PROT\_NORMAL\_NC两种类型。实际在pgtable-prot.h中还有PROT\_DEVICE\_nGnRnE、PROT\_NORMAL\_WT和PROT\_NORMAL。ioremap\_接口的功能属性都是由\_\_pgprot(X)决定的，那X具体是什么含义呢？

## 2.1、内存属性

在内核源码memory.h中有定义ioremap接口中可用的内存类型：

/\*

\* Memory types available.

\*/

#define MT\_DEVICE\_nGnRnE 0

#define MT\_DEVICE\_nGnRE 1

#define MT\_DEVICE\_GRE 2

#define MT\_NORMAL\_NC 3

#define MT\_NORMAL 4

#define MT\_NORMAL\_WT 5

各字段具体含义可参照ARMv8手册中对内存属性的描述，内存可以分为DEVICE和NORMAL两大类型，Device memory依据是否可合并、无序分为：

|  |
| --- |
| Device-nGnRnE  Device non-Gathering, non-Reordering, No Early write acknowledgement.  Equivalent to the Strongly-ordered memory type in earlier versions of the architecture. |
| Device-nGnRE  Device non-Gathering, non-Reordering, Early Write Acknowledgement.  Equivalent to the Device memory type in earlier versions of the architecture. |
| Device-nGRE  Device non-Gathering, Reordering, Early Write Acknowledgement.  Armv8 adds this memory type to the translation table formats found in earlier versions of  the architecture. The use of barriers is required to order accesses to Device-nGRE memory. |
| **Device-GRE**  Device Gathering, Reordering, Early Write Acknowledgement  Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that Speculative accesses to Device-GRE memory is forbidden |

Normal memory依据是否可共享、是否可cache分为下面几类

|  |
| --- |
| **Shareable**  Shareable, Inner Shareable, and Outer Shareable Normal memory |
| **Non-shareable** |
| **Cacheability attributes** |

从上面的描述可见，只有Normal memory具备共享和cache属性。

## 2.2、ioremap涉及到的页表属性

上面ioremap\_接口中用到的属性参数主要是PROT\_DEVICE\_nGnRE和PROT\_NORMAL\_NC；下面来看下这两个prot\_val分别代表什么含义。内容摘自pgtable-prot.h

|  |
| --- |
| #define PROT\_DEVICE\_nGnRE (PROT\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_WRITE | PTE\_ATTRINDX(MT\_DEVICE\_nGnRE)) |
| #define PROT\_NORMAL\_NC (PROT\_DEFAULT | PTE\_PXN | PTE\_UXN | PTE\_WRITE | PTE\_ATTRINDX(MT\_NORMAL\_NC)) |

Bit字段拆解

|  |  |
| --- | --- |
| PROT\_DEFAULT | PTE\_TYPE\_PAGE | PTE\_AF | PTE\_SHARED |
| #define PTE\_TYPE\_PAGE (\_AT(pteval\_t, 3) << 0) |
| #define PTE\_SHARED (\_AT(pteval\_t, 3) << 8) |
| #define PTE\_AF (\_AT(pteval\_t, 1) << 10) |
| PTE\_PXN | (\_AT(pteval\_t, 1) << 53) /\* Privileged XN \*/ |
| PTE\_UXN | (\_AT(pteval\_t, 1) << 54) /\* User XN \*/ |
| PTE\_WRITE | (\_AT(pteval\_t, 1) << 51) /\* Dirty Bit Management \*/ |
| PTE\_ATTRINDX(t) | (\_AT(pteval\_t, (t)) << 2) |
| MT\_DEVICE\_nGnRnE： (\_AT(pteval\_t, (0)) << 2) |
| MT\_NORMAL\_NC ： (\_AT(pteval\_t, (3)) << 2) |
| PROT\_DEVICE\_nGnRE =  3 << 0 | 1 << 10 | 3 << 8 | 1 << 53 | 1 << 54 | 1 << 55 | 1 << 51 | 1 << 2 | |
| PROT\_NORMAL\_NC =  3 << 0 | 1 << 10 | 3 << 8 | 1 << 53 | 1 << 54 | 1 << 55 | 1 << 51 | 3 << 2 | |

