# NBD 网络块设备-内核篇

前面写过nbd的协议和实践篇，大致讲清楚了nbd的使用和用户态实现。

我们知道，使用linux自带的nbd设备，需要有用户态和内核态两部分。其中，用户态通过ioctl命令对核态nbd设备进行控制。查看内核nbd模块的实现，可以看到，大致上存在以下几种控制命令：

1. 设置nbd设备client端与server端的连接套接字命令：NBD\_SET\_SOCK；
2. Client端进入内核循环，发送请求并接收回复的命令：NBD\_DO\_IT；
3. 设置socket超时时间的命令：NBD\_SET\_TIMEOUT；
4. 断开client与server端连接的命令：NBD\_DISCONNECT；
5. Client端设置块设备的命令：NBD\_SET\_SIZE\_BLOCKS。

内核定义详细命令列表如下：

**#define** NBD\_SET\_SOCK \_IO( 0xab, 0 )

**#define** NBD\_SET\_BLKSIZE \_IO( 0xab, 1 )

**#define** NBD\_SET\_SIZE \_IO( 0xab, 2 )

**#define** NBD\_DO\_IT \_IO( 0xab, 3 )

**#define** NBD\_CLEAR\_SOCK \_IO( 0xab, 4 )

**#define** NBD\_CLEAR\_QUE \_IO( 0xab, 5 )

**#define** NBD\_PRINT\_DEBUG \_IO( 0xab, 6 )

**#define** NBD\_SET\_SIZE\_BLOCKS \_IO( 0xab, 7 )

**#define** NBD\_DISCONNECT \_IO( 0xab, 8 )

**#define** NBD\_SET\_TIMEOUT \_IO( 0xab, 9 )

**#define** NBD\_SET\_FLAGS \_IO( 0xab, 10)

下面从几个部分介绍NBD设备内核态与用户态的配合使用，过程中，理解linux内核nbd模块的实现原理。

1. **NBD client与server的交互模型**

我们已经通过nbd协议知道，nbd设备的使用需要经过两个阶段

1. client与server握手阶段：这个阶段在用户态完成，主要完成一些参数的设定，使得客户端感知server端实际“物理设备”的一些属性以及协议参数的协商。
2. 数据传输阶段：这个阶段用户态的client进程和server进程不再直接通信，而是通过nbd内核模块（nbd设备驱动）进行通信，传输数据。

由此NBD设备交互的组件如下：

* **nbd设备的驱动程序**
* **nbd client端进程**
* **nbd server端进程**

nbd设备的工作模型如下：



可以知道nbd设备的交互模型大致要点如下：

1. nbd client所在主机需要加载nbd内核模块（server端不需要）；
2. nbd client进程与server进程在用户态建立socket连接，并完成握手阶段；
3. 握手结束，nbd client端将socket告知内核，并由内核nbd驱动生成出虚拟的nbd块设备（/dev/nbd\*），关联到socket连接到的远端实际“物理设备”（远端的本地块设备或文件）；
4. 数据传输阶段，nbd的请求封装（封装成nbd协议）由nbd内核驱动完成的，并通过socket发送到远端。而远端server的回复，则是通过用户态进程完成协议封装并通过socket回复的。

**2. nbd 用户态的实现**

笔者分析的是nbd-3.10版本，用户态代码分析并非本文的重点，其完全按照nbd协议实现。大致流程是：

**1）握手；**

**2）client通过ioctl命令设置内核nbd虚拟设备参数；**

**3）client调用NBD\_DO\_IT进入数据传输阶段循环。**

Client端nbd虚拟设备一旦生成，任何对设备（/dev/nbd\*）的操作都会走内核的IO栈，最终走到nbd驱动模块封装成nbd协议的请求通过socket发送给server端，IO在server端落盘。

代码主要包含client和server两个组件，感兴趣者可以自行分析。

**3. nbd内核代码分析(linux-4.x版本)**

3.1 内核nbd设备

内核nbd设备的nbd\_device结构体主要字段定义如下：

**struct** nbd\_device {

// NBD设备标志

u32 flags;

