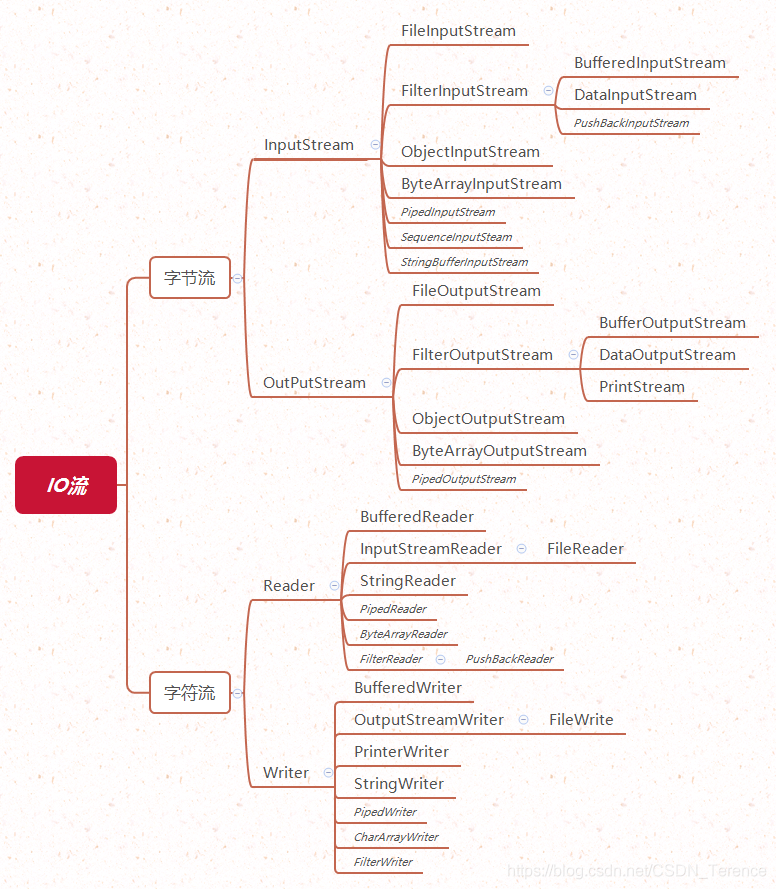
# IO

# Java IO

## BIO：传统的阻塞式IO



### BIO中又分为：

字节流IO：InputStream/OutputStream

字符流IO：Reader/Writer

### 常用的BIO类：

#### 字节流：

FileInputStream/FileoutputStream 最基础的字节流

BufferedInputStream/BufferedOutputStream 面向缓冲区的，减少系统调用次数。

DataInputStream/DataOutputStream：基本类型持久化，序列化的类

ObjectInputStream：对象持久化，序列话的类（需要实现Serializable接口）

#### 字符流：

FileReader/FileWriter 最基础的字符流

BufferedReader/BufferedWriter 面向缓冲区的字符流

#### 特殊的流：

RandomAccessFile：随机读写流，有seek指针，可以同时read/write，还可以使用mmap的，映射pageCache地址空间使用，提神效率。

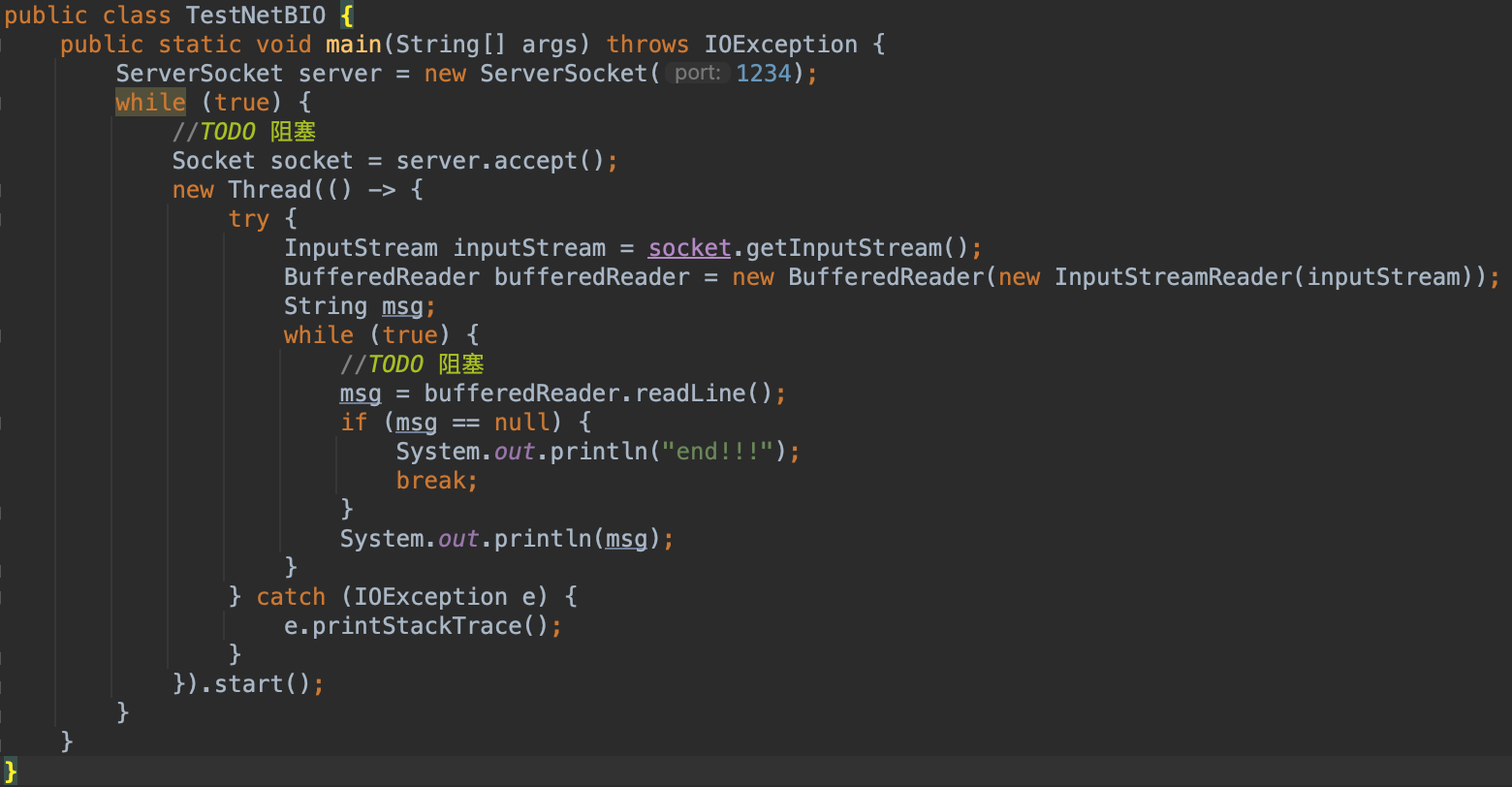
#### 网络IO：

ServerSocket

Socket socket = serverSocket.accept() 阻塞

reader = Socket.getInputStream()

reader.readLine() 阻塞



#### 总结：

BIO中所有的操作都是阻塞的，文件IO好像没有啥影响，但是面向网络IO的时候，就会有大问题：

1:SocketServer.accept是阻塞的，所以需要一个线程去维护 accept

2:Socket的read是阻塞的，所以要一个线程去维护一个Socket

这样无法解决C10K的问题，一台机器能创建的线程数量是有限的

## NIO：New IO（非阻塞的）

传统IO是面向**单向流**的，**阻塞**的，比如

网络：

Socket.getInputStream/getOutputStream

文件：

new FileOutputStream(new File())/new FileInutStream(new File())

而NIO是面向**Channel**，**缓冲区的**，**非阻塞式**的（**可配置/网络IO**），比如

网络：

ServerSocketChannel serverChannel = ServerSocketChannel.open()

serverChannel.configureBlocking(false)

文件：

FileChannel fileChannel = FileChnnel.open()

### NIO关键类：

#### ServerSocketChannel：

对应BIO中的ServerSocket

#### SocketChannel：

对应BIO中的Sokcet

#### FileChannel：

双向读写，对应BIO中的InputStream和OutputStream

#### Selector:

多路复用器，底层可以是select/poll/epoll

#### ByteBuffer/MappedByteBuffer：

读写缓冲区，对应了我们自己定义的bytes[]