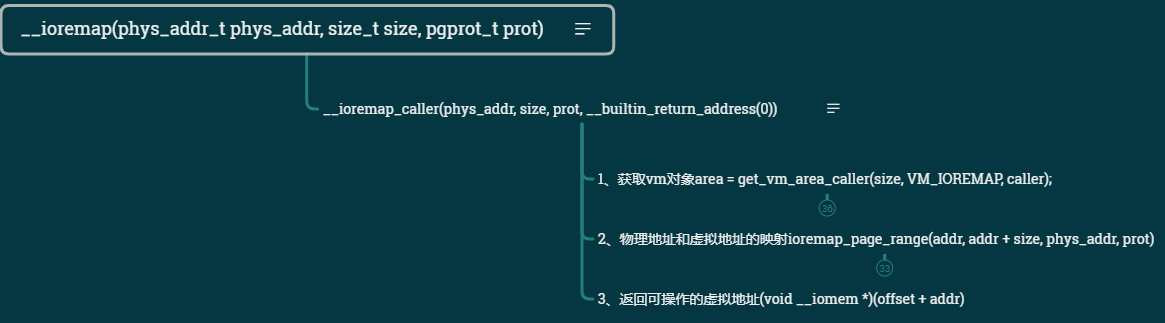
具体每个bit的含义，我们在内存之页表章节中有描述，此处不再重复。

# 3、接口实现

## 3.1、整体结构

从流程结构上看ioremap的实现主要有两步：

1. 获取一段合适的虚拟地址空间，使用get\_vm\_area\_caller接口；
2. 进行物理地址和虚拟地址的映射，使用ioremap\_page\_range接口；



虽然看似只有两步就完成了整体功能，但在每一步的里面又包含了许多的设计与接口调用。另外，ioremap接口的内部设计都是按照页对齐来设计的，因此在上图中第一步之前有一些对phys\_addr和size进行页对齐的操作，此处不做详细描述。

Ioremap接口映射的虚拟地址处在VMALLOC区，这点是由接口设计实现决定的；在使用类似usb设备的dma功能时要注意地址类型，因为有些设计会判断地址若是处在vmalloc\_start~vmalloc\_end则判定其没有连续的物理地址而直接异常退出。个人理解，以连续的物理地址进行ioremap接口进行映射，所得到的结果是虚拟连续、物理也连续；但并不通用，不能从任何字段或标识上判定地址的映射方式，因此在dma相关使用中将其排除在外是合情合理的。

Get\_vm\_area\_caller前后涉及两个数据结构vm\_struct和vmap\_area；在最外层联通上下文的是vm\_struct因为其是内核态虚拟地址管理的数据结构，而vmap\_area仅是vmalloc区内存申请分配的管理单元。该接口的核心功能是找到一个还未使过用的大小合适hole。

Ioremap\_page\_range是将已确定的虚拟地址和物理地址进行页表的映射，这和我们在业务程序中使用malloc申请内存页表建立过程不同(malloc流程是内核先提供一段虚拟地址，在用户实际读写时通过page\_fault触发为其分配真正的物理内存，建立页表映射)。使用的平台内核配置为39bit虚拟地址、三级页表。

