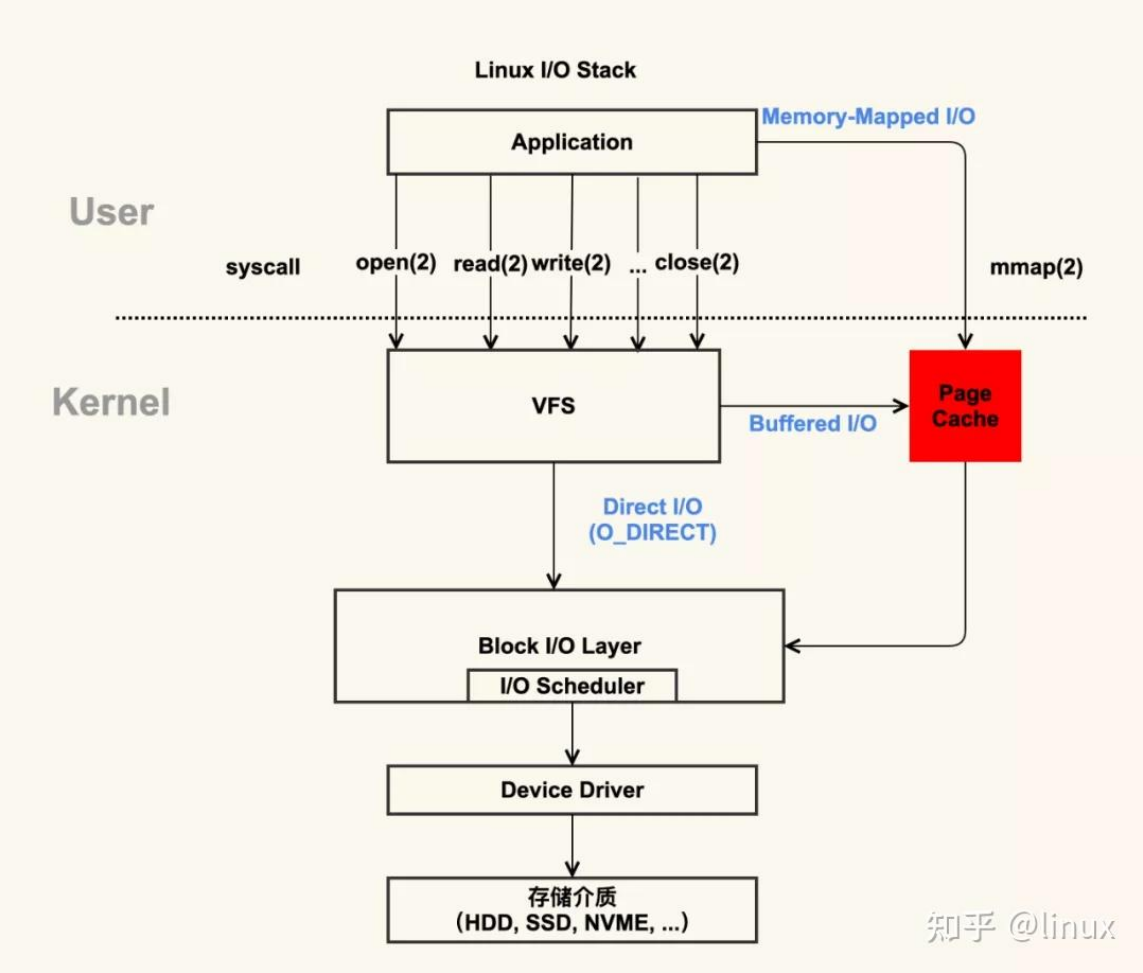
Linux 页缓存模块

在学习页缓存之前，我们先通过 Linux 文件系统 I/O 系统来了解页缓存处在哪个模块。Page Cache 本质上是 Linux 内存管理的内存区域。通过 mmap 内存映射以及 buffered I/O 都会经过页缓存的部分，如果用户在进行系统调用的时候，传入了 `O_DIRECT` 这个 flag 的话，就不会经过页缓存的部分。

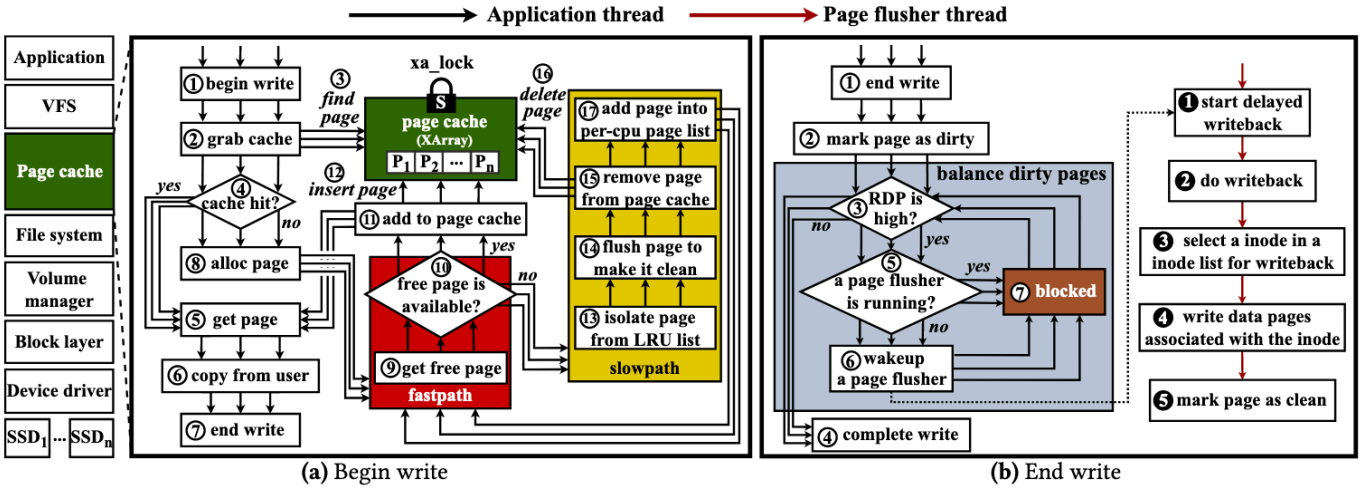


Linux 系统上可供用户访问的页面可以分为两个类型：

1. File-backed pages：文件备份页也就是 Page Cache 中的 page，对应于磁盘上的若干数据块；对于这些页最大的问题是脏页回盘。
2. Anonymous pages：匿名页不对应磁盘上的任何磁盘数据块，它们是进程的运行是内存空间（例如方法栈、局部变量表等属性）。

1. 页面分配

当进程发现访问的数据不在内存的时候，操作系统会将页面加载到内存当中。这个过程成为缺页中断 `filemap_fault`。例如：应用程序通过系统调用 write() 对文件进行写，我们在没有用 O_DIRECT 的方式打开这个文件的情况下，这个时候是需要经过页缓存的部分的。它会经过文件系统的缺页处理函数 `ext4_filemap_fault`（这里是 ext4 文件系统）。如何会进入到内核的 `filemap_fault`。下面是页缓存的内部示意图，下面进行分析。



(a) Begin write (b) End write

Figure 2. Procedure of existing page reclamation and flushing (S: spinlock, RDP: ratio of dirty pages).

如图 2 所示，这里展示更多的细节，是如何一步一步的往下执行的。

1.1 缺页中断

前面提到，当进程发现访问的数据不在内存的时候，会将页面加载到内存，但是这里是如何做到的呢？一旦开始访问虚拟内存的某个地址，如果没有对应的物理页面，会进入到页错误处理函数 `do_page_fault` 当中。

```
dotraplinkage void notrace
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    unsigned long address = read_cr2(); /* Get the faulting address */
    // ....
    __do_page_fault(regs, error_code, address);
}
```

在 `__do_page_fault` 当中，有两种情况，一种是地址错误是发送在内核空间，另一种是发送在用户空间。而这里使用了一个优化 `unlikely`，表明大多数情况下是发送在用户空间的地址错误，内核空间的地址错误，通常与内核编写错误有关，而且一旦发送这种错误会导致内核崩溃。

```
static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long hw_error_code,
    unsigned long address)
{
       // ....
    if (unlikely(fault_in_kernel_space(address)))
      do_kern_addr_fault(regs, hw_error_code, address);
    else
      do_user_addr_fault(regs, hw_error_code, address);
}
```

紧接着，在 do_user_addr_fault 当中，会先通过 find_vma(mm, address) 查找与触发错误地址相关的虚拟内存区域。每个进程的虚拟内存空间可以被划分成若干区域，每个区域都成为虚拟内存区域(VMA)，一个 VMA 是一块连续的地址空间，也拥有自身的权限(可读、可写、可执行)，在内核当中的结构是 vm_area_struct。其次，如果访问的地址属于文件映射的区域，而文件的页面没有被加载到内存，操作系统会通过 handle_mm_fault 来尝试处理这个页错误。

```
void do_user_addr_fault(struct pt_regs *regs,
            unsigned long hw_error_code,
            unsigned long address)
{
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct mm_struct *mm;
    vm_fault_t fault, major = 0;

    vma = find_vma(mm, address);

    fault = handle_mm_fault(vma, address, flags);
}
```

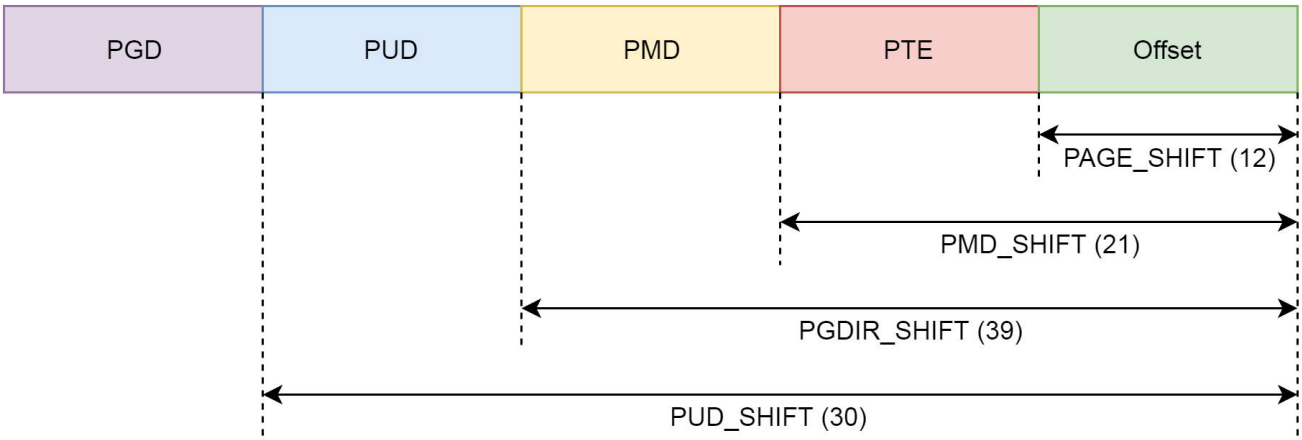
如果虚拟内存区域采用的是大页的方式，会调用专门的 hugetlb_fault ，否则调用 __handle_mm_fault 来处理页错误。

```
vm_fault_t handle_mm_fault(struct vm_area_struct *vma, unsigned long address,
        unsigned int flags)
{
    vm_fault_t ret;

    // ....
    if (unlikely(is_vm_hugetlb_page(vma)))
        ret = hugetlb_fault(vma->vm_mm, vma, address, flags);
    else
        ret = __handle_mm_fault(vma, address, flags);

    return ret;
}
```

