在高并发的，高吞吐的 IO 系统中常常见到 epoll 的身影。

后台的程序只需要 1 个就可以负责管理多个 fd 句柄，负责应对所有的业务方的 IO 请求。这种一对多的 IO 模式我们就叫做 IO 多路复用。

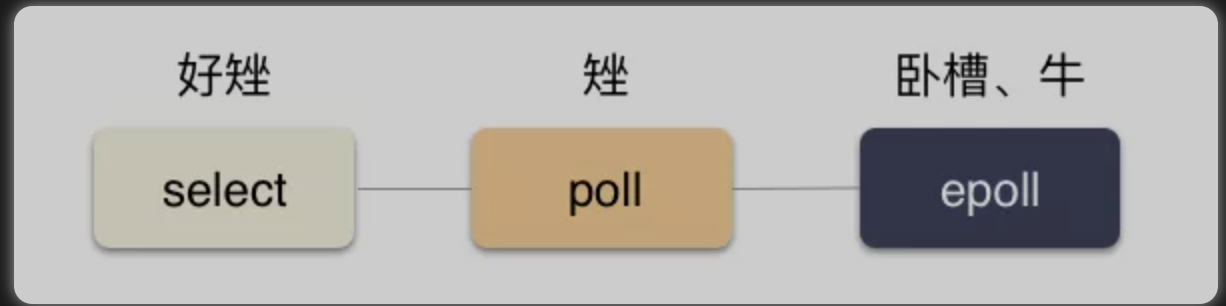
多路是指？多个业务方（句柄）并发下来的 IO 。

复用是指？复用这一个后台处理程序。

那有人又问了，那我一个 IO 请求（比如 write ）对应一个线程来处理，这样所有的 IO 不都并发了吗？是可以，但是有瓶颈，线程数一旦多了，性能是反倒会差的。

IO 多路复用就是 1 个线程处理 多个 fd 的模式。我们的要求是：这个 “1” 就要尽可能的快，避免一切无效工作，要把所有的时间都用在处理句柄的 IO 上，不能有任何空转，sleep 的时间浪费。

有没有一种工具，我们把一箩筐的 fd 放到里面，只要有一个 fd 能够读写数据，后台 loop 线程就要立马唤醒，全部马力跑起来。其他时间要把 cpu 让出去。



这 3 种都能够管理 fd 的可读可写事件，在所有 fd 不可读不可写无所事事的时候，可以阻塞线程，切走 cpu 。fd 有情况的时候，都要线程能够要能被唤醒。

而这三种方式以 epoll 池的效率最高。为什么效率最高？

其实很简单，这里不详说，其实无非就是 epoll 做的无用功最少，select 和 poll 或多或少都要多余的拷贝，盲猜（遍历才知道）fd ，所以效率自然就低了。

举个例子，以 select 和 epoll 来对比举例，池子里管理了 1024 个句柄，loop 线程被唤醒的时候，select 都是蒙的，都不知道这 1024 个 fd 里谁 IO 准备好了。这种情况怎么办？只能遍历这 1024 个 fd ，一个个测试。假如只有一个句柄准备好了，那相当于做了 1 千多倍的无效功。

epoll 则不同，从 epoll\_wait 醒来的时候就能精确的拿到就绪的 fd 数组，不需要任何测试，拿到的就是要处理的。

epoll 的使用非常简单，只有下面 3 个系统调用。

epollcreate 负责创建一个池子，一个监控和管理句柄 fd 的池子；

epollctl 负责管理这个池子里的 fd 增、删、改；

epollwait 就是负责打盹的，让出 CPU 调度，但是只要有“事”，立马会从这里唤醒；

首先，epoll 的第一步是创建一个池子。这个使用 epoll\_create 来做：

int epoll\_create(int size);

我们拿到了一个 epollfd ，这个 epollfd 就能唯一代表这个 epoll 池。注意，这里又有一个细节：用户可以创建多个 epoll 池。

然后，我们就要往这个 epoll 池里放 fd 了，这就要用到 epoll\_ctl 了

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

第一个跟高效相关的问题来了，添加 fd 进池子也就算了，如果是修改、删除呢？怎么做到快速？

这里就涉及到你怎么管理 fd 的数据结构了。

最常见的思路：用 list ，可以吗？功能上可以，但是性能上拉垮。list 的结构来管理元素，时间复杂度都太高 O(n)，每次要一次次遍历链表才能找到位置。池子越大，性能会越慢。