## 3.2、核心接口实现

下面分别仔细看下两个核心接口get\_vm\_area\_caller和ioremap\_page\_range的具体实现。

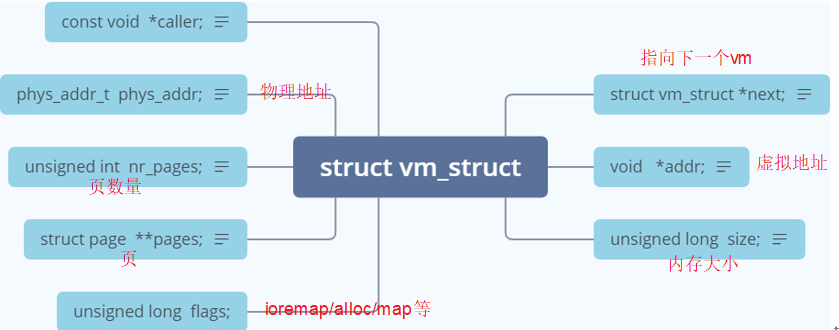
### 3.2.1、get\_vm\_area\_caller

get\_vm\_area\_caller的大体流程结构如下图所示：

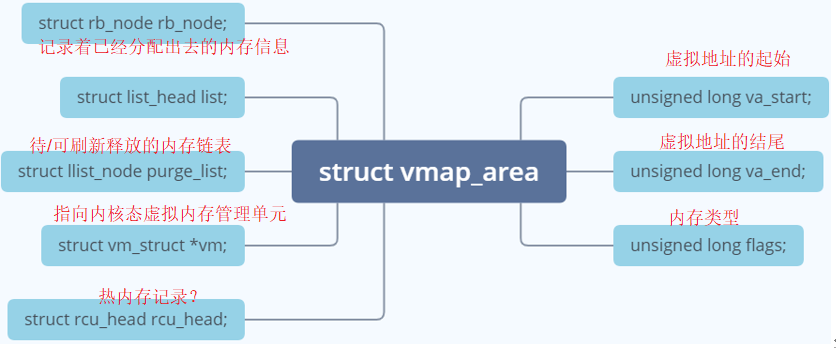


此处涉及两个主要数据结构：struct vm\_struct和struct vmap\_area。前文也有提到vm\_struct是内核态用来管理虚拟地址的；vmap\_area是内核态管理vmalloc内存区虚拟地址的。

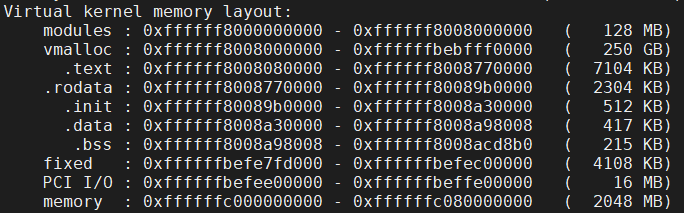
Struct vm\_struct的成员如下：



Strcut vmap\_area的成员如下：



在设备启动过程中一般都会打印出当前设备内核虚拟内存地址的分布情况，下面是nt98336平台的数据：



kernel space 中常用的内存分配接口kmalloc、kzalloc、get\_free\_pages获取的地址位于memory区；vmalloc接口申请的地址位于vmalloc区，其获取的地址范围位于：VMALLOC\_START~VMALLOC\_END。

get\_free\_pages验证下其对应的虚拟地址位于哪个区呢？理论上应该是memory区才是。搞个小测试程序验证下。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。

在整个get\_vm\_area\_caller接口中vmap\_area是核心，还是先看下大概流程再研究细节。从下图看到整个流程由几个分支构成：

1. 当struct rb\_node free\_vmap\_cache不为空时，获取该节点上vmap\_area first；
2. 当free\_vmap\_cache为空时，获取满足条件的的vmap\_area first；
3. 遍历以first为索引的list上记录的vmap\_area 节点，寻找满足条件的hole；
4. Found时将对应的va进行插入操作并返回对应的vmap\_area 地址；
5. 历经上述步骤没有满足条件的地址时，执行lazy\_area区域内存的释放，再次从头来过；



各分支代码流程

------------------------------------------------------------------------

1、free\_vmap\_cache不为空时

if (free\_vmap\_cache)

{

/\*以rb\_node为成员，获取free\_vmap\_cache中的vmap\_area\*/

first = rb\_entry(free\_vmap\_cache, struct vmap\_area, rb\_node);

addr = ALIGN(first->va\_end, align);

if (addr < vstart)

goto nocache; //free\_vmap\_cache中记录的地址不满足当前需求

if (addr + size < addr)

goto overflow;