我们先不考虑大页的情况，`__handle_mm_fault` 是处理页错误的细节，这里涉及到 Linux 的多级页表结构来查找或创建一个对应的物理页面，并将其映射到虚拟地址空间中。Linux 采用的五级页表结构(PGD -> P4D -> PUD -> PMD -> PTE)，但是如果是在四级页表的情况下，p4d 相关请求会被内核代码直接忽略，相当于这一部分做了特殊处理，这里介绍一下给出四级页表的细节图。




```
static vm_fault_t __handle_mm_fault(struct vm_area_struct *vma,
        unsigned long address, unsigned int flags)
{
    // 初始化 vm_fault 结构体，报错与页错误相关的上下文信息
    struct vm_fault vmf = {
        .vma = vma,
        .address = address & PAGE_MASK,
        .flags = flags,
        .pgoff = linear_page_index(vma, address),
```

```
        .gfp_mask = __get_fault_gfp_mask(vma),
    };
    unsigned int dirty = flags & FAULT_FLAG_WRITE;
    struct mm_struct *mm = vma->vm_mm;
    pgd_t *pgd;
    p4d_t *p4d;
    vm_fault_t ret;

    // 如果是四级页表的情况下，相当于 p4d_alloc 是被忽略的
    pgd = pgd_offset(mm, address);
    p4d = p4d_alloc(mm, pgd, address);
    if (!p4d)
        return VM_FAULT_OOM;

    vmf.pud = pud_alloc(mm, p4d, address);
    if (!vmf.pud)
        return VM_FAULT_OOM;



    vmf.pmd = pmd_alloc(mm, vmf.pud, address);
    if (!vmf.pmd)
        return VM_FAULT_OOM;

    return handle_pte_fault(&vmf);
}
```

最终，会通过 handle_pte_fault 来处理页错误，在其中，当页表项会空的时候，表面需要分配一个新的页面，对于匿名页面来说，则会调用 do_anonymous_page 来处理，否则调用 do_fault 来处理。

```c
static vm_fault_t handle_pte_fault(struct vm_fault *vmf)
{
    // ....
    if (!vmf->pte) {
        if (vma_is_anonymous(vmf->vma))
            return do_anonymous_page(vmf);
        else
            return do_fault(vmf);
    }

    // ....
    return 0;
}
```

紧接着，do_fault 里面会访问文件系统所设置好的 fault 函数。在 ext4 文件系统模块被加载的时候，会注册一系列的 VFS 操作，其中会设置一系列的文件操作，当用户进程将文件映射到内存的时候， VFS 会调用其中的 mmap 方法 ext4_file_mmap ，它会设置对应的 vm_operations_struct 结构，下面是其实现方式。如果我们在打开文件的时候传入 O_DIRECT 的 flag 的情况下，那么 IS_DAX(file_inode(file)) == true，我们这里考虑的是使用页缓存的情况。

```c
static int ext4_file_mmap(struct file *file, struct vm_area_struct *vma)
{
    struct inode *inode = file->f_mapping->host;

    // DAX 直接访问，表示绕过页缓存，允许文件直接映射到物理内存
    if (!IS_DAX(file_inode(file)) && (vma->vm_flags & VM_SYNC))
        return -EOPNOTSUPP;

    file_accessed(file);
    if (IS_DAX(file_inode(file))) {
        vma->vm_ops = &ext4_dax_vm_ops;
        vma->vm_flags |= VM_HUGEPAGE;
    } else {
        vma->vm_ops = &ext4_file_vm_ops;
    }
    return 0;
}
```

```
static const struct vm_operations_struct ext4_file_vm_ops = {
    .fault      = ext4_filemap_fault,
    .map_pages  = filemap_map_pages,
    .page_mkwrite   = ext4_page_mkwrite,
};
```

回到 do_fault当中，它会检查 vma->vm_ops->fault 是否存在，然后调用处理特定文件系统的页错误。我们先不考虑 fault函数不存在的情况，下面会根据这次错误请求的标志来调用相应的操作，如果页错误是读取操作，会调用 do_read_fault；如果页错误是写操作，则会调用 do_cow_fault 来进行写时复制。COW 会在写入的时候创建一个新的页面副本，避免修改原始页面。其他情况其实指的是共享操作，会调用 do_shared_fault 处理。

```
static vm_fault_t do_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    struct mm_struct *vm_mm = vma->vm_mm;
    vm_fault_t ret;

    if (!vma->vm_ops->fault) {
       // ...
    } else if (!(vmf->flags & FAULT_FLAG_WRITE))
        ret = do_read_fault(vmf);
    else if (!(vma->vm_flags & VM_SHARED))
        ret = do_cow_fault(vmf);
    else
        ret = do_shared_fault(vmf);

    return ret;
}
```

所有的前面提到的这些函数当中，最终都会调用 __do_fault，来处理具体的页错误。下面分别进行分析。

- 读取操作(初次)。当进程第一次访问文件页面的时候，会从文件系统中读取数据到内存。

```c
static vm_fault_t do_read_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    vm_fault_t ret = 0;

    ret = __do_fault(vmf);

    ret |= finish_fault(vmf);

    return ret;
}
```

- 写入操作。当进程尝试对这个页面进行写入操作的时候，无论是不是第一次写入的时候，都会去分配 COW 页面，放置在 `vmf->cow_page` 当中。

```c
static vm_fault_t do_cow_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    vm_fault_t ret;

    vmf->cow_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, vmf->address);

    ret = __do_fault(vmf);

    if (ret & VM_FAULT_DONE_COW)
        return ret;

    // 将原始页面的内容复制到新分配的 COW 页面
    copy_user_highpage(vmf->cow_page, vmf->page, vmf->address, vma);

    ret |= finish_fault(vmf);

    return ret;
}
```

- 共享操作。当一个进程尝试对共享映射的文件页面进行写操作时，内核需要确保多个进程可以安全地共享该页面，并且在必要时通知文件系统或设备驱动程序，以便它们可以采取适当的措施。

```c
static vm_fault_t do_shared_fault(struct vm_fault *vmf)
```

```
{
    struct vm_area_struct *vma = vmf->vma;
    vm_fault_t ret, tmp;

    ret = __do_fault(vmf);

         // 通知文件系统或设备驱动程序，页面即将变为可写状态。
    //  可以允许文件系统采取一些必要措施。
    if (vma->vm_ops->page_mkwrite) {
        tmp = do_page_mkwrite(vmf);
    }

    ret |= finish_fault(vmf);

    // 将共享映射的文件页面标记为脏，并确保文件系统知道页面内容已被修改。
    fault_dirty_shared_page(vma, vmf->page);
    return ret;
}
```

可以看到在进行具体的页错误处理之后，所有函数都会调用 `finish_fault` 来完成最终的页错误处理。它确保页面正确地映射到进程的虚拟地址空间，例如插入 PTE 以及将页面添加到 LRU 列表中。

```
vm_fault_t finish_fault(struct vm_fault *vmf)
{
    struct page *page;
    vm_fault_t ret = 0;

    // 判断是否是 COW 页面，这里与前面的对应上了
    if ((vmf->flags & FAULT_FLAG_WRITE) &&
        !(vmf->vma->vm_flags & VM_SHARED))
        page = vmf->cow_page;
    else
        page = vmf->page;

    // 将页面映射到虚拟地址空间当中，将创建的 PTE 条目插入到页表中。并处理内存控制组
 (memcg) 和 LRU 管理
    if (!ret)
        ret = alloc_set_pte(vmf, vmf->memcg, page);
```

```
    return ret;
}
```

OK，我们再次回到 __do_fault 上面，这时候就会调用文件系统提供的 fault函数，这里是ext4，所以会调用 ext4_filemap_fault。

```
static vm_fault_t __do_fault(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    vm_fault_t ret;

    // 调用文件系统对应的 fault 函数
    ret = vma->vm_ops->fault(vmf);

    return ret;
}
```

在 ext4_filemap_fault 当中，最后也会直接调用 filemap_fault。缺页中断部分就已经完成了，接下来是从页缓存当中查找一个页面。

```
vm_fault_t ext4_filemap_fault(struct vm_fault *vmf)
{
    vm_fault_t ret;

    ret = filemap_fault(vmf);

    return ret;
}
```

1.2 查找页面

在 filemap_fault，首先会尝试查找页面 find_get_page，然后在页面缓存当中查找页面 pagecache_get_page。记住这里初次传入的 fgp_flags 和 gfp_mask都是为 0，相当于没有设置任何有关的标志和掩码。

```
static inline struct page *find_get_page(struct address_space *mapping,
                    pgoff_t offset)
{
    return pagecache_get_page(mapping, offset, 0, 0);
}
```

这里先简单的介绍一下 FGP 标志，用于修改 `pagecache_get_page` 的行为的，具体标志如下：

标志位	作用	实现
FGP_ACCESSED	找到的页面会被标记已访问，会影响页面在 LRU 的位置，会被认为是"活跃"的。	`mark_page_accessed(page)`
FGP_LOCK	找到的页面将被锁定。锁定的页面不能被其他线程或进程修改，直到解锁为止。	`lock_page(page)`
FGP_CREAT	页面不在页面缓存中，则会分配一个新的页面并将其添加到页面缓存中。新页面将被锁定并增加引用计数。	`__page_cache_alloc(gfp_mask)`
FGP_FOR_MMAP	专用于内存映射（mmap）场景。它允许调用者在页面已经存在于缓存中时进行自己的锁定操作，或者在新页面添加到页面缓存后解锁页面。	如果页面已经存在于缓存中，调用者可以自己决定是否锁定页面；如果页面是新创建的，则在添加到页面缓存后解锁页面。
FGP_NOWAIT	与 `FGP_LOCK` 一起使用时，表示尝试非阻塞地锁定页面。如果无法立即锁定页面，则返回 `NULL`，而不是等待。	`trylock_page(page)`
FGP_WRITE	如果设置了此标志，并且文件系统支持脏页面记账（account dirty），则分配页面时会设置 `__GFP_WRITE` 标志，确保页面可以被标记为脏页面	`mapping_cap_account_dirty(mapping)`
FGP_NOFS	则在分配页面时禁止调用文件系统的操作（如分配磁盘空间）。	通过清除 `gfp_mask` 中的 `__GFP_FS` 标志来禁止文件系统操作。

1.2.1 xArray

我们从图 2 可以看到，实际上我们的页缓存是与一个结构体 `xArray` 有关，所以这里也简单解读一下这个结构体。每个文件都有一个自己的地址空间 `struct address_space`，其中一个字段 `i_pages` 就是该类型的，用来保存当前文件所管理的缓存页面。它是基于原先的 radix tree 进行修改的，因此结构上看起来像一棵树，但是逻辑上类似于一个非常大的指针数组，可以通过索引快速访问对象。

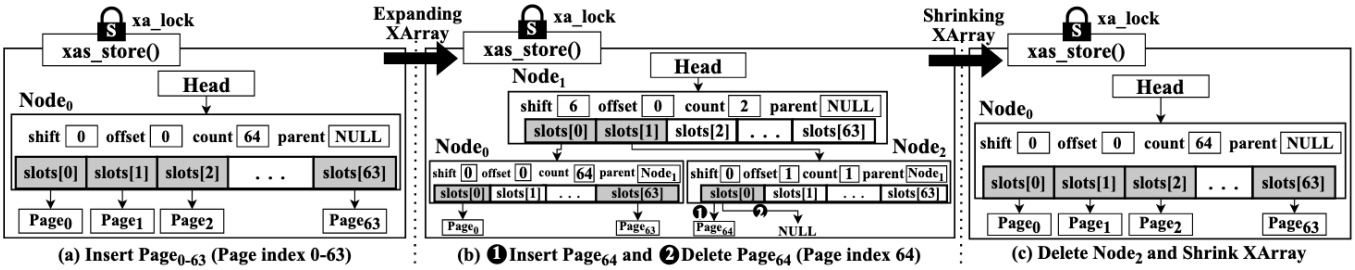


Figure 3. XArray structure and its operations. (a) and (c) denote one depth of XArray and (b) denotes two depths of XArray.

