block inode dentry

1、首先 block 这没什么好说的了，就是底层磁盘存储文件信息及文件的playload数据信息的单位

2、inode 文件在磁盘中存在的对象。 inode 块中信息，表明磁盘块中属于此文件对象的块是哪些块

3、dentry，被用来索引和访问 inode，每个dentry对应了一个 inode，通过dentry可以找到并操作对应的inode。

注意：一个路径字符串，在内核中，会被解析为一组路径节点 object，dentry就是路径中的一个节点。 与 inode 和 super block不同的是，dentry在磁盘中其实没有实体存储，dentry链表都是在运行时通过解析 path 字符串构造出来的

dentry

一个路径字符串，在内核中会被解析为一组路径节点object。dentry就是路径中的一个节点。

dentry被用来索引和访问inode，每个dentry对应一个inode，通过dentry可以找到并操作其所对应的inode。

与inode和super block不同，dentry在磁盘中并没有实体储存，dentry链表都是在运行时通过解析path字符串构造出来的。

dentry cache

由于每次通过path构造出dentry是一个费时过程，且相同路径没必要每次重复解析。开发者为dentry添加了一个缓存，dentry cache(或dcache)

dcache是一个slab cache，在 `dcache_init` 过程中被初始化，保存在 `dentry_cache` 全局变量中，所有dentry会通过 `d_alloc` 接口，从这个缓存池分配。

`dentry_hashtable` 是dcache的索引表， `d_lookup` 函数利用这个表可以快速找到path string对应的被缓存的dentry object。

dentry status

dentry有三种状态：

- used： 指向一个有效inode，且d_count不为0（有效的）
- unused： 指向一个有效inode，但d_count为0（缓存中）
- negtive： 指向一个NULL inode（无效的，缓存中）

inode中的i_dentry，保存了那些正在使用中的dentry。

dentry所在super block的s_dentry_lru中，保存了可以被随时回收释放的dentry。

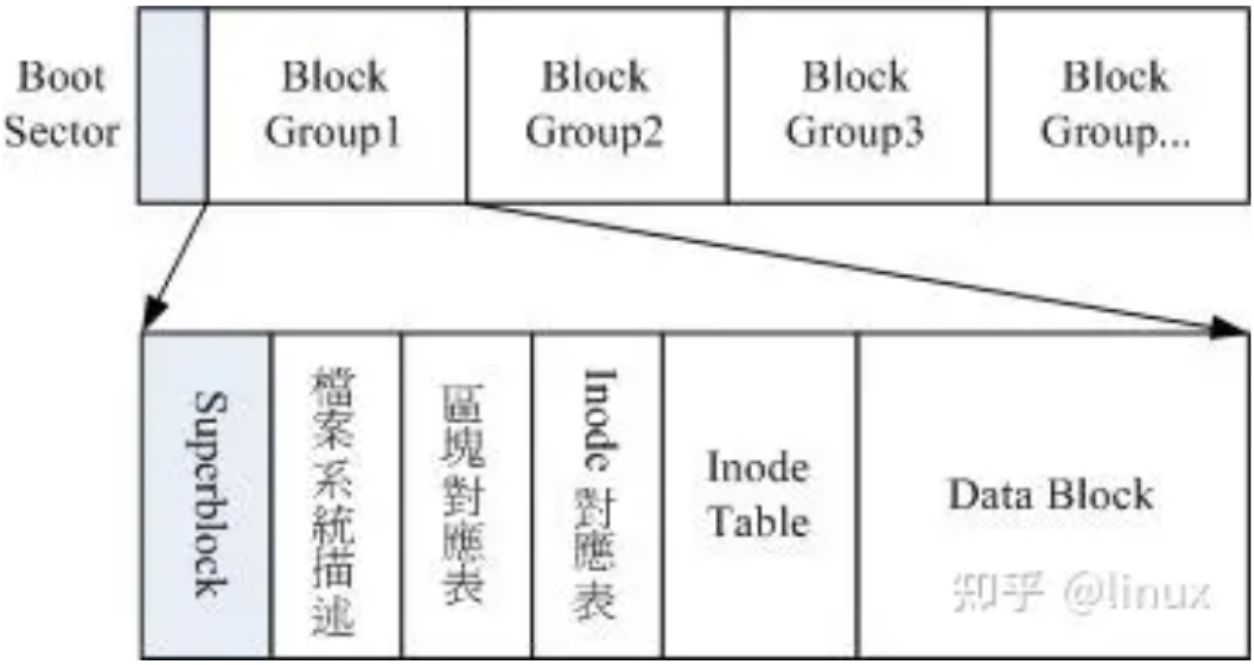
Structure

```
dentry
d_name          // 本级路径节点名
d_parent        // 指向上级目录节点dentry
d_inode         // 本级目录节点对应的文件inode
d_sb            // 关联的super block
d_op            // dentry操作
d_mounted       // 是否是一个mount point
d_count         // 使用引用数量，refcount
```

inode：

我们的磁盘在进行分区、格式化的时候会分为两个区域，一个是 数据区，用于存储文件中的数据；另一个是 inode区，用于存放 inode table （inode表），inode table 中存放的是一个一个的 inode （也称为inode节点），不同的 inode 就可以表示不同的文件，每一个文件都必须对应一个 inode，inode 实质上是一个结构体，这个结构体中有很多的元素，不同的元素记录了文件了不同信息，譬如：