// 指向client连接到server端的套接字结构。

// 用户空间客户端程序通过： ioctl命令NBD\_SET\_SOCK将连接好的套接字文件句柄传递到内核驱动程序。

**struct** socket \* sock;

// nbd magic

**int** magic;

// 请求队列锁

spinlock\_t queue\_lock;

// 该结构，用于nbd客户端发送请求后等待服务端回复的队列，用于保存请求的上下文

// 客户端接收到请求以后，从该队列中查找到客户端的请求，向上层报告。同时从队列中移除已经完成的请求。

**struct** list\_head queue\_head; /\* Requests waiting result \*/

**struct** request \*active\_req;

wait\_queue\_head\_t active\_wq;

// 待发送请求的队列

**struct** list\_head waiting\_queue; /\* Requests to be sent \*/

wait\_queue\_head\_t waiting\_wq;

**struct** mutex tx\_lock;

// nbd设备的通用磁盘结构结构体

**struct** gendisk \*disk;

// 块设备size

**int** blksize;

loff\_t bytesize;

// 超时时间，nbd客户端使用：ioctl命令NBD\_SET\_TIMEOUT设置

**int** xmit\_timeout;

// 用户断开连接的标志

bool disconnect; /\* a disconnect has been requested by user \*/

// 设置socket超时时使用的定时器

**struct** timer\_list timeout\_timer;

spinlock\_t tasks\_lock;

// 接收server端回复的进程

**struct** task\_struct \*task\_recv;

// client端发送进程

**struct** task\_struct \*task\_send;

};

3.2 报文结构体

（1）nbd请求报文

Client端需要对请求的数据进行nbd协议的封装，封装报文结构体request定义如下：

**struct** nbd\_request {

\_\_be32 magic; // nbd请求magic, 0x25609513

\_\_be32 type; /\* == READ || == WRITE \*/ // 请求类型

**char** handle[8]; // 请求报文句柄为请求的内存地址

\_\_be64 from; // 请求偏移量，磁盘的开始位置

\_\_be32 len; // 请求内容长度（字节为单位）

} **\_\_attribute\_\_**((packed));

如果是写请求，报文后面紧跟len字节的数据。

1. nbd回复报文

Server端对收到的请求进行相应，并发送回复报文，封装的回复报文结构体response定义如下：

**struct** nbd\_reply {

\_\_be32 magic; // nbd回复magic，0x67446698

\_\_be32 error; /\* 0 = ok, else error \*/

**char** handle[8]; /\* handle you got from request \*/ // 从请求报文中拷贝的句柄，当收到回复报文后，移除回复等待队列中的request请求

};

如果是读请求的回复，成功时，报文后面紧跟len字节的数据。

3.3 流程分析

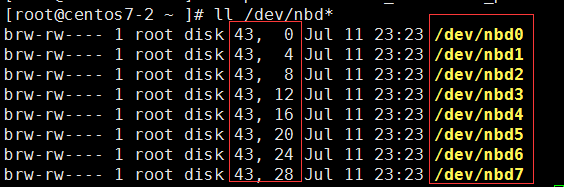
3.3.1 模块初始化

按照linux内核通用的模块机制，当执行modprobe nbd时，执行宏module\_init定义的初始化函数，这里是nbd\_init。其中，nbd模块加载时，有两个模块参数可以传入

1）nbds\_max：表示初始化生成的nbd设备数量（/dev/nbd\*），默认是16个（nbd0-nbd15）。

2）max\_part：表示每个nbd设备的最大分区（由于linux块设备的每一个分区占用一个次设备号，所以这里也即最大次设备数目），这里用户传入的数字会转化成满足“>=max\_part”条件的最小2的次幂的数。例如：传入3，则内核会允许最大分区（次设备）数为4。