下面是 xarray 结构体的代码，单看这个数据结构会觉得很奇怪，好像什么也没有。然后，我们在看到 `struct xa_node` 的内容，就和图上的对应了。`xarray->xa_head = xa_node0`，这里采用了 RCU 机制，使得在读取的时候不需要加锁，写入的时候可以提供 RCU 提供的 API 来更新指针。由于 `xa_head` 和 `slots[]` 是在并发环境下访问的，RCU 的使用确保了高效的无锁读取。

```
struct xarray {
    spinlock_t  xa_lock;
/* private: The rest of the data structure is not to be used directly. */
    gfp_t       xa_flags;
    void __rcu *    xa_head;
};

struct xa_node {
    unsigned char   shift;      /* Bits remaining in each slot */
    unsigned char   offset;     /* Slot offset in parent */
    unsigned char   count;      /* Total entry count */
    unsigned char   nr_values;  /* Value entry count */
    struct xa_node __rcu *parent;   /* NULL at top of tree */
    struct xarray   *array;     /* The array we belong to */
    union {
        struct list_head private_list;  /* For tree user */
        struct rcu_head rcu_head;   /* Used when freeing node */
    };
    void __rcu  *slots[XA_CHUNK_SIZE];
    union {
        unsigned long   tags[XA_MAX_MARKS][XA_MARK_LONGS];
        unsigned long   marks[XA_MAX_MARKS][XA_MARK_LONGS];
    };
};
```

在介绍具体的操作之前，这里解释一下其中几个关键参数。

- slots[]。槽数组，每个结点有 64 个槽位，每个槽会存放一个页面或者一个结点。
- count。指的是当前的已有的槽的个数。
- offset。指的是当前的结点是在自己的父节点的第几个槽位中，如果没有父节点，则该值为 0。

- shift。通过该值，可以查找根据当前页面的索引查找到具体的槽位在哪个地方。内核通过计算`(index >> node->shift) & XA_CHUNK_MASK`进行获取。在图 4(b) 当中，结点 1 的 shift 为 6，结点 2 的 shift 为 0。现在要查找页面索引为 64 的页面，这个时候会先从头节点查找，然后计算`(64 >> 6) & 63 = 1`，说明 64 这个页面在槽位 1 的结点上面，然后继续往下查找，然后计算`64 & (XA_CHUNK_SIZE - 1) = 64 & (64 - 1) = 0`，最终找到在结点 2 的槽位为 0 的地方。
- parent。当前结点的父节点。

在 xarray 代码中，经常能看到一些奇怪的操作，比如`xa_is_internal`、`xas_invalid`，其中的内容直呼看不懂，在代码的开头，作者对 entry 的种类有一些描述，我们这里汇总一下。

```
(00)Pointer:
(x1)Value:
(10)Internal:
   [0, 255]       sibling entries       --+- advanced entry
   256            retry entry           -/
   257            zero entry
   > 4096         node entry(without left shift)
   >= -MAX_ERRNO  Error(never stored in slots array)
```

条目类型	条件	值范围（或标记）	含义
Value 条目	`(unsigned long)entry & 1`	`x1`	值条目或带标记的指针
Pointer 条目	—	`00`	普通指针
Sibling 条目	`xa_is_internal() && entry < XA_CHUNK_SIZE`	`[0, 62]`（当前代码用）	指向兄弟节点
Retry 条目	`entry == xa_mk_internal(256)`	`256`	提示需要重试的条目
Zero 条目	`entry == xa_mk_internal(257)`	`257`	表示零值条目
Node 条目	`xa_is_internal() && entry > 4096`	大于 `4096`	指向树的子节点
Error 条目	`xa_is_internal() && entry >= -MAX_ERRNO`	`[-MAX_ERRNO, -2]`	错误状态，不存储在 slots 中，仅用于返回错误信息

紧接着，会介绍一些与 xarray 相关的操作，在介绍之前，这里还要在介绍一个结构 xa_state，它是描述单次操作的上下文。

```
struct xa_state {
    struct xarray *xa;          // 指向当前操作的 xarray
    unsigned long xa_index;     // 操作的目标索引
    unsigned char xa_shift;     // 当前状态的位移量，描述当前层次的索引范围
    unsigned char xa_sibs;      // 兄弟结点
    unsigned char xa_offset;    // 当前 xa_node 的 slots 的偏移量
    unsigned char xa_pad;       /* Helps gcc generate better code */
    struct xa_node *xa_node;    // 当前访问的节点指针
    struct xa_node *xa_alloc;   // 用于延迟分配新节点的内存空间
    xa_update_node_t xa_update; // 更新节点的状态
};
```

其中有一个 xa_node 表示操作过程中存储的结点，被用作存储操作过程中的 **状态信息** 和 **返回值**。这种设计减少了额外的标志变量使用，提高了操作效率。

```
// 将错误代码（errno）编码成一个伪造的 xa_node 指针，以表示错误状态。
#define XA_ERROR(errno) ((struct xa_node *)(((unsigned long)errno << 2) | 2UL))

//  表示 xas 已经超出索引边界
#define XAS_BOUNDS ((struct xa_node *)1UL)

// 表示需要重新开始操作
#define XAS_RESTART ((struct xa_node *)3UL)

// 判断 xas->xa_node 是否表示无效状态
xas_invalid(): (unsigned long)xas->xa_node & 3

 // 判断结点处于冻结状态。
 xas_frozen(): node & 2

 // 表示当前操作是否位于树的顶部或需要重新开始
 xas_top(): node <= XAS_RESTART

 //  判断当前是否是实际结点指针，也就是最低两位是不是为 00
 xas_is_node(): !((unsigned long)xas->xa_node & 3) && xas->xa_node

 //  提取对应的错误码
```

```
xas_error(): xa_err(xas->xa_node)
```

xas_load 会根据本次记录的操作上下文，然后找到具体要加载的页面。

```
void *xas_load(struct xa_state *xas)
{
    // 获取 xarray 的头节点的地址
    void *entry = xas_start(xas);

    // 判断当前是否是一个内部结点，判断 entry 的低两位是否是 01
    while (xa_is_node(entry)) {
        // 将 entry 的低两位 01 去掉，然后就是对应的 xa_node 的地址
        struct xa_node *node = xa_to_node(entry);

        if (xas->xa_shift > node->shift)
            break;
        // 往下查找对应的结点
        entry = xas_descend(xas, node);

        // 已经到达最底层了
        if (node->shift == 0)
            break;
    }
    return entry;
}

static void *xas_descend(struct xa_state *xas, struct xa_node *node)
{
    // 获取当前页面对应在哪个槽位当中
    unsigned int offset = get_offset(xas->xa_index, node);
    void *entry = xa_entry(xas->xa, node, offset);

    // 更新当前 node 结点
    xas->xa_node = node;

    xas->xa_offset = offset;
    return entry;
}
```

相比较 load 操作，store 操作会复杂的多，因为这里还涉及到了结构的调整、标记的清理等等。在理解 xas_store 之前，还需要理解 xas_create 这个函数的使用。第二个参数 `allow_root` 的作用是是否允许将条目直接存储在根节点，如果该值为 `false`，则可能需要创建新的结点来存放数据。

整个代码的部分比较复杂，这里配合案例与代码进行解读，下面是代码的原样，没有做任何删减。

- 案例一。目前，我们当前的文件还没有任何的页缓存页面的时候。这时，假设我们要在索引 0 处插入一个新的数据条目 1，并且允许直接在根节点 xa_head 处存储条目。目前初始状态是 `xas->xa_shift = 0` 、`allow_root = true` 、`xas->xa_head = NULL` 且 `xas->xa_node = XAS_RESTART(3UL)`。因为当前操作位于顶层(根节点)，通过获取根节点的条目 `xa_head_locked(xa)`，假设当前的根节点为空，那么需要拓展 xArray 的层级。由于 entry 为空，且 xa->inode 为 0，因此返回 0，表示无需拓展层级。slot 指向根节点，由于 `shift == order`，因此返回 NULL，表示根节点处创建了一个槽，准备存储新的数据条目 1。
- 案例二。假设我们要在索引 64 处插入一个新的数据条目 2，并且不允许在根节点处存储条目。如图 4(a)所示，因为默认的 XA_CHUNK_SHIFT 为 6，表示一个结点的槽位只能存放 64 个条目，因此初始化是 `xas->xa_shift = 6` 且 `allow_root = false`。因为当前操作仍位于顶层，获取根节点的条目之后，开始尝试进行拓展 `xas_expand`，这里就不展示对应的代码了，具体是这样做的，目前 entry 是根节点的条目，它是一个内部结点，最大索引为 63，而当前插入的索引是 64，这时候会创建一个新的结点 node1，这个结点的 count 为 1，这个结点的 slots[0] 指向这 head 结点。如果原先 head 是一个内部结点，则将其 offset 设置为 0，并且修改其父结点指针指向 node1，然后将当前 node1 结点设置为头节点，`shift += XA_CHUNK_SHIFT`。之后 `xas->xa_node` 指向当前的 node1 结点，然后返回 12。现在的 shift 是 12，order 仍是 6，然后进入循环，拿到 node1 的结点，并往下开始查找。计算处在其第二个槽上，然后这时候 entry 是空的，因此最后返回空。
- 案例三。在已有的索引处 77 插入一个新的数据条目 3，并且目前是有一个子节点 node2。这个时候因为 node 存在，因此通过 `xa_entry_locked` 获取该节点对应的条目，然后返回。

```
static void *xas_create(struct xa_state *xas, bool allow_root)
{
    struct xarray *xa = xas->xa;
    void *entry;
    void __rcu **slot;
    struct xa_node *node = xas->xa_node;
    int shift;
```

```
unsigned int order = xas->xa_shift;

if (xas_top(node)) {
    entry = xa_head_locked(xa);
    xas->xa_node = NULL;
    if (!entry && xa_zero_busy(xa))
        entry = XA_ZERO_ENTRY;
    shift = xas_expand(xas, entry);
    if (shift < 0)
        return NULL;
    if (!shift && !allow_root)
        shift = XA_CHUNK_SHIFT;
    entry = xa_head_locked(xa);
    slot = &xa->xa_head;
} else if (xas_error(xas)) {
    return NULL;
} else if (node) {
    unsigned int offset = xas->xa_offset;

    shift = node->shift;
    entry = xa_entry_locked(xa, node, offset);
    slot = &node->slots[offset];
} else {
    shift = 0;
    entry = xa_head_locked(xa);
    slot = &xa->xa_head;
}

while (shift > order) {
    shift -= XA_CHUNK_SHIFT;
    if (!entry) {
        node = xas_alloc(xas, shift);
        if (!node)
            break;
        if (xa_track_free(xa))
            node_mark_all(node, XA_FREE_MARK);
        rcu_assign_pointer(*slot, xa_mk_node(node));
    } else if (xa_is_node(entry)) {
        node = xa_to_node(entry);
```

```
        } else {
            break;
        }
        entry = xas_descend(xas, node);
        slot = &node->slots[xas->xa_offset];
    }

    return entry;
}
```