- 文件字节大小
- 文件所有者
- 文件对应的读/写/执行权限
- 文件时间戳（创建时间、更新时间等）
- 文件类型
- 文件数据存储的block（块）位置
- .....



inode

> "inode is a file or directory of a filesystem"

inode是文件系统中文件和目录对象。

inode and filesystem

每个inode在一个挂载的文件系统中，有一个唯一编号：inode ID，文件的inode ID可以通过 `ls -i` 查看。

一个filesystem，inode ID的总量是固定的，可以通过 `df -i` 查看inode余量。

inode and blocks

inode保存在内存中，inode所表示的文件的内容，保存在磁盘中（或内存）。

一个inode关联了其磁盘中物理block的链表，通过inode就可以从磁盘对应的sectors中读取到最终的文件内容。

磁盘的物理block一般为512字节，而访问磁盘时，为提高性能，一般会一次读取8个block，叫做一个4K IO Block

inode and dentry

inode在VFS中，通过dentry进行索引和操作。shell的文件访问命令，会通过系统调用，经过VFS框架，最终回调到对应的inode operation中。

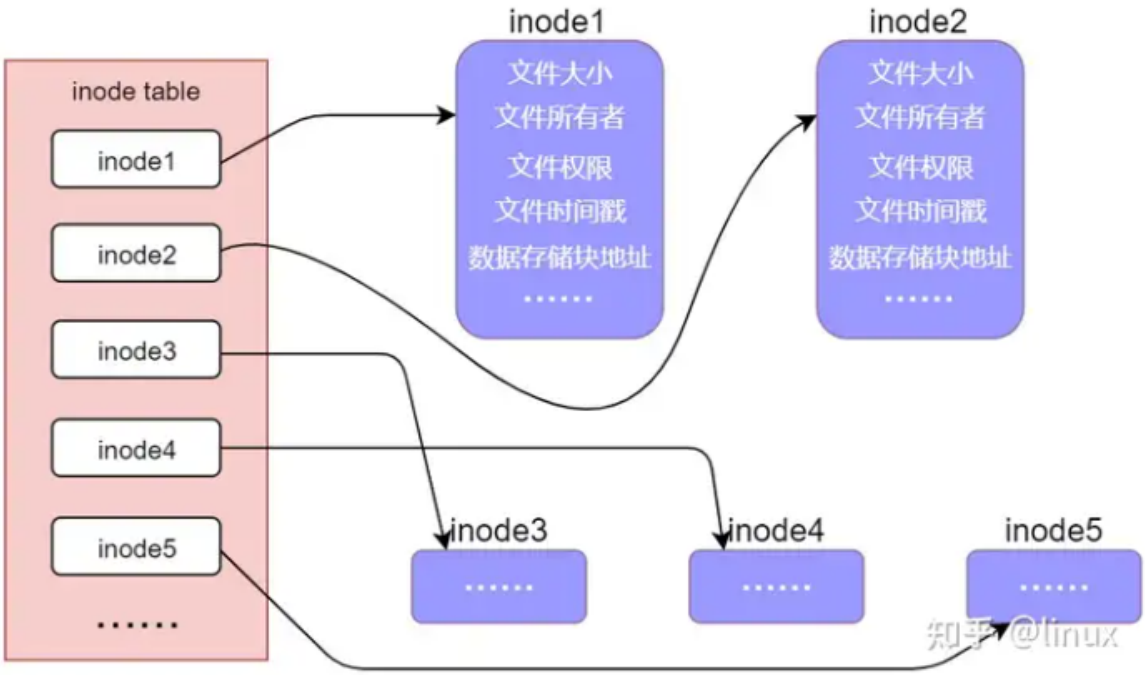
inode的operation通过传入的dentry和父目录inode，访问和操作出dentry对应的inode，最后通过 `d_instantiate` 将操作后的inode重新绑定到dentry中，完成一次文件操作。