示例：modprobe nbd nbds\_max=8 max\_part=3



nbd\_init初始化流程主要如下：

1. nbd\_part和nbds\_max参数校验；
2. 内核分配nbd\_device设备内存；
3. 为每一个nbd设备分配gendisk通用磁盘结构体，并初始化gendisk的请求队列，注册请求队列的处理函数为**nbd\_request\_handler**，gendisk的一些参数设置；
4. 注册nbd设备的主设备号为43；
5. 初始化每一个nbd设备的各个字段，主要是定时器和队列，设定次设备的起始号（例如nbd0次设备号0起始，nbd1次设备号4起始）。**注册通用磁盘设备gendisk的块设备驱动为nbd\_fops，即用户态ioctl在内核的回调。**

**static** **const** **struct** block\_device\_operations nbd\_fops =

{

.owner = THIS\_MODULE,

.ioctl = **nbd\_ioctl**,

};

代码可自行分析。

3.3.2 模块退出

按照linux内核通用的模块机制，当执行rmmod nbd时，执行宏module\_init定义的初始化函数，这里是nbd\_cleanup。

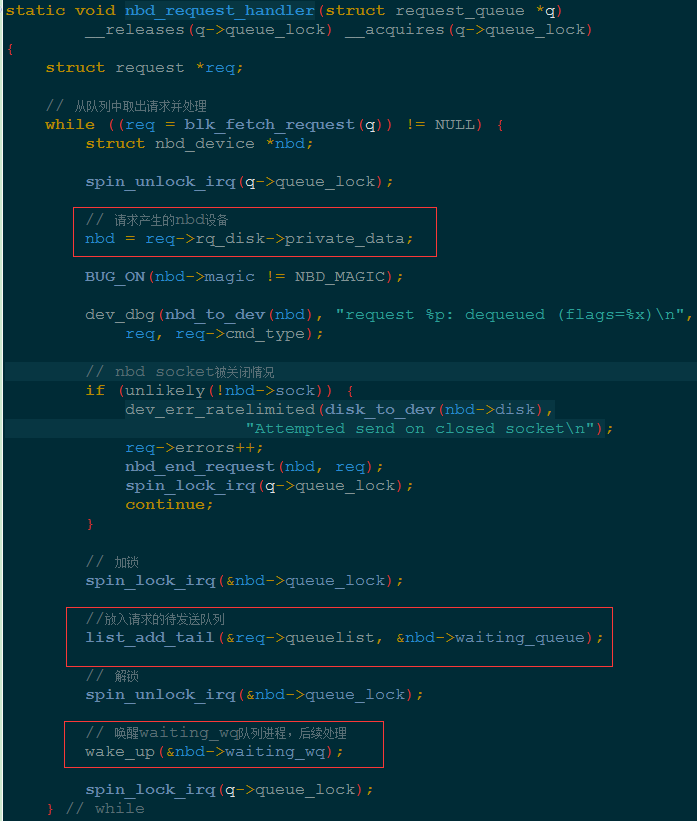
nbd\_cleanup主要完成通用磁盘结构体gendisk的卸载和释放，以及nbd设备的解注册和释放。

3.3.3 nbd\_request\_handler

我们了解到，每一个nbd设备关联到了一个通用磁盘gendisk结构体，而这个磁盘的请求处理函数被注册为nbd\_request\_handler函数，也就是说，当用户态对/dev/nbd\*设备进行读写操作时，都会最终回调到这里。事实上，根据代码可以看到，这里的处理十分简单，

1. 找到该磁盘对应的核态nbd设备
2. 将请求插入到该nbd设备的“等待发送队列”中
3. 唤醒client发送进程

如下：



3.3.4 nbd\_ioctl

nbd\_ioctl被注册为nbd块设备操作ioctl的回调函数，调用关系nbd\_ioctl –>\_\_nbd\_ioctl。

重点分析\_\_nbd\_ioctl函数。

\_\_nbd\_ioctl是典型的ioctl回调处理逻辑：

static int \_\_nbd\_ioctl(struct block\_device \*bdev, struct nbd\_device \*nbd,

unsigned int cmd, unsigned long arg)