}

2、free\_vmap\_cache为空时

addr = ALIGN(vstart, align); /\*其实就是VMALLOC\_START\*/

if (addr + size < addr)

goto overflow;

n = vmap\_area\_root.rb\_node; /\*vmap\_area rb\_root根节点\*/

first = NULL;

while (n)

{

struct vmap\_area \*tmp;

/\*分配时是从小地址开始往上增长，记录在前的自然是地址大的\*/

tmp = rb\_entry(n, struct vmap\_area, rb\_node);

if (tmp->va\_end >= addr)

{

first = tmp;

if (tmp->va\_start <= addr)

break;

n = n->rb\_left; /\*tmp变量结束地址大于addr时 取左节点\*/

}

else

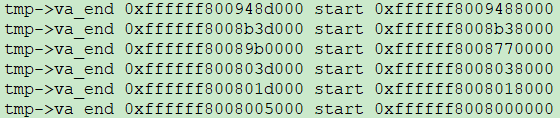
n = n->rb\_right; /\*相反取右值\*/

}

if (!first) /\*若rb\_root为空，说明从未有内存分配过，直接使用vmalloc\_start地址\*/

goto found;

测试数据如下：



{

PS：从上图应该可以管中窥豹大致看清虚拟内存分配的规则，从对应的区域由小到大逐次使用，当然里面是有空洞的，应有地址对齐和大小是否满足的需要

}

1. 从找到的struct vmap\_area \*first对应的链表上找满足条件的hole

while (addr + size > first->va\_start && addr + size <= vend)

{

if (addr + cached\_hole\_size < first->va\_start)

cached\_hole\_size = first->va\_start - addr;

addr = ALIGN(first->va\_end, align);

if (addr + size < addr)

goto overflow;

if (list\_is\_last(&first->list, &vmap\_area\_list))

goto found; /\*取该链表上最后一个元素\*/

first = list\_next\_entry(first, list);

}

1. found流程

if (addr + size > vend || addr < vstart)

goto overflow;

va->va\_start = addr; /\*使用找到的虚拟地址填充va\*/

va->va\_end = addr + size;

va->flags = 0;

/\*将va->rb\_node添加到vmap\_area\_root根节点，将va->list添加到list中\*/

\_\_insert\_vmap\_area(va);

/\*以最新使用的va对free\_vmap\_cache进行赋值，提高系统中同样请求的处理效率\*/

free\_vmap\_cache = &va->rb\_node;

spin\_unlock(&vmap\_area\_lock);

return va;

1. lazy area释放

spin\_unlock(&vmap\_area\_lock);

if (!purged) {

purge\_vmap\_area\_lazy(); /\*主要是处理vmap\_purge\_list中的要素\*/

purged = 1;

goto retry;

}

if (gfpflags\_allow\_blocking(gfp\_mask)) {

unsigned long freed = 0;

blocking\_notifier\_call\_chain(&vmap\_notify\_list, 0, &freed);

if (freed > 0) {

purged = 0;

goto retry;

}

}

if (!(gfp\_mask & \_\_GFP\_NOWARN) && printk\_ratelimit())

pr\_warn("vmap allocation for size %lu failed: use vmalloc=<size> to increase size\n",

size);

kfree(va);

return ERR\_PTR(-EBUSY);

虚拟地址的寻找流程大致如此，insert和purge的流程并未展开分析，感兴趣的读者可以再剖析下。

3.2、vm\_struct 与 vmap\_area关系的建立

setup\_vmalloc\_vm(area, va, flags, caller);

static void setup\_vmalloc\_vm(struct vm\_struct \*vm, struct vmap\_area \*va,

unsigned long flags, const void \*caller)

{

spin\_lock(&vmap\_area\_lock);

vm->flags = flags;

vm->addr = (void \*)va->va\_start;

vm->size = va->va\_end - va->va\_start;

vm->caller = caller;

va->vm = vm;

va->flags |= VM\_VM\_AREA;

spin\_unlock(&vmap\_area\_lock);