总得来说，`xa_create` 的作用其实就是找到要修改的条目的 entry，然后返回。紧接着，我们回到 `xa_store`上面。同样，我们继续对这三个案例进行解读。

- 案例一。案例一是在索引 0 的上，插入一个新的数值条目 1，目前 xArray 没有任何结点。这时候 allow_root 为 true，经过 `xa_create` 的创建，返回 `first = NULL`。

```
void *xas_store(struct xa_state *xas, void *entry)
{
    struct xa_node *node;
    void __rcu **slot = &xas->xa->xa_head;
    unsigned int offset, max;
    int count = 0;
    int values = 0;
    void *first, *next;
    bool value = xa_is_value(entry);

    if (entry) {
        bool allow_root = !xa_is_node(entry) && !xa_is_zero(entry);
        first = xas_create(xas, allow_root);
    } else {
        first = xas_load(xas);
    }

    if (xas_invalid(xas))
        return first;
    node = xas->xa_node;
    if (node && (xas->xa_shift < node->shift))
        xas->xa_sibs = 0;
    if ((first == entry) && !xas->xa_sibs)
        return first;
```

```
next = first;
offset = xas->xa_offset;
max = xas->xa_offset + xas->xa_sibs;


if (node) {
    slot = &node->slots[offset];
    if (xas->xa_sibs)
        xas_squash_marks(xas);
}
if (!entry)
    xas_init_marks(xas);

for (;;) {
    rcu_assign_pointer(*slot, entry);
    if (xa_is_node(next) && (!node || node->shift))
        xas_free_nodes(xas, xa_to_node(next));
    if (!node)
        break;
    count += !next - !entry;
    values += !xa_is_value(first) - !value;
    if (entry) {
        if (offset == max)
            break;
        if (!xa_is_sibling(entry))
            entry = xa_mk_sibling(xas->xa_offset);
    } else {
        if (offset == XA_CHUNK_MASK)
            break;
    }
    next = xa_entry_locked(xas->xa, node, ++offset);
    if (!xa_is_sibling(next)) {
        if (!entry && (offset > max))
            break;
        first = next;
    }
    slot++;
}
```

```
    update_node(xas, node, count, values);
    return first;
}
```

1.2.2 查找页缓存

根据图 2 可以知道，通过快路径查找页缓存的过程，是先通过查找空闲链表是否有空闲的页面，如果有的话，就获取一个可用的页面，然后放入到页缓存当中。下面，通过代码来进行解释，`pagecache_get_page` 是找一个页缓存页面的函数，其中会先通过 `page = find_get_entry(mapping, offset)`，从文件的地址空间中的指定偏移量查找一个页面。首先，将当前查找 xArray 的过程初始化一个状态 xas，这个状态当中的 xArray 保存着当前文件的所属的页缓存页面，查找页面的索引会根据 offset 所决定，之后通过 `xas_load` 获取该页面。

```
struct page *find_get_entry(struct address_space *mapping, pgoff_t offset)
{
    XA_STATE(xas, &mapping->i_pages, offset);
    struct page *head, *page;

    rcu_read_lock();
repeat:
    xas_reset(&xas);
    page = xas_load(&xas);
    if (xas_retry(&xas, page))
        goto repeat;
    /*
     * A shadow entry of a recently evicted page, or a swap entry from
     * shmem/tmpfs.  Return it without attempting to raise page count.
     */
    if (!page || xa_is_value(page))
        goto out;

    head = compound_head(page);
    if (!page_cache_get_speculative(head))
        goto repeat;

    /* The page was split under us? */
    if (compound_head(page) != head) {
        put_page(head);
```

```
        goto repeat;
    }

    if (unlikely(page != xas_reload(&xas))) {
        put_page(head);
        goto repeat;
    }
out:
    rcu_read_unlock();

    return page;
}
```

1.3 分配页面

在 pagecache_get_page 当中，通过 find_get_entry 查找页面，如果查找不到，则返回 NULL，根据判断，会跳转到 no_page，通过 page = __page_cache_alloc(gfp_mask) 分配一个新的页面。在 __page_cache_alloc 当中，这里涉及到了对页面放在哪个 NUMA 结点进行了考虑。默认情况下，会选择一个合适的结点进行分配，如果用户将 cpuset.memory_spread_page 标志为 false，则按照用户进程的 NUMA 结点分配策略执行。

```
struct page *__page_cache_alloc(gfp_t gfp)
{
    int n;
    struct page *page;

    if (cpuset_do_page_mem_spread()) {
        unsigned int cpuset_mems_cookie;
        do {
            cpuset_mems_cookie = read_mems_allowed_begin();
            n = cpuset_mem_spread_node();
            page = __alloc_pages_node(n, gfp, 0);
        } while (!page && read_mems_allowed_retry(cpuset_mems_cookie));

        return page;
    }
    return alloc_pages(gfp, 0);
}
```

这里做一个补充知识，什么是 cpuset?

Cpusets 是 Linux 提供的一种机制，用于将一组 CPU 和 内存节点(NUMA 节点) 分配给一组进程。它的核心目标是约束进程的 CPU 和 内存分配范围，从而实现更高效的资源管理，尤其是在大规模、多核和 NUMA 系统中。在大规模、多核和 NUMA 系统中，**CPU 和内存访问性能高度依赖于资源的分配方式**。如果没有合理的资源隔离和调度，可能会导致：

- 内存访问延迟增加（尤其在 NUMA 系统中，远程节点访问代价高）。
- CPU 负载不平衡，进而影响性能。
- 不同进程之间资源争用，导致性能抖动。

可以通过将 CPU 和内存分配到不同到进程组，减少进程之间资源的争用。也可以通过它提供的资源绑定功能，确保进程使用其本地 NUMA 结点的内存，降低跨结点的开销。在运行时，可以根据工作负载的变化，灵活调整进程与资源的绑定关系。

其中有几个标志与 NUMA 结点相关：

- cpuset.mem_hardwall。如果启用，会确保分配的内存严格局限于当前 cpuset 的结点，防止系统跨界点分配内存。
- cpuset.mem_migrate 。在调整 cpuset 的内存结点范围时，可以选择是否将已经分配的页迁移到新的内存节点。
- cpuset.sched_load_balance。决定是否在指定的 CPU 集内启用负载均衡。
- cpuset.memory_spread_page。如果设置，则在允许的节点上均匀分配页面缓存。默认情况下，这个值设置为 0。

通过一系列的策略选择好指定的 NUMA 结点之后，会将新的页面分配到这个 NUMA 结点上面。`__alloc_pages_node` -> `__alloc_pages` -> `__alloc_pages_nodemask`，此时的 order 是 0，preferred_nid 是要分配的 NUMA 结点，nodemask 为 NULL。

```
struct page *__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
                        int preferred_nid, nodemask_t *nodemask)
{
    struct page *page;
    unsigned int alloc_flags = ALLOC_WMARK_LOW;
    gfp_t alloc_mask; /* The gfp_t that was actually used for allocation */
    // 分配上下文
    struct alloc_context ac = {};

    if (unlikely(order >= MAX_ORDER)) {
        WARN_ON_ONCE(!(gfp_mask & __GFP_NOWARN));
        return NULL;
```

```
    }

    gfp_mask &= gfp_allowed_mask;
    alloc_mask = gfp_mask;
    // 做好分配前的准备工作，会从指定的 NUMA 结点当中获取 zonelist
    // zonelist 会在每个结点当中存在两个，如果当前 GFP 设置了 GFP_THISNODE，则会返回只包含当前 NUMA 结点的 zonelist，否则返回整个内存的 zonelist。
    if (!prepare_alloc_pages(gfp_mask, order, preferred_nid, nodemask, &ac,
                 &alloc_mask, &alloc_flags))
        return NULL;

    // 获取 zonelist 的第一个 zone 区域，放到 ac->preferred_zoneref 当中
    finalise_ac(gfp_mask, &ac);

    // 设置 ALLOC_NOFRAGMENT 来减少内存碎片
    alloc_flags |=
        alloc_flags_nofragment(ac.preferred_zoneref->zone, gfp_mask);

    // 快路径查找
    page = get_page_from_freelist(alloc_mask, order, alloc_flags, &ac);
    if (likely(page))
        goto out;

    alloc_mask = current_gfp_context(gfp_mask);
    ac.spread_dirty_pages = false;

    if (unlikely(ac.nodemask != nodemask))
        ac.nodemask = nodemask;

    // 慢路径查找
    page = __alloc_pages_slowpath(alloc_mask, order, &ac);

out:
    if (memcg_kmem_enabled() && (gfp_mask & __GFP_ACCOUNT) && page &&
        unlikely(__memcg_kmem_charge(page, gfp_mask, order) != 0)) {
        __free_pages(page, order);
        page = NULL;
    }
```

```
    trace_mm_page_alloc(page, order, alloc_mask, ac.migratetype);

    return page;
}
```