{ switch(cmd) {

case x:

case y:

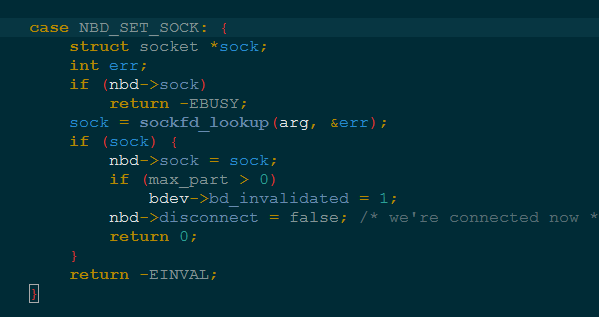
…

}

}

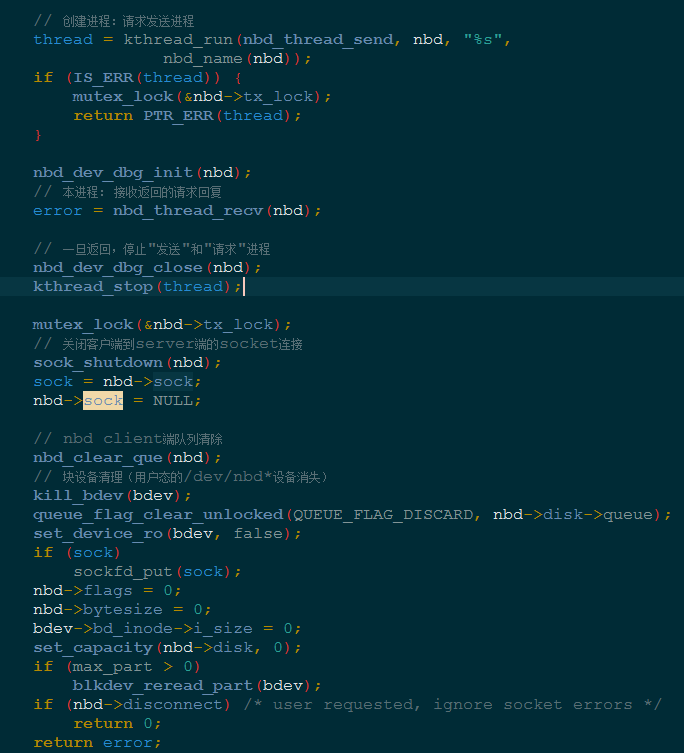
其中x、y就是nbd设备支持的ioctl命令，前文已经提到。其中最重要的就是NBD\_SET\_SOCK和NBD\_DO\_IT。

（1）NBD\_SET\_SOCK告知内核，client端和server端使用的socket，nbd设备会找到socket的指针，设置到nbd设备的sock字段上。



1. NBD\_DO\_IT是nbd驱动模块的核心功能，当用户态调用该ioctl时，正常情况不会返回，直到出错或者主动断开。

核心代码如下：



内核创建了一个核态线程（nbd\_thread\_send），持续不断的从nbd设备的“等待发送队列”取请求，通过socket发到server端，未收到回复的请求插入到“等待回复队列”中。同时本进程进入接收回复的循环中(nbd\_thread\_recv)，收到的回复根据handler字段（还记得前文的nbd回复报文协议吗？）找到请求，从nbd设备的“等待回复队列”中删除该完成的请求。

至此，大致分析清楚了nbd设备的大致工作模型、原理和重要代码流程。nbd模块是一个典型的支持ioctl命令的块设备驱动，核心代码量只有1000 LOC+，里面涉及到编写一个块设备驱动的方方面面，包含信号量屏蔽、socket处理、内核数据读写等内核通用处理机制，十分适合学习和编写内核驱动者走读。