Structures

inode结构体很大，但是并非所有property都被使用，使用哪些property取决于文件系统的实现。

这里主要列出与其他vfs class有关联的inode property。

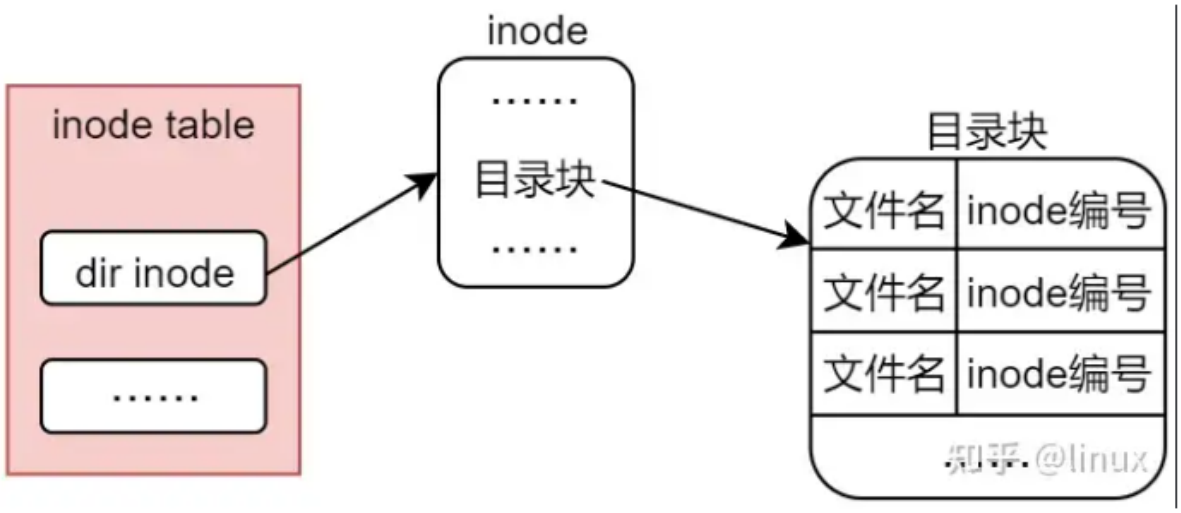
```
inode
i_op            // inode operation
i_sb            // 关联的super block
i_fop           // file operation
i_ino           // inode ID
i_dentry        // 文件的dentry节点
...
```



所以由此可知， inode table✦ 本身也需要占用磁盘的存储空间。在同一个文件系统中，每一个文件都有唯一的一个 inode ， 每一个 inode 都有一个与之相对应的数字编号 ， 内核可以根据索引节点号的哈希值✦查找其 inode 结构，前提是内核要知道索引节点号和对应文件所在文件系统的超级块对象的地址。 在Linux系统下，我们可以通过"ls -i"命令查看文件的 inode编号✦ ，如下所示:

- 普通文件由 `inode节点` 和 `数据块` 构成
- 目录由 `inode节点` 和 `目录块` 构成

其存储形式如下图所示:



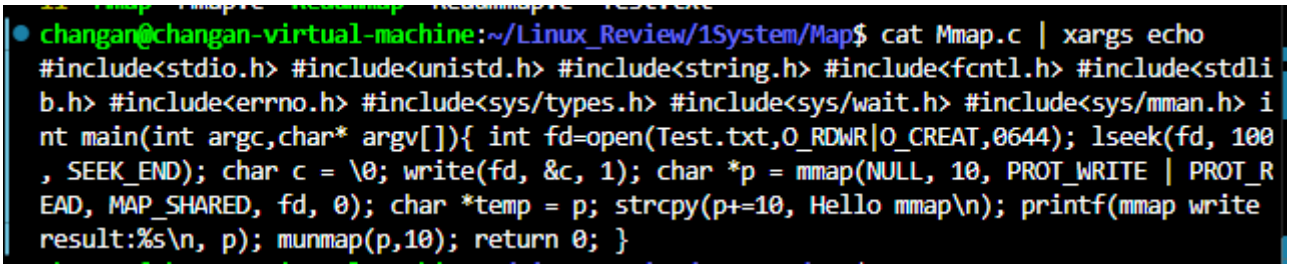
对于 dentry 和 inode 的区别可以如此总结:

- dentry 结构代表的是逻辑意义上的文件，描述的是文件逻辑上的属性， 目录项对象在磁盘上并没有对应的映像 。
- inode 结构代表的是物理意义上的文件，记录的是物理上的属性，对于一个具体的文件系统，其inode结构在磁盘上就有对应的映像 。

xargs 的有趣使用

把 **终端输出** 转为 **终端输入**

1、不仅可以给 一般的系统调用使用



2、可以进行 可执行程序运行时直接的传参！ 有一个 print 程序： 其内部代码如下

```
#include<stdio.h>

int main(){
    printf("%d %d", 1, 2);

    return 0;
}
```

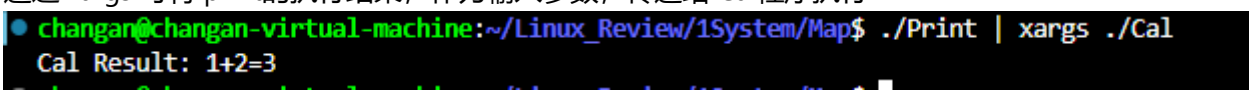
另外有一个 Cal 程序： 其内部代码如下

```
#include<stdio.h>

int main(int argc,char* argv[]){
    if(argc==1){
        printf("No Argumetns! exit\n");
    }else if(argc<=2){
        printf("Test Failed Plase Restart\n");
    }else{
        int a = *argv[1] - '0';
        int b = *argv[2] - '0';
        printf("Cal Result: %d+%d=%d\n", a, b, a + b);
    }

    return 0;
}
```

通过 xargs 可将 print的执行结果，作为输入参数，传递给 Cal程序执行



注意： 对于 一个字符串 str指针 p来说， printf p 则打印 str字符串; 而对 p进行解引用，则结果相当于 str 的首字母！！