#### ByteBuffer详细使用：

ByteBufferbuffer=ByteBuffer.allocateDirect(1024);

关键三个指针，position，limit，capacity

position: 当前操作的起始位置，读开始位置 或者写开始位置

limit：辅助标记位

capacity：buffer的容量

1：buffer.put(bytes)

写数据，position会往后移

2：buffer.flip

读写交替，position执行指到初始位置，limit指到position的位置。

3：buffer.get()

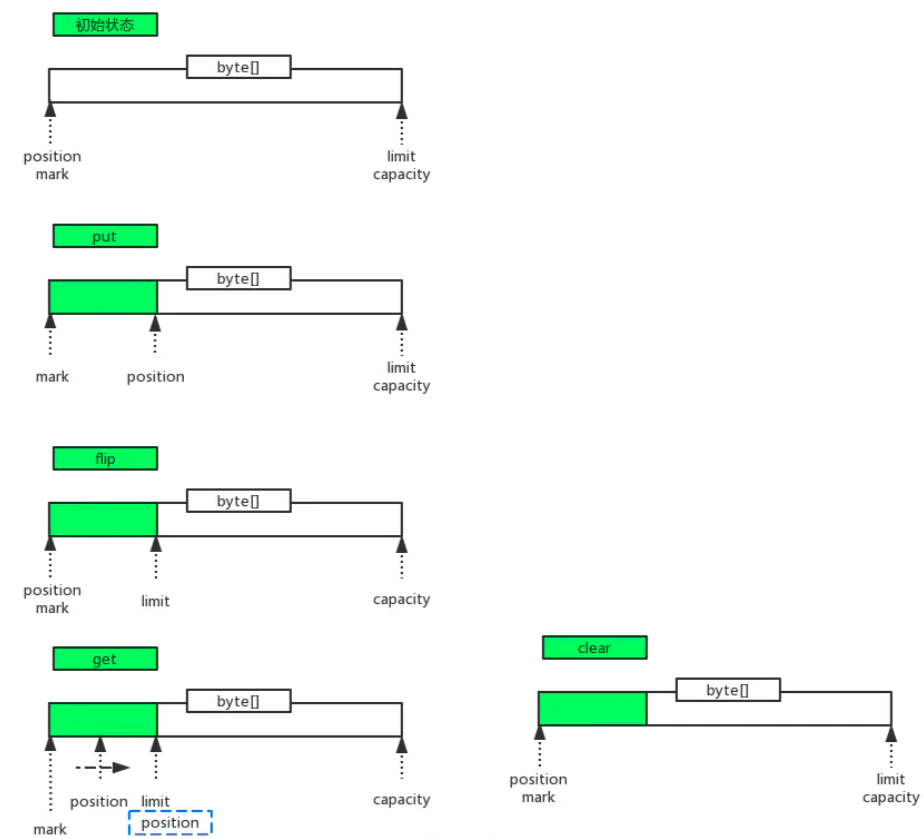
读数据，position往后移

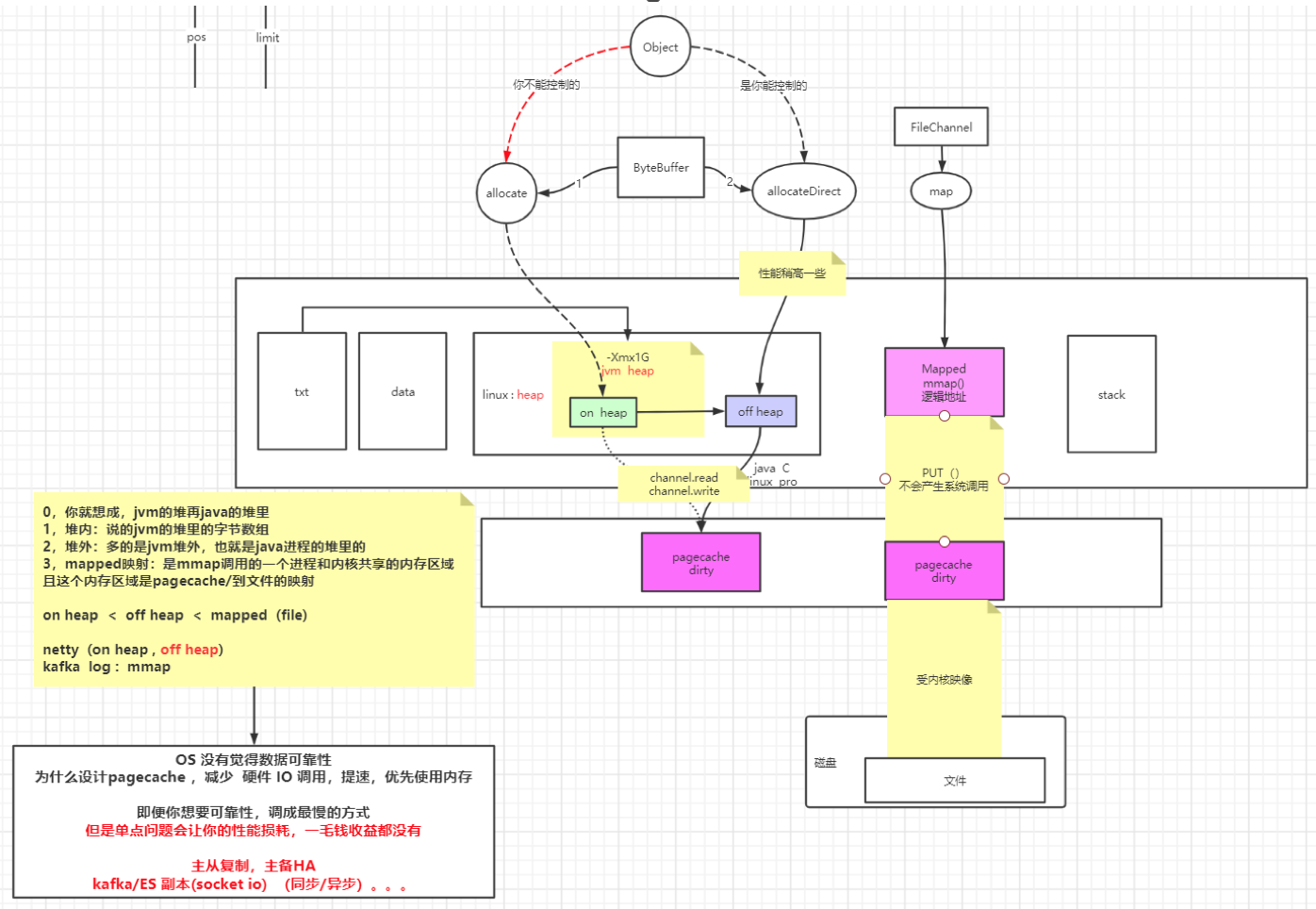
4：buffer.campat()

读完了之后，进行压缩，读过的数据就清除了，position位置到limit，limit到capacity

5：buffer.clear()