1.3.1 快路径分配页面

内核提供 `get_page_from_freelist` 进行快路径的查找，查找过程如下：

- 会先遍历一遍从 prepare_alloc_pages 拿到的 zonelist，然后寻找一个可用的 zone 区域。如果系统启动了 cpuset 功能。会限制内存分配到指定的 NUMA 节点。`__cpuset_zone_allowed()` 检查当前区域是否满足这些约束。
- 如果启动了 `spread_dirty_pages`，如果上一次检查的结点已经超过了脏页限制，则跳过该区域。如果当前节点脏页限制不足，则标记为已检查，并跳过该区域。确保分配写缓存页面时不会过度集中在某个节点，以避免单一节点的脏页过多而影响系统性能。
- 如果 ALLOC_NOFRAGMENT 标志被设置，并且有多个在线节点，但当前区域不是首选区域。判断当前区域的结点和首选区域的结点是不是同样的，如果不是的话，说明当前区域的结点是跨界点，这时候会尝试重新寻找一个新的区域。
- 进行水印检查，判断当前区域是否有足够的空闲内存。如果没有，采取额外的操作。
- 如果已经找到了指定区域且存在可以分配的页面的话，会通过 `rmqueue` 从空闲链表中取出页面。
- 如果没有找到的话，期间会判断 node_reclaim_mode 有没有被设置，默认情况是 0，表示在当前区域找不到的时候不会进行回收，否则会进行回收策略。

```
static struct page *get_page_from_freelist(gfp_t gfp_mask, unsigned int order,
                        int alloc_flags,
                        const struct alloc_context *ac)
{
    struct zoneref *z;
    struct zone *zone;
    struct pglist_data *last_pgdat_dirty_limit = NULL;
    bool no_fallback;

retry:
    /*
     * Scan zonelist, looking for a zone with enough free.
     * See also __cpuset_node_allowed() comment in kernel/cpuset.c.
     */
```

```
no_fallback = alloc_flags & ALLOC_NOFRAGMENT;
z = ac->preferred_zoneref;

for_next_zone_zonelist_nodemask(zone, z, ac->zonelist, ac->high_zoneidx,
                ac->nodemask)
{
    struct page *page;
    unsigned long mark;

    if (cpusets_enabled() && (alloc_flags & ALLOC_CPUSET) &&
        !__cpuset_zone_allowed(zone, gfp_mask))
        continue;
    /*
     * When allocating a page cache page for writing, we
     * want to get it from a node that is within its dirty
     * limit, such that no single node holds more than its
     * proportional share of globally allowed dirty pages.
     * The dirty limits take into account the node's
     * lowmem reserves and high watermark so that kswapd
     * should be able to balance it without having to
     * write pages from its LRU list.
     *
     * XXX: For now, allow allocations to potentially
     * exceed the per-node dirty limit in the slowpath
     * (spread_dirty_pages unset) before going into reclaim,
     * which is important when on a NUMA setup the allowed
     * nodes are together not big enough to reach the
     * global limit.  The proper fix for these situations
     * will require awareness of nodes in the
     * dirty-throttling and the flusher threads.
     */
    if (ac->spread_dirty_pages) {
        if (last_pgdat_dirty_limit == zone->zone_pgdat)
            continue;

        if (!node_dirty_ok(zone->zone_pgdat)) {
            last_pgdat_dirty_limit = zone->zone_pgdat;
            continue;
        }
```

```
        }

        if (no_fallback && nr_online_nodes > 1 &&
            zone != ac->preferred_zoneref->zone) {
            int local_nid;

            /*
             * If moving to a remote node, retry but allow
             * fragmenting fallbacks. Locality is more important
             * than fragmentation avoidance.
             */
            local_nid = zone_to_nid(ac->preferred_zoneref->zone);
            if (zone_to_nid(zone) != local_nid) {
                alloc_flags &= ~ALLOC_NOFRAGMENT;
                goto retry;
            }
        }

        mark = wmark_pages(zone, alloc_flags & ALLOC_WMARK_MASK);
        if (!zone_watermark_fast(zone, order, mark,
                       ac_classzone_idx(ac), alloc_flags)) {
            int ret;

#ifdef CONFIG_DEFERRED_STRUCT_PAGE_INIT
            /*
             * Watermark failed for this zone, but see if we can
             * grow this zone if it contains deferred pages.
             */
            if (static_branch_unlikely(&deferred_pages)) {
                if (_deferred_grow_zone(zone, order))
                    goto try_this_zone;
            }
#endif
            /* Checked here to keep the fast path fast */
            BUILD_BUG_ON(ALLOC_NO_WATERMARKS < NR_WMARK);
            if (alloc_flags & ALLOC_NO_WATERMARKS)
                goto try_this_zone;

            if (node_reclaim_mode == 0 ||
```

```
                !zone_allows_reclaim(ac->preferred_zoneref->zone,
                        zone))
                continue;

            ret = node_reclaim(zone->zone_pgdat, gfp_mask, order);
            switch (ret) {
            case NODE_RECLAIM_NOSCAN:
                /* did not scan */
                continue;
            case NODE_RECLAIM_FULL:
                /* scanned but unreclaimable */
                continue;
            default:
                /* did we reclaim enough */
                if (zone_watermark_ok(zone, order, mark,
                            ac_classzone_idx(ac),
                            alloc_flags))
                    goto try_this_zone;

                continue;
            }
        }

    try_this_zone:
        page = rmqueue(ac->preferred_zoneref->zone, zone, order,
                gfp_mask, alloc_flags, ac->migratetype);
        if (page) {
            prep_new_page(page, order, gfp_mask, alloc_flags);

            /*
             * If this is a high-order atomic allocation then check
             * if the pageblock should be reserved for the future
             */
            if (unlikely(order && (alloc_flags & ALLOC_HARDER)))
                reserve_highatomic_pageblock(page, zone, order);

            return page;
        } else {
#ifdef CONFIG_DEFERRED_STRUCT_PAGE_INIT
```

```
            /* Try again if zone has deferred pages */
            if (static_branch_unlikely(&deferred_pages)) {
                if (_deferred_grow_zone(zone, order))
                    goto try_this_zone;
            }
#endif
        }
    }

    /*
     * It's possible on a UMA machine to get through all zones that are
     * fragmented. If avoiding fragmentation, reset and try again.
     */
    if (no_fallback) {
        alloc_flags &= ~ALLOC_NOFRAGMENT;
        goto retry;
    }

    return NULL;
}
```

__cpuset_zone_allowed 是判断指定的 zone 区域是否能分配内存，其实也是判断当前进程能否从这个 zone 区域对应的 NUMA 结点分配内存。

```
bool __cpuset_node_allowed(int node, gfp_t gfp_mask)
{
    struct cpuset *cs;      /* current cpuset ancestors */
    int allowed;            /* is allocation in zone z allowed? */
    unsigned long flags;

    if (in_interrupt())
        return true;
    // 如果当前结点是在进程的 mems_allowed 当中，表示可以分配。
    if (node_isset(node, current->mems_allowed))
        return true;

    if (unlikely(tsk_is_oom_victim(current)))
        return true;
    if (gfp_mask & __GFP_HARDWALL)  /* If hardwall request, stop here */
        return false;
```

```
    if (current->flags & PF_EXITING) /* Let dying task have memory */
        return true;

    /* Not hardwall and node outside mems_allowed: scan up cpusets */
    spin_lock_irqsave(&callback_lock, flags);

    rcu_read_lock();
    // 寻找一个最近的硬墙祖先的 cpuset ，判断其 mems_allowed 是否设置了允许当前结点分配
    cs = nearest_hardwall_ancestor(task_cs(current));
    allowed = node_isset(node, cs->mems_allowed);
    rcu_read_unlock();

    spin_unlock_irqrestore(&callback_lock, flags);
    return allowed;
}
```

1.3.2 慢路径分配页面

`page = __alloc_pages_slowpath(alloc_mask, order, &ac)` 从慢路径分配页面，其中会涉及到很多与页面回收相关的内容，可以看 第二部分 页面回收 进行了解，这里就不做详细的介绍。慢路径的过程当中，会先尝试各种策略进行页面回收，下面是慢路径分配页面的过程：