}

3.3、页表映射

只有将虚拟地址和物理地址进行了映射后，CPU才能有效使用获取的地址进行一系列操作。映射关系的建立使用ioremap\_page\_range接口。



从图中看映射流程貌似使用了五级页表：pgd--->p4d--->pud--->pmd--->pte，实际上只有三层(pgd(p4d/pud)->pmd->pte)，在内核config配置中可找到明确参数；代码中如此设计仅是为了兼容。

该接口的入参有：addr(分配得到的虚拟地址的起始)、addr + size(分配得到的虚拟地址的结尾)、phys\_addr(物理地址的起始)和prot(属性)；个人理解页表映射一是将虚拟地址和物理地址建立起一定的对应关系；二是将预设的属性信息记录起来。

下面一点一点的来看具体的操作流程：

int ioremap\_page\_range(unsigned long addr,

unsigned long end, phys\_addr\_t phys\_addr, pgprot\_t prot)

{

pgd\_t \*pgd;

unsigned long start;

unsigned long next;

int err;

might\_sleep();

BUG\_ON(addr >= end);

start = addr;

phys\_addr -= addr;

pgd = pgd\_offset\_k(addr); /\*查找内核空间addr对应的页目录项pgd条目，具体定义见附录1\*/

do {

next = pgd\_addr\_end(addr, end); /\*获取addr对应的下一个边界，具体定义见附录2\*/

err = ioremap\_p4d\_range(pgd, addr, next, phys\_addr+addr, prot); /\*\*/

if (err)

break;

} while (pgd++, addr = next, addr != end); /\*do…while仅执行一次\*/

flush\_cache\_vmap(start, end); /\*空函数，aarch64不需要，PIPT or VIPT non-aliasing D-cache\*/

return err;

}

static inline int ioremap\_p4d\_range(pgd\_t \*pgd, unsigned long addr,

unsigned long end, phys\_addr\_t phys\_addr, pgprot\_t prot)

{

p4d\_t \*p4d;

unsigned long next;

phys\_addr -= addr;

p4d = p4d\_alloc(&init\_mm, pgd, addr);

if (!p4d)

return -ENOMEM;

do {

next = p4d\_addr\_end(addr, end); /\*获取addr对应的下一个边界，具体定义见附录3\*/

if (ioremap\_p4d\_enabled() &&

((next - addr) == P4D\_SIZE) &&

IS\_ALIGNED(phys\_addr + addr, P4D\_SIZE)) {

if (p4d\_set\_huge(p4d, phys\_addr + addr, prot)) /\*空函数\*/

continue;

}

if (ioremap\_pud\_range(p4d, addr, next, phys\_addr + addr, prot))

return -ENOMEM;

} while (p4d++, addr = next, addr != end);

return 0;

}

static inline int ioremap\_pud\_range(p4d\_t \*p4d, unsigned long addr,

unsigned long end, phys\_addr\_t phys\_addr, pgprot\_t prot)

{

pud\_t \*pud;

unsigned long next;

phys\_addr -= addr;

pud = pud\_alloc(&init\_mm, p4d, addr);

if (!pud)

return -ENOMEM;

do {

next = pud\_addr\_end(addr, end);

if (ioremap\_pud\_enabled() &&

((next - addr) == PUD\_SIZE) &&

IS\_ALIGNED(phys\_addr + addr, PUD\_SIZE) &&

pud\_free\_pmd\_page(pud, addr)) {

if (pud\_set\_huge(pud, phys\_addr + addr, prot))

continue;

}

if (ioremap\_pmd\_range(pud, addr, next, phys\_addr + addr, prot))

return -ENOMEM;

} while (pud++, addr = next, addr != end);

return 0;

}

static inline int ioremap\_pmd\_range(pud\_t \*pud, unsigned long addr,

unsigned long end, phys\_addr\_t phys\_addr, pgprot\_t prot)