初始化，position = 0 limit=capacity





# 其他IO

## Linux网络IO：

<https://zhuanlan.zhihu.com/p/127170201>

### 前置知识：

在网络中读取数据主要分为两个步骤

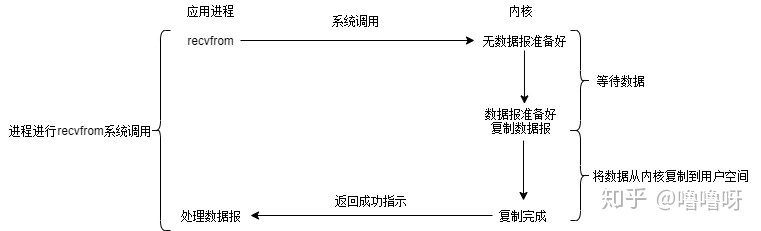
1:等待数据（socket.read/socketServer.accpet）

2:拷贝数据（内核空间拷贝到用户空间）

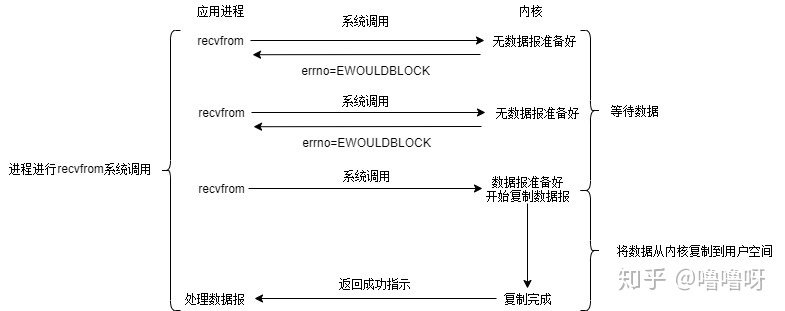
阻塞非阻塞：在等待数据阶段，如果没有数据是否会把调用者阻塞住。

同步非同步：在数据拷贝阶段，是否是应用程序自己拷贝

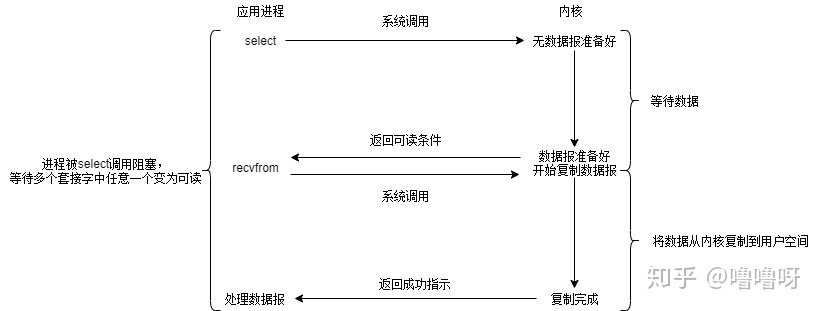
### 1:阻塞IO



### 2:非阻塞IO

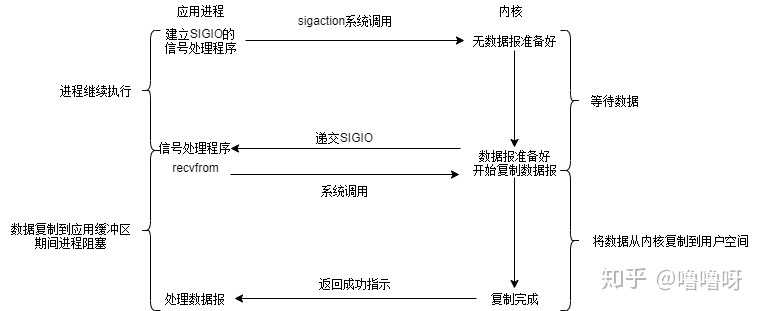


### 3:多路复用IO



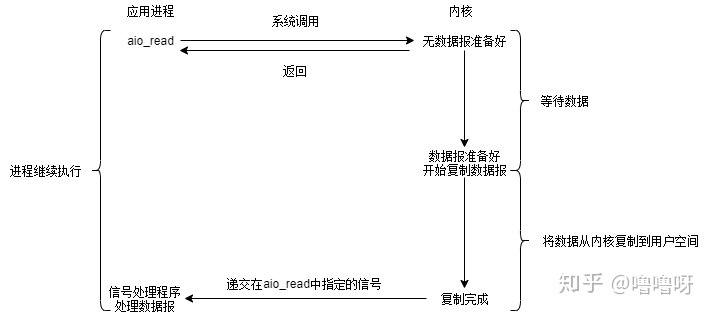
### 4:信号驱动IO

异步IO的变种，不用一直询问，而是传一个回调函数，来数据了，调用我的回调函数即可。在没数据来之前可以做其他事



### 5:异步IO

与信号驱动IO区别是，信号驱动IO在数据拷贝阶段也是自己拷贝的，阻塞式的，而异步IO，不是自己拷贝的，并且数据准备和拷贝阶段都是非阻塞的。



### 注意：

除了5：异步IO 之外，前面4个都是同步IO（自己拷贝数据到用户空间）

# 多路复用器：

<https://www.cnblogs.com/Courage129/p/14295476.html>