- 首先，如果分配标志中包含 ALLOC_KSWAPD，则会唤醒所有的回收线程 kswapd 进行页面的回收。唤醒之后，会采用保守的方式重新从快路径去获取页面。
- 紧接着回去判断是否采取 `direct_compact`，这个操作会影响系统性能。首先，它会扫描内存区域，识别空洞和可移动的页面，然后将可移动的页面迁移到新的位置，创建连续的空闲空间。通过页面迁移来整理内存碎片，这里可以满足一些大页面的请求，提高内存利用效率和减少内存碎片。

```
static inline struct page *__alloc_pages_slowpath(gfp_t gfp_mask,
                        unsigned int order,
                        struct alloc_context *ac)
{
    bool can_direct_reclaim = gfp_mask & __GFP_DIRECT_RECLAIM;
    const bool costly_order = order > PAGE_ALLOC_COSTLY_ORDER;
    struct page *page = NULL;
    unsigned int alloc_flags;
    unsigned long did_some_progress;
```

```
    enum compact_priority compact_priority;
    enum compact_result compact_result;
    int compaction_retries;
    int no_progress_loops;
    unsigned int cpuset_mems_cookie;
    int reserve_flags;

    /*
     * We also sanity check to catch abuse of atomic reserves being used by
     * callers that are not in atomic context.
     */
    if (WARN_ON_ONCE((gfp_mask & (__GFP_ATOMIC | __GFP_DIRECT_RECLAIM)) ==
                (__GFP_ATOMIC | __GFP_DIRECT_RECLAIM)))
        gfp_mask &= ~__GFP_ATOMIC;

retry_cpuset:
    compaction_retries = 0;
    no_progress_loops = 0;
    compact_priority = DEF_COMPACT_PRIORITY;
    cpuset_mems_cookie = read_mems_allowed_begin();

    /*
     * The fast path uses conservative alloc_flags to succeed only until
     * kswapd needs to be woken up, and to avoid the cost of setting up
     * alloc_flags precisely. So we do that now.
     */
    alloc_flags = gfp_to_alloc_flags(gfp_mask);

    /*
     * We need to recalculate the starting point for the zonelist iterator
     * because we might have used different nodemask in the fast path, or
     * there was a cpuset modification and we are retrying - otherwise we
     * could end up iterating over non-eligible zones endlessly.
     */
    ac->preferred_zoneref = first_zones_zonelist(
        ac->zonelist, ac->high_zoneidx, ac->nodemask);
    if (!ac->preferred_zoneref->zone)
        goto nopage;
```

```
if (alloc_flags & ALLOC_KSWAPD)
    wake_all_kswapds(order, gfp_mask, ac);

/*
 * The adjusted alloc_flags might result in immediate success, so try
 * that first
 */
page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
if (page)
    goto got_pg;

/*
 * For costly allocations, try direct compaction first, as it's likely
 * that we have enough base pages and don't need to reclaim. For non-
 * movable high-order allocations, do that as well, as compaction will
 * try prevent permanent fragmentation by migrating from blocks of the
 * same migratetype.
 * Don't try this for allocations that are allowed to ignore
 * watermarks, as the ALLOC_NO_WATERMARKS attempt didn't yet happen.
 */
if (can_direct_reclaim &&
    (costly_order ||
     (order > 0 && ac->migratetype != MIGRATE_MOVABLE)) &&
    !gfp_pfmemalloc_allowed(gfp_mask)) {
    page = __alloc_pages_direct_compact(gfp_mask, order,
                        alloc_flags, ac,
                        INIT_COMPACT_PRIORITY,
                        &compact_result);
    if (page)
        goto got_pg;

    /*
     * Checks for costly allocations with __GFP_NORETRY, which
     * includes THP page fault allocations
     */
    if (costly_order && (gfp_mask & __GFP_NORETRY)) {
        /*
         * If compaction is deferred for high-order allocations,
         * it is because sync compaction recently failed. If
```

```
             * this is the case and the caller requested a THP
             * allocation, we do not want to heavily disrupt the
             * system, so we fail the allocation instead of entering
             * direct reclaim.
             */
            if (compact_result == COMPACT_DEFERRED)
                goto nopage;

            /*
             * Looks like reclaim/compaction is worth trying, but
             * sync compaction could be very expensive, so keep
             * using async compaction.
             */
            compact_priority = INIT_COMPACT_PRIORITY;
        }
    }

retry:
    /* Ensure kswapd doesn't accidentally go to sleep as long as we loop */
    if (alloc_flags & ALLOC_KSWAPD)
        wake_all_kswapds(order, gfp_mask, ac);

    reserve_flags = __gfp_pfmemalloc_flags(gfp_mask);
    if (reserve_flags)
        alloc_flags = reserve_flags;

    /*
     * Reset the nodemask and zonelist iterators if memory policies can be
     * ignored. These allocations are high priority and system rather than
     * user oriented.
     */
    if (!(alloc_flags & ALLOC_CPUSET) || reserve_flags) {
        ac->nodemask = NULL;
        ac->preferred_zoneref = first_zones_zonelist(
            ac->zonelist, ac->high_zoneidx, ac->nodemask);
    }

    /* Attempt with potentially adjusted zonelist and alloc_flags */
    page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
```

```
if (page)
    goto got_pg;

/* Caller is not willing to reclaim, we can't balance anything */
if (!can_direct_reclaim)
    goto nopage;

/* Avoid recursion of direct reclaim */
if (current->flags & PF_MEMALLOC)
    goto nopage;

/* Try direct reclaim and then allocating */
page = __alloc_pages_direct_reclaim(gfp_mask, order, alloc_flags, ac,
                    &did_some_progress);
if (page)
    goto got_pg;

/* Try direct compaction and then allocating */
page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flags, ac,
                    compact_priority, &compact_result);
if (page)
    goto got_pg;

/* Do not loop if specifically requested */
if (gfp_mask & __GFP_NORETRY)
    goto nopage;

/*
 * Do not retry costly high order allocations unless they are
 * __GFP_RETRY_MAYFAIL
 */
if (costly_order && !(gfp_mask & __GFP_RETRY_MAYFAIL))
    goto nopage;

if (should_reclaim_retry(gfp_mask, order, ac, alloc_flags,
             did_some_progress > 0, &no_progress_loops))
    goto retry;

/*
```

```
	 * It doesn't make any sense to retry for the compaction if the order-0
	 * reclaim is not able to make any progress because the current
	 * implementation of the compaction depends on the sufficient amount
	 * of free memory (see __compaction_suitable)
	 */
	if (did_some_progress > 0 &&
			should_compact_retry(ac, order, alloc_flags, compact_result,
					&compact_priority, &compaction_retries))
			goto retry;

	/* Deal with possible cpuset update races before we start OOM killing */
	if (check_retry_cpuset(cpuset_mems_cookie, ac))
			goto retry_cpuset;

	/* Reclaim has failed us, start killing things */
	page = __alloc_pages_may_oom(gfp_mask, order, ac, &did_some_progress);
	if (page)
			goto got_pg;

	/* Avoid allocations with no watermarks from looping endlessly */
	if (tsk_is_oom_victim(current) &&
			(alloc_flags == ALLOC_OOM || (gfp_mask & __GFP_NOMEMALLOC)))
			goto nopage;

	/* Retry as long as the OOM killer is making progress */
	if (did_some_progress) {
			no_progress_loops = 0;
			goto retry;
	}

nopage:
	/* Deal with possible cpuset update races before we fail */
	if (check_retry_cpuset(cpuset_mems_cookie, ac))
			goto retry_cpuset;

	/*
	 * Make sure that __GFP_NOFAIL request doesn't leak out and make sure
	 * we always retry
	 */
```

```
    if (gfp_mask & __GFP_NOFAIL) {
        /*
         * All existing users of the __GFP_NOFAIL are blockable, so warn
         * of any new users that actually require GFP_NOWAIT
         */
        if (WARN_ON_ONCE(!can_direct_reclaim))
            goto fail;

        /*
         * PF_MEMALLOC request from this context is rather bizarre
         * because we cannot reclaim anything and only can loop waiting
         * for somebody to do a work for us
         */
        WARN_ON_ONCE(current->flags & PF_MEMALLOC);

        /*
         * non failing costly orders are a hard requirement which we
         * are not prepared for much so let's warn about these users
         * so that we can identify them and convert them to something
         * else.
         */
        WARN_ON_ONCE(order > PAGE_ALLOC_COSTLY_ORDER);

        /*
         * Help non-failing allocations by giving them access to memory
         * reserves but do not use ALLOC_NO_WATERMARKS because this
         * could deplete whole memory reserves which would just make
         * the situation worse
         */
        page = __alloc_pages_cpuset_fallback(gfp_mask, order,
                                ALLOC_HARDER, ac);
        if (page)
            goto got_pg;

        cond_resched();
        goto retry;
    }
fail:
    warn_alloc(gfp_mask, ac->nodemask, "page allocation failure: order:%u",
```

```
            order);
got_pg:
    return page;
}
```

2. 页面回收

当内存从伙伴系统当中申请内存失败的时候，首先会通过快路径 `get_page_free_freelist` 去申请内存，当快路径申请内存失败的时候，会进入慢路径 `__alloc_pages_slowpath` 中尝试进行内存回收和规整的手段，其中会整理出空闲的内存。

2.1 kswapd 回收线程

kswapd 线程属于内核线程，在系统启动的时候，内核会创建 `kswapd<node-id>`线程，为了防止对系统的影响，每个 node 都会有一个回收线程，在内核启动完毕之后并且对内存进行初始化之后，会创建回收线程，处于睡眠状态。

`__alloc_pages_slowpath` 当中第一个做的回收策略就是唤醒所有的回收线程，`wake_all_kswapds` 启动回收线程，`balance_pgdat` 对指定节点的内存从 LRU 和 slab 中进行回收，内存足够但是碎片化严重的情况下，会唤醒内存规整。

下面对 kswapd 进行详细的解读。

2.1.1 kswapd 数据结构

每个节点当有一个 kswapd 回收线程，每个 node 节点都有一个 `struct gplist_data` 结构，kswapd 相关的内容会放到这个结构当中，下面单独精简出来。

```
typedef struct pglist_data {
     ... ...
    int node_id;                    // 该结点对应的id
    wait_queue_head_t kswapd_wait; // kswpad线程等待队列
    wait_queue_head_t pfmemalloc_wait; // 用于用户直接回收内存等待队列，该队列会同步等待 kswapd 完成内存回收后，在进行内存申请
    struct task_struct *kswapd; // 该节点 kswapd 线程 task
    int kswapd_order;           // kswapd内存回收order，内存回收大小
    enum zone_type kswapd_highest_zoneidx; // 扫描内存最高的 zone index
    int kswapd_failures;  // 内存回收失败次数
     ... ...
} pg_data_t;
```

当 kswapd 回收线程模块被加载的时候，会通过初始化创建 kswapd 线程。

```
static int __init kswapd_init(void)
{
    int nid;

    swap_setup();
    for_each_node_state(nid, N_MEMORY)
      kswapd_run(nid);
    return 0;
}

int kswapd_run(int nid)
{
    pg_data_t *pgdat = NODE_DATA(nid);
    int ret = 0;

    if (pgdat->kswapd)
      return 0;

    pgdat->kswapd = kthread_run(kswapd, pgdat, "kswapd%d", nid);
    if (IS_ERR(pgdat->kswapd)) {
      /* failure at boot is fatal */
      BUG_ON(system_state < SYSTEM_RUNNING);
      pr_err("Failed to start kswapd on node %d\n", nid);
      ret = PTR_ERR(pgdat->kswapd);
      pgdat->kswapd = NULL;
```

```
    }
    return ret;
}
```

3. inode 结点

这一部分相当于是一个补充知识，inode 是文件系统当中用于存储文件或者目录的元数据的数据结构，会包含文件的权限、大小、所有者、时间戳。这里重点关注一个字段，就是 i_mapping，管理文件的页缓存和内存映射的。

```
struct inode {
    umode_t i_mode;                 // 文件的类型和权限，表示文件的模式（如普通文件、目录等）
    unsigned short i_opflags;       // inode 操作的标志（用于标记 inode 的状态）
    kuid_t i_uid;                   // 文件所有者的用户 ID
    kgid_t i_gid;                   // 文件所有者的组 ID
    unsigned int i_flags;           // inode 标志（例如只读、不可删除等）

#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl *i_acl;        // 文件的 POSIX ACL（访问控制列表）
    struct posix_acl *i_default_acl; // 文件的默认 POSIX ACL
#endif

    const struct inode_operations *i_op;  // inode 操作函数指针（包括打开、读取、写入等操作）
    struct super_block *i_sb;       // 指向超级块的指针，表示文件系统的基本信息
    struct address_space *i_mapping; // 页缓存映射，指向文件的映射区域（用于缓存文件数据）

#ifdef CONFIG_SECURITY
    void *i_security;               // 安全相关的数据（如 SELinux、AppArmor 等）
#endif

    unsigned long i_ino;            // inode 号，唯一标识一个文件
    union {
        const unsigned int i_nlink;    // 链接计数，表示文件的硬链接数量
        unsigned int __i_nlink;         // 链接计数（可以修改）
    };
    dev_t i_rdev;                   // 设备号，仅用于设备文件
```

```
    loff_t i_size;                  // 文件的大小（以字节为单位）
    struct timespec64 i_atime;      // 最后访问时间
    struct timespec64 i_mtime;      // 最后修改时间
    struct timespec64 i_ctime;      // inode 最近更改时间
    spinlock_t i_lock;              // 用于保护 i_blocks、i_bytes 和 i_size 等字段的自
旋锁
    unsigned short i_bytes;         // 文件的字节数（一般用来标识实际占用的空间）
    u8 i_blkbits;                   // 块的位数，文件系统的块大小（通常是 4096）
    u8 i_write_hint;                // 文件的写入提示（例如是否为日志文件）
    blkcnt_t i_blocks;              // 文件占用的块数

#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t i_size_seqcount;     // 序列计数器，用于同步文件大小的访问
#endif

    unsigned long i_state;          // inode 状态标志（例如 inode 是否被删除）
    struct rw_semaphore i_rwsem;    // 用于同步的读写信号量

    unsigned long dirtied_when;     // 文件首次变脏的时间（以 jiffies 为单位）
    unsigned long dirtied_time_when; // 文件上次变脏的时间

    struct hlist_node i_hash;       // 哈希链表节点，用于 inode 哈希表管理
    struct list_head i_io_list;     // 后备设备的 IO 列表，用于 IO 操作
#ifdef CONFIG_CGROUP_WRITEBACK
    struct bdi_writeback *i_wb;     // 关联的 cgroup 写回数据
#endif
    struct list_head i_lru;         // inode 的 LRU（最近最少使用）链表
    struct list_head i_sb_list;     // inode 在超级块中的链表
    struct list_head i_wb_list;     // 写回操作链表
    union {
        struct hlist_head i_dentry; // 目录项（dentry）链表头
        struct rcu_head i_rcu;      // RCU 回收头
    };
    atomic64_t i_version;           // inode 的版本号，用于并发修改控制
    atomic_t i_count;               // inode 引用计数，表示当前有多少进程正在引用该
inode
    atomic_t i_dio_count;           // 文件直接 IO 的引用计数
    atomic_t i_writecount;          // 文件写入操作的计数
```

```
    const struct file_operations *i_fop; // 文件操作函数指针（包括打开、读取、写入等文件操作）
    struct file_lock_context *i_flctx;  // 文件锁上下文（用于文件锁管理）

    struct address_space i_data;    // inode 数据的地址空间，表示与文件内容相关的内存映射

    struct list_head i_devices;     // 设备文件的设备列表
    union {
        struct pipe_inode_info *i_pipe;  // 管道相关的信息
        struct block_device *i_bdev;     // 块设备相关的信息
        struct cdev *i_cdev;             // 字符设备相关的信息
        char *i_link;                    // 符号链接指向的路径
        unsigned i_dir_seq;              // 目录序列号（用于目录遍历）
    };

    __u32 i_generation;             // inode 的生成号（用于文件系统的多版本支持）

#ifdef CONFIG_FSNOTIFY
    __u32 i_fsnotify_mask;          // 文件系统通知的事件掩码，表示 inode 关心的事件
    struct fsnotify_mark_connector __rcu *i_fsnotify_marks; // 文件系统通知的标记连接器
#endif

#ifdef CONFIG_FS_ENCRYPTION
    struct fscrypt_info *i_crypt_info;  // 文件加密信息
#endif

    void *i_private;                // 文件系统或设备的私有数据指针
} __randomize_layout;
```