{

pmd\_t \*pmd;

unsigned long next;

phys\_addr -= addr;

pmd = pmd\_alloc(&init\_mm, pud, addr);

if (!pmd)

return -ENOMEM;

do {

next = pmd\_addr\_end(addr, end);

if (ioremap\_pmd\_enabled() &&

((next - addr) == PMD\_SIZE) &&

IS\_ALIGNED(phys\_addr + addr, PMD\_SIZE) &&

pmd\_free\_pte\_page(pmd, addr))

{

if (pmd\_set\_huge(pmd, phys\_addr + addr, prot))

continue;

}

if (ioremap\_pte\_range(pmd, addr, next, phys\_addr + addr, prot))

return -ENOMEM;

} while (pmd++, addr = next, addr != end);

return 0;

}

static int ioremap\_pte\_range(pmd\_t \*pmd, unsigned long addr,

unsigned long end, phys\_addr\_t phys\_addr, pgprot\_t prot)

{

pte\_t \*pte;

u64 pfn;

pfn = phys\_addr >> PAGE\_SHIFT; /\*获取物理地址对应的页帧号\*/

pte = pte\_alloc\_kernel(pmd, addr); /\*分配内存，也可能是查找已分配过的内存地址\*/

if (!pte)

return -ENOMEM;

do {

BUG\_ON(!pte\_none(\*pte));

set\_pte\_at(&init\_mm, addr, pte, pfn\_pte(pfn, prot)); /\*对pte内容赋值\*/

pfn++;

} while (pte++, addr += PAGE\_SIZE, addr != end);

return 0;

}

static inline void set\_pte\_at(struct mm\_struct \*mm, unsigned long addr,

pte\_t \*ptep, pte\_t pte)

{

pte\_t old\_pte;

/\*判断pte是否在内存中、用户态可执行、special?\*/

if (pte\_present(pte) && pte\_user\_exec(pte) && !pte\_special(pte))

\_\_sync\_icache\_dcache(pte);

/\*

\* If the existing pte is valid, check for potential race with

\* hardware updates of the pte (ptep\_set\_access\_flags safely changes

\* valid ptes without going through an invalid entry).

\*/

old\_pte = READ\_ONCE(\*ptep);

if (IS\_ENABLED(CONFIG\_DEBUG\_VM) && pte\_valid(old\_pte) && pte\_valid(pte) &&

(mm == current->active\_mm || atomic\_read(&mm->mm\_users) > 1)) {

VM\_WARN\_ONCE(!pte\_young(pte),

"%s: racy access flag clearing: 0x%016llx -> 0x%016llx",

\_\_func\_\_, pte\_val(old\_pte), pte\_val(pte));

VM\_WARN\_ONCE(pte\_write(old\_pte) && !pte\_dirty(pte),

"%s: racy dirty state clearing: 0x%016llx -> 0x%016llx",

\_\_func\_\_, pte\_val(old\_pte), pte\_val(pte));

}

set\_pte(ptep, pte);

}

static inline void set\_pte(pte\_t \*ptep, pte\_t pte)

{

WRITE\_ONCE(\*ptep, pte);

/\*

\* Only if the new pte is valid and kernel, otherwise TLB maintenance

\* or update\_mmu\_cache() have the necessary barriers.

\*/

if (pte\_valid\_not\_user(pte)) {

dsb(ishst);

isb();

}

}

附录1

/\* to find an entry in a page-table-directory \*/

#define pgd\_index(addr) (((addr) >> PGDIR\_SHIFT) & (PTRS\_PER\_PGD - 1))

#define pgd\_offset\_raw(pgd, addr) ((pgd) + pgd\_index(addr))

#define pgd\_offset(mm, addr) (pgd\_offset\_raw((mm)->pgd, (addr)))

/\* to find an entry in a kernel page-table-directory \*/

#define pgd\_offset\_k(addr) pgd\_offset(&init\_mm, addr)

附录2

#define pgd\_addr\_end(addr, end) \

({ unsigned long \_\_boundary = ((addr) + PGDIR\_SIZE) & PGDIR\_MASK; \

(\_\_boundary - 1 < (end) - 1)? \_\_boundary: (end); \

})

附录3