## Select：

主要流程，每次用户线程 将所有的 文件描述符传给内核，然后内核一次遍历所有的fds是否有数据，然后返回结果；

### 问题：

1:select对于文件描述符的数量有限制，是1024（受限于FD\_SETSIZE）

2:需要用户空间一直维护fds，并且每次调用都要从用户空间拷贝到内核空间

3:select采用遍历的方式，比较慢

## Poll：

解决了1024的问题，其他的和select差不多

## Epoll：

是poll和select的改良版，解决了select的那些问题：

1：没有上限（只受限于系统可以打开文件的数量）

2：内核空间维护一个红黑树（可以快速判断是否是重复添加），用来维护fds，不用每次用户空间->内核空间拷贝

3:维护一个双向链表，有事件就往链表中加，用户空间获取的时候，字节返回链表（网卡回调函数实现的）

主要函数：

epoll\_create：创建epoll

epoll\_ctl：注册事件

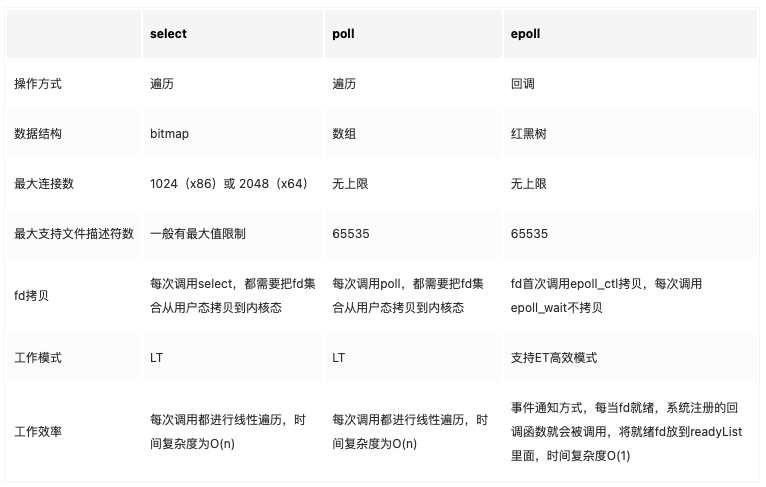
epoll\_wait：获取活跃事件

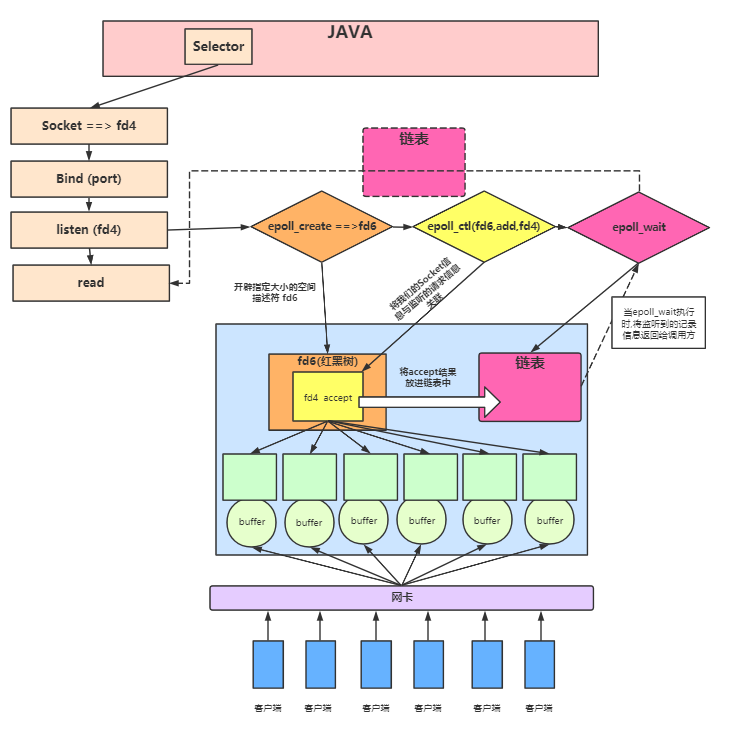
工作模式：

LT模式：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll\_wait时，会再次响应应用程序并通知此事件。

ET模式：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll\_wait时，不会再次响应应用程序并通知此事件。