文件系统中，每个文件都会有一个唯一的 inode，假设，我们现在要打开一个文件 myfile.txt，这时候我们通过系统调用 open，SYSCALL_DEFINE3 是一个创建系统调用的宏定义，所以下面是系统调用 open 的实现。

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

在 do_sys_open 当中，实际上 do_filp_open 是具体的打开文件的函数。

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
    struct open_flags op;
    int fd = build_open_flags(flags, mode, &op);
    struct filename *tmp;

    if (fd)
        return fd;

    tmp = getname(filename);
    if (IS_ERR(tmp))
        return PTR_ERR(tmp);
    // 获取一个未使用的文件描述符
    fd = get_unused_fd_flags(flags);
    if (fd >= 0) {
        // 打开文件，这里的 file 是每个打开的文件都会有一个
        struct file *f = do_filp_open(dfd, tmp, &op);
        if (IS_ERR(f)) {
            put_unused_fd(fd);
            fd = PTR_ERR(f);
        } else {
            // 发送文件打开的事件通知
            fsnotify_open(f);
            // 文件指针安装到文件描述符表中
            fd_install(fd, f);
        }
    }
    putname(tmp);
    return fd;
```

```
}
```

dfd 是 目录文件描述符。pathname 是内核中的目录名，op 为打开文件时使用的标志信息，这里会设置好元数据，用于路径解析，我们重点看路径解析的过程。

```
struct file *do_filp_open(int dfd, struct filename *pathname, const struct open_flags
*op)
{
    struct nameidata nd;
    int flags = op->lookup_flags;
    struct file *filp;

    set_nameidata(&nd, dfd, pathname);
    filp = path_openat(&nd, op, flags | LOOKUP_RCPU);
    if (unlikely(filp == ERR_PTR(-ECHILD)))
        filp = path_openat(&nd, op, flags);
    if (unlikely(filp == ERR_PTR(-ESTALE)))
        filp = path_openat(&nd, op, flags | LOOKUP_REVAL);
    restore_nameidata();
    return filp;
}
```

LOOKUP_RCPU 是表示使用 RCU 的方式查找路径，可以提示查找路径的效率。我们这里只关注与页缓存相关的部分，主要是与文件 inode 结点相关的，因此，我们只考虑初始化的过程。

```
static struct file *path_openat(struct nameidata *nd, const struct open_flags *op,
unsigned flags)
{
    struct file *file;
    int error;

    // 分配一个空的 file 结构
    file = alloc_empty_file(op->open_flag, current_cred());
    if (IS_ERR(file))
        return file;

    if (unlikely(file->f_flags & __O_TMPFILE)) {
        error = do_tmpfile(nd, flags, op, file);
    } else if (unlikely(file->f_flags & O_PATH)) {
        error = do_o_path(nd, flags, file);
```

```
    } else {
        const char *s = path_init(nd, flags);
        while (!(error = link_path_walk(s, nd)) && (error = do_last(nd, file, op)) >
0) {
            nd->flags &= ~(LOOKUP_OPEN|LOOKUP_CREATE|LOOKUP_EXCL);
            s = trailing_symlink(nd);
        }
        terminate_walk(nd);
    }

    if (likely(!error)) {
        if (likely(file->f_mode & FMODE_OPENED))
            return file;
        WARN_ON(1);
        error = -EINVAL;
    }

    fput(file);

    if (error == -EOPENSTALE) {
        if (flags & LOOKUP_RCPU)
            error = -ECHILD;
        else
            error = -ESTALE;
    }

    return ERR_PTR(error);
}
```

这里重点关注 `link_path_walk` 的所在的循环部分，这个函数是 Linux 内核中路径查找到函数，主要是将一个路径名称解析成最终的目录条目。在解析过程中，文件系统会检查每个路径组件是否存在。如果遇到符号链接或路径中间的目录，操作系统会跟踪这些链接并继续查找。在 `do_last` 函数中，会调用 `lookup_open` 函数执行 lookup and maybe create 操作。

```
error = lookup_open(nd, &path, file, op, got_write);
```

在 `lookup_open` 的过程中，这里会先从缓存当中查找 `dentry` ，如果没有找到，则调用文件系统的 lookup 方法。最后，还是没有找到的情况下，说明该文件不存在，当我们设置了 `O_CREAT`，会调用文件系统的 creat 函数去进行创建。我们这里使用的是 ext4 文件系统，因此

```
static int lookup_open(struct nameidata *nd, struct path *path,
            struct file *file,
            const struct open_flags *op,
            bool got_write)
{
    struct dentry *dir = nd->path.dentry;
    struct inode *dir_inode = dir->d_inode;
    int open_flag = op->open_flag;
    struct dentry *dentry;
    int error, create_error = 0;
    umode_t mode = op->mode;
    DECLARE_WAIT_QUEUE_HEAD_ONSTACK(wq);

    if (unlikely(IS_DEADDIR(dir_inode)))
        return -ENOENT;

    file->f_mode &= ~FMODE_CREATED;
    dentry = d_lookup(dir, &nd->last);
    for (;;) {
        if (!dentry) {
            dentry = d_alloc_parallel(dir, &nd->last, &wq);
            if (IS_ERR(dentry))
                return PTR_ERR(dentry);
        }
        if (d_in_lookup(dentry))
            break;

        error = d_revalidate(dentry, nd->flags);
        if (likely(error > 0))
            break;
        if (error)
            goto out_dput;
        d_invalidate(dentry);
        dput(dentry);
        dentry = NULL;
    }
    if (dentry->d_inode) {
        /* Cached positive dentry: will open in f_op->open */
        goto out_no_open;
```

```
}

/*
 * Checking write permission is tricky, bacuse we don't know if we are
 * going to actually need it: O_CREAT opens should work as long as the
 * file exists.  But checking existence breaks atomicity.  The trick is
 * to check access and if not granted clear O_CREAT from the flags.
 *
 * Another problem is returing the "right" error value (e.g. for an
 * O_EXCL open we want to return EEXIST not EROFS).
 */
if (open_flag & O_CREAT) {
    if (!IS_POSIXACL(dir->d_inode))
        mode &= ~current_umask();
    if (unlikely(!got_write)) {
        create_error = -EROFS;
        open_flag &= ~O_CREAT;
        if (open_flag & (O_EXCL | O_TRUNC))
            goto no_open;
        /* No side effects, safe to clear O_CREAT */
    } else {
        create_error = may_o_create(&nd->path, dentry, mode);
        if (create_error) {
            open_flag &= ~O_CREAT;
            if (open_flag & O_EXCL)
                goto no_open;
        }
    }
} else if ((open_flag & (O_TRUNC|O_WRONLY|O_RDWR)) &&
       unlikely(!got_write)) {
    /*
     * No O_CREATE -> atomicity not a requirement -> fall
     * back to lookup + open
     */
    goto no_open;
}

if (dir_inode->i_op->atomic_open) {
    error = atomic_open(nd, dentry, path, file, op, open_flag,
```

```
                    mode);
        if (unlikely(error == -ENOENT) && create_error)
            error = create_error;
        return error;
    }

no_open:
    if (d_in_lookup(dentry)) {
        struct dentry *res = dir_inode->i_op->lookup(dir_inode, dentry,
                                nd->flags);
        d_lookup_done(dentry);
        if (unlikely(res)) {
            if (IS_ERR(res)) {
                error = PTR_ERR(res);
                goto out_dput;
            }
            dput(dentry);
            dentry = res;
        }
    }

    /* Negative dentry, just create the file */
    if (!dentry->d_inode && (open_flag & O_CREAT)) {
        file->f_mode |= FMODE_CREATED;
        audit_inode_child(dir_inode, dentry, AUDIT_TYPE_CHILD_CREATE);
        if (!dir_inode->i_op->create) {
            error = -EACCES;
            goto out_dput;
        }
        error = dir_inode->i_op->create(dir_inode, dentry, mode,
                        open_flag & O_EXCL);
        if (error)
            goto out_dput;
        fsnotify_create(dir_inode, dentry);
    }
    if (unlikely(create_error) && !dentry->d_inode) {
        error = create_error;
        goto out_dput;
    }
```

```
out_no_open:
    path->dentry = dentry;
    path->mnt = nd->path.mnt;
    return 0;

out_dput:
    dput(dentry);
    return error;
}
```

在ext4_create 内部，我们重点关注 inode 结点的创建。ext4_new_inode_start_handle 函数的内部调用了 __ext4_new_inode，而在其中，最为重要的就是通过超级块创建相应的 inode 结点，其中值得关注的是 new_inode(sb) 这个函数。

```
/*
 * By the time this is called, we already have created
 * the directory cache entry for the new file, but it
 * is so far negative - it has no inode.
 *
 * If the create succeeds, we fill in the inode information
 * with d_instantiate().
 */
static int ext4_create(struct inode *dir, struct dentry *dentry, umode_t mode,
               bool excl)
{
    handle_t *handle;
    struct inode *inode;
    int err, credits, retries = 0;

    err = dquot_initialize(dir);
    if (err)
        return err;

    credits = (EXT4_DATA_TRANS_BLOCKS(dir->i_sb) +
           EXT4_INDEX_EXTRA_TRANS_BLOCKS + 3);
retry:
    inode = ext4_new_inode_start_handle(dir, mode, &dentry->d_name, 0,
                        NULL, EXT4_HT_DIR, credits);
    handle = ext4_journal_current_handle();
```