<https://juejin.cn/post/6882984260672847879>





# Linux中IO相关知识：

mmap/sendfile/pagecache

重定向 2>&1 注意如果右边是 文件描述符，那么应该加&

0:标准输入

1:标准输出

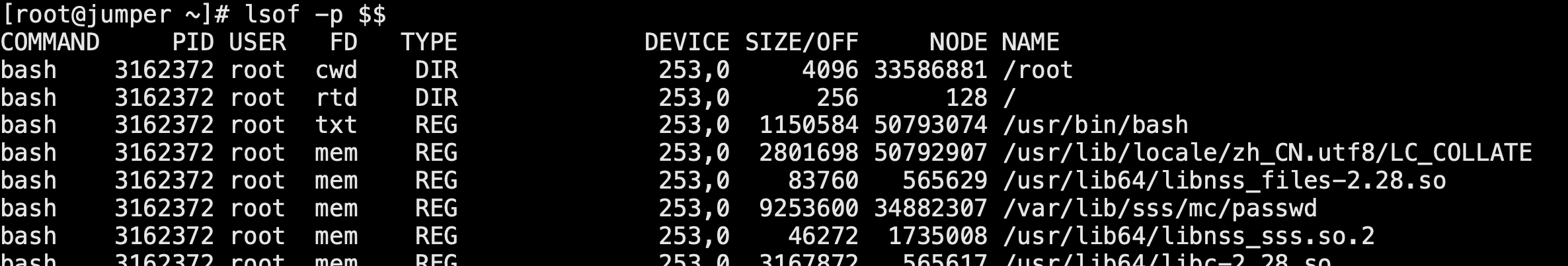
2:错误输出

**注意：一个文件描述符对应一个文集**

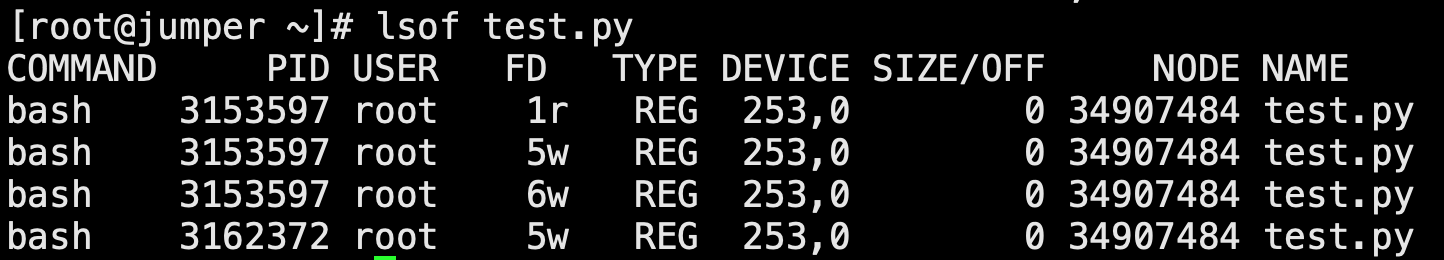
<https://linuxtools-rst.readthedocs.io/zh_CN/latest/tool/lsof.html>

### lsof命令：

查看进程打开了哪些文件lsof –op pid



查看文件被哪些进程打开了 lsof 文件名



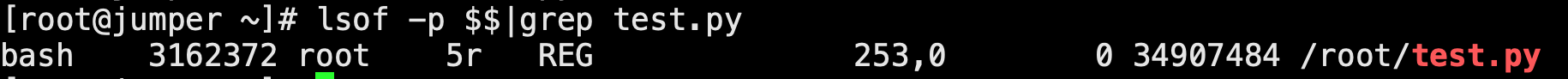
### exec命令

bash中手动打开文件：

exec 5>hello.py (w模式)



exec 5<hello.py(r模式)



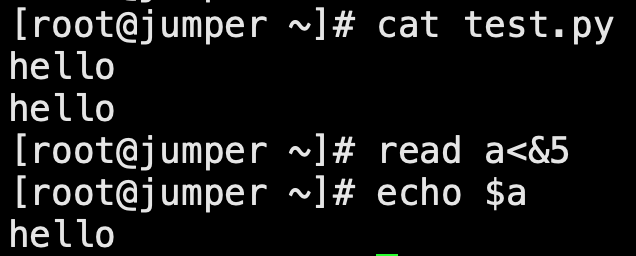
exec 5<>hello.py（rw模式）



### read命令：

从文件描述符读取数据到变量

read [var] <& fd



### exec高阶玩法：

1:使用命令创建tcp连接，并指定文件描述符

exec 8<> /dev/tcp/www.baidu.com/80

2:往文件描述符写数据

echo -e 'GET /HTTP/1.0\n' 1>& 8

3:从文件描述符读数据出来

cat 0<& 8