```
    err = PTR_ERR(inode);
    if (!IS_ERR(inode)) {
        inode->i_op = &ext4_file_inode_operations;
        inode->i_fop = &ext4_file_operations;
        ext4_set_aops(inode);
        err = ext4_add_nondir(handle, dentry, inode);
        if (!err && IS_DIRSYNC(dir))
            ext4_handle_sync(handle);
    }
    if (handle)
        ext4_journal_stop(handle);
    if (err == -ENOSPC && ext4_should_retry_alloc(dir->i_sb, &retries))
        goto retry;
    return err;
}
```

```
#define ext4_new_inode_start_handle(dir, mode, qstr, goal, owner, \
            type, nblocks)        \
  __ext4_new_inode(NULL, (dir), (mode), (qstr), (goal), (owner), \
       0, (type), __LINE__, (nblocks))
```

```
struct inode *__ext4_new_inode(handle_t *handle, struct inode *dir,
                   umode_t mode, const struct qstr *qstr,
                   __u32 goal, uid_t *owner, __u32 i_flags,
                   int handle_type, unsigned int line_no,
                   int nblocks)
{
    inode = new_inode(sb);
}
```

我们经过一系列的分析，接下来的部分其实才是真正去创建 inode 结点的内容，整个调用过程是 new_inode -> new_inode_pseudo -> alloc_inode，在 `alloc_inode` 内部，会判断当前的超级块是否有设置 `alloc_inode`，没有的话则调用 `kmem_cache_alloc` 来创建。但是最后内部还是通过 `kmem_cache_alloc` 来创建。紧接着会通过 slab 分配器去分配一个记录 inode 相关的信息 `ext4_inode_info`，其中保存着对应的 inode 结点并返回。

```
struct inode *new_inode(struct super_block *sb)
{
    struct inode *inode;

    spin_lock_prefetch(&sb->s_inode_list_lock);

    inode = new_inode_pseudo(sb);
    if (inode)
        inode_sb_list_add(inode);
    return inode;
}
```

kmem_cache_alloc 的调用过程是 kmem_cache_alloc -> slab_alloc -> slab_alloc_node。slab_alloc_node 是一个分配内存的函数，首先会从当前 CPU 的私有自由链表中获取对象来完成内存分配。如果无法获取，则调用 __slab_alloc 进行更复杂的内存分配。

```
void *kmem_cache_alloc(struct kmem_cache *s, gfp_t gfpflags)
{
    void *ret = slab_alloc(s, gfpflags, _RET_IP_);

    trace_kmem_cache_alloc(_RET_IP_, ret, s->object_size,
                s->size, gfpflags);

    return ret;
}
```

```
static __always_inline void *slab_alloc(struct kmem_cache *s,
        gfp_t gfpflags, unsigned long addr)
{
    return slab_alloc_node(s, gfpflags, NUMA_NO_NODE, addr);
}
```

```
static __always_inline void *slab_alloc_node(struct kmem_cache *s,
        gfp_t gfpflags, int node, unsigned long addr)
{
    void *object;
    struct kmem_cache_cpu *c;
    struct page *page;
    unsigned long tid;
```

```
    // 会在分配之前做一些额外的检查和准备工作
    s = slab_pre_alloc_hook(s, gfpflags);
    if (!s)
        return NULL;
redo:
    // 获取当前 CPU 私有的内存缓存数据结构
    do {
        tid = this_cpu_read(s->cpu_slab->tid);
        c = raw_cpu_ptr(s->cpu_slab);
    } while (IS_ENABLED(CONFIG_PREEMPT) &&
         unlikely(tid != READ_ONCE(c->tid)));

    barrier();

    // 读取当前自由链表
    object = c->freelist;
    page = c->page;
    if (unlikely(!object || !node_match(page, node))) {
        // 从系统内存分配
        object = __slab_alloc(s, gfpflags, node, addr, c);
        stat(s, ALLOC_SLOWPATH);
    } else {
        void *next_object = get_freepointer_safe(s, object);


        if (unlikely(!this_cpu_cmpxchg_double(
                s->cpu_slab->freelist, s->cpu_slab->tid,
                object, tid,
                next_object, next_tid(tid)))) {

            note_cmpxchg_failure("slab_alloc", s, tid);
            goto redo;
        }
        prefetch_freepointer(s, next_object);
        // 更新统计数据，标记为使用了快路径
        stat(s, ALLOC_FASTPATH);
    }

    if (unlikely(gfpflags & __GFP_ZERO) && object)
```

```
        memset(object, 0, s->object_size);

    slab_post_alloc_hook(s, gfpflags, 1, &object);

    return object;
}
```

因为我们这里使用的是 ext4 文件系统，因此，前面分配结点的过程是在 `ext4_alloc_inode` 上面，调用 `kmem_cache_alloc` 从 `ext4_inode_cachep` 的内存池中分配一个 `ext4_inode_info*`，经过一系列的参数设置后，最后返回 inode 结点给到 `alloc_inode`中。

```
static struct inode *ext4_alloc_inode(struct super_block *sb)
{
    struct ext4_inode_info *ei;

    ei = kmem_cache_alloc(ext4_inode_cachep, GFP_NOFS);
    if (!ei)
        return NULL;

    inode_set_iversion(&ei->vfs_inode, 1);
    spin_lock_init(&ei->i_raw_lock);
    INIT_LIST_HEAD(&ei->i_prealloc_list);
    spin_lock_init(&ei->i_prealloc_lock);
    ext4_es_init_tree(&ei->i_es_tree);
    rwlock_init(&ei->i_es_lock);
    INIT_LIST_HEAD(&ei->i_es_list);
    ei->i_es_all_nr = 0;
    ei->i_es_shk_nr = 0;
    ei->i_es_shrink_lblk = 0;
    ei->i_reserved_data_blocks = 0;
    ei->i_da_metadata_calc_len = 0;
    ei->i_da_metadata_calc_last_lblock = 0;
    spin_lock_init(&(ei->i_block_reservation_lock));
    ext4_init_pending_tree(&ei->i_pending_tree);
#ifdef CONFIG_QUOTA
    ei->i_reserved_quota = 0;
    memset(&ei->i_dquot, 0, sizeof(ei->i_dquot));
#endif
    ei->jinode = NULL;
```

```
    INIT_LIST_HEAD(&ei->i_rsv_conversion_list);
    spin_lock_init(&ei->i_completed_io_lock);
    ei->i_sync_tid = 0;
    ei->i_datasync_tid = 0;
    atomic_set(&ei->i_unwritten, 0);
    INIT_WORK(&ei->i_rsv_conversion_work, ext4_end_io_rsv_work);
    return &ei->vfs_inode;
}
```

最后，我们回到最为重要的部分，也就是 `alloc_inode` 的内容，其中 `inode_init_always` 是对 inode 结点进行初始化的部分。

```
static struct inode *alloc_inode(struct super_block *sb)
{
    struct inode *inode;

    if (sb->s_op->alloc_inode)
        inode = sb->s_op->alloc_inode(sb);
    else
        inode = kmem_cache_alloc(inode_cachep, GFP_KERNEL);

    if (!inode)
        return NULL;

    if (unlikely(inode_init_always(sb, inode))) {
        if (inode->i_sb->s_op->destroy_inode)
            inode->i_sb->s_op->destroy_inode(inode);
        else
            kmem_cache_free(inode_cachep, inode);
        return NULL;
    }

    return inode;
}
```

`inode_init_always` 函数是在每次分配一个新的 inode 结构体的时候被调用。涉及到了 inode 的基本信息，例如超级块、块大小和 inode 标志等。同时还有与 inode 的权限、大小和计数有关，当然最为重要的是涉及到 inode 和其管理的文件数据的映射 `mapping`，这里会在初始化的时候设置一个 GFP 标志 `GFP_HIGHUSER_MOVABLE`。

```c
/**
 * inode_init_always - perform inode structure initialisation
 * @sb: superblock inode belongs to
 * @inode: inode to initialise
 *
 * These are initializations that need to be done on every inode
 * allocation as the fields are not initialised by slab allocation.
 */
int inode_init_always(struct super_block *sb, struct inode *inode)
{
    static const struct inode_operations empty_iops;
    static const struct file_operations no_open_fops = {.open = no_open};
    struct address_space *const mapping = &inode->i_data;

    inode->i_sb = sb;
    inode->i_blkbits = sb->s_blocksize_bits;
    inode->i_flags = 0;
    atomic_set(&inode->i_count, 1);
    inode->i_op = &empty_iops;
    inode->i_fop = &no_open_fops;
    inode->__i_nlink = 1;
    inode->i_opflags = 0;
    if (sb->s_xattr)
        inode->i_opflags |= IOP_XATTR;
    i_uid_write(inode, 0);
    i_gid_write(inode, 0);
    atomic_set(&inode->i_writecount, 0);
    inode->i_size = 0;
    inode->i_write_hint = WRITE_LIFE_NOT_SET;
    inode->i_blocks = 0;
    inode->i_bytes = 0;
    inode->i_generation = 0;
    inode->i_pipe = NULL;
    inode->i_bdev = NULL;
    inode->i_cdev = NULL;
    inode->i_link = NULL;
    inode->i_dir_seq = 0;
    inode->i_rdev = 0;
    inode->dirtied_when = 0;
```

```
#ifdef CONFIG_CGROUP_WRITEBACK
    inode->i_wb_frn_winner = 0;
    inode->i_wb_frn_avg_time = 0;
    inode->i_wb_frn_history = 0;
#endif

    if (security_inode_alloc(inode))
        goto out;
    spin_lock_init(&inode->i_lock);
    lockdep_set_class(&inode->i_lock, &sb->s_type->i_lock_key);

    init_rwsem(&inode->i_rwsem);
    lockdep_set_class(&inode->i_rwsem, &sb->s_type->i_mutex_key);

    atomic_set(&inode->i_dio_count, 0);

    mapping->a_ops = &empty_aops;
    mapping->host = inode;
    mapping->flags = 0;
    mapping->wb_err = 0;
    atomic_set(&mapping->i_mmap_writable, 0);
    mapping_set_gfp_mask(mapping, GFP_HIGHUSER_MOVABLE);
    mapping->private_data = NULL;
    mapping->writeback_index = 0;
    inode->i_private = NULL;
    inode->i_mapping = mapping;
    INIT_HLIST_HEAD(&inode->i_dentry);  /* buggered by rcu freeing */
#ifdef CONFIG_FS_POSIX_ACL
    inode->i_acl = inode->i_default_acl = ACL_NOT_CACHED;
#endif

#ifdef CONFIG_FSNOTIFY
    inode->i_fsnotify_mask = 0;
#endif
    inode->i_flctx = NULL;
    this_cpu_inc(nr_inodes);

    return 0;
```

```
out:
    return -ENOMEM;
}
```