那有简单高效的数据结构吗？

有，红黑树。Linux 内核对于 epoll 池的内部实现就是用红黑树的结构体来管理这些注册进程来的句柄 fd。红黑树是一种平衡二叉树，时间复杂度为 O(log n)，就算这个池子就算不断的增删改，也能保持非常稳定的查找性能。

现在思考第二个高效的秘密：怎么才能保证数据准备好之后，立马感知呢？

epoll\_ctl 这里会涉及到一点。秘密就是：回调的设置。在 epoll\_ctl 的内部实现中，除了把句柄结构用红黑树管理，另一个核心步骤就是设置 poll 回调。

思考来了：poll 回调是什么？怎么设置？

先说说 file\_operations->poll 是什么？

在 [文件描述符 fd 究竟是什么](http://mp.weixin.qq.com/s?__biz=Mzg3NTU3OTgxOA==&mid=2247489347&idx=1&sn=e897fd2f3584fe0fe0c011d4e6503274&chksm=cf3e0786f8498e903b463ac2ddaac2a0fb4cebac7c6cbf02ff02348fbc71dcd80d09a26c4257&scene=21" \l "wechat_redirect" \t "/private/var/folders/t2/whv3kyh93m763nbgz34swknc0000gn/T/com.kingsoft.wpsoffice.mac/wps-zpw/x/_blank)说过，Linux 设计成一切皆是文件的架构，这个不是说说而已，而是随处可见。实现一个文件系统的时候，就要实现这个文件调用，这个结构体用 struct file\_operations 来表示。这个结构体有非常多的函数，精简了一些，如下：

struct file\_operations {  
    ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);  
    ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);  
    \_\_poll\_t (\*poll) (struct file \*, struct poll\_table\_struct \*);  
    int (\*open) (struct inode \*, struct file \*);  
    int (\*fsync) (struct file \*, loff\_t, loff\_t, int datasync);  
    // ....  
};

你看到了 read，write，open，fsync，poll 等等，这些都是对文件的定制处理操作，对于文件的操作其实都是在这个框架内实现逻辑而已，比如 ext2 如果有对 read/write 做定制化，那么就会是 ext2\_read，ext2\_write，ext4 就会是 ext4\_read，ext4\_write。在 open 具体“文件”的时候会赋值对应文件系统的 file\_operations 给到 file 结构体。

那我们很容易知道 read 是文件系统定制 fd 读的行为调用，write 是文件系统定制 fd 写的行为调用，file\_operations->poll 呢？

这个是定制监听事件的机制实现。通过 poll 机制让上层能直接告诉底层，我这个 fd 一旦读写就绪了，请底层硬件（比如网卡）回调的时候自动把这个 fd 相关的结构体放到指定队列中，并且唤醒操作系统。

举个例子：网卡收发包其实走的异步流程，操作系统把数据丢到一个指定地点，网卡不断的从这个指定地点掏数据处理。请求响应通过中断回调来处理，中断一般拆分成两部分：硬中断和软中断。poll 函数就是把这个软中断回来的路上再加点料，只要读写事件触发的时候，就会立马通知到上层，采用这种事件通知的形式就能把浪费的时间窗就完全消失了。

划重点：这个 poll 事件回调机制则是 epoll 池高效最核心原理。

划重点：epoll 池管理的句柄只能是支持了 file\_operations->poll 的文件 fd。换句话说，如果一个“文件”所在的文件系统没有实现 poll 接口，那么就用不了 epoll 机制。

第二个问题：poll 怎么设置？

在 epoll\_ctl 下来的实现中，有一步是调用 vfs\_poll 这个里面就会有个判断，如果 fd 所在的文件系统的 file\_operations 实现了 poll ，那么就会直接调用，如果没有，那么就会报告响应的错误码。

static inline \_\_poll\_t vfs\_poll(struct file \*file, struct poll\_table\_struct \*pt)  
{  
    if (unlikely(!file->f\_op->poll))  
        return DEFAULT\_POLLMASK;  
    return file->f\_op->poll(file, pt);  
}

总结概括来说：挂了个钩子，设置了唤醒的回调路径。epoll 跟底层对接的回调函数是：ep\_poll\_callback，这个函数其实很简单，做两件事情：

把事件就绪的 fd 对应的结构体放到一个特定的队列（就绪队列，ready list）；

唤醒 epoll ，活来啦！

当 fd 满足可读可写的时候就会经过层层回调，最终调用到这个回调函数，把对应 fd 的结构体放入就绪队列中，从而把 epoll 从 epoll\_wait 出唤醒。

这个对应结构体是什么？

结构体叫做 epitem ，每个注册到 epoll 池的 fd 都会对应一个。

就绪队列需要用很高级的数据结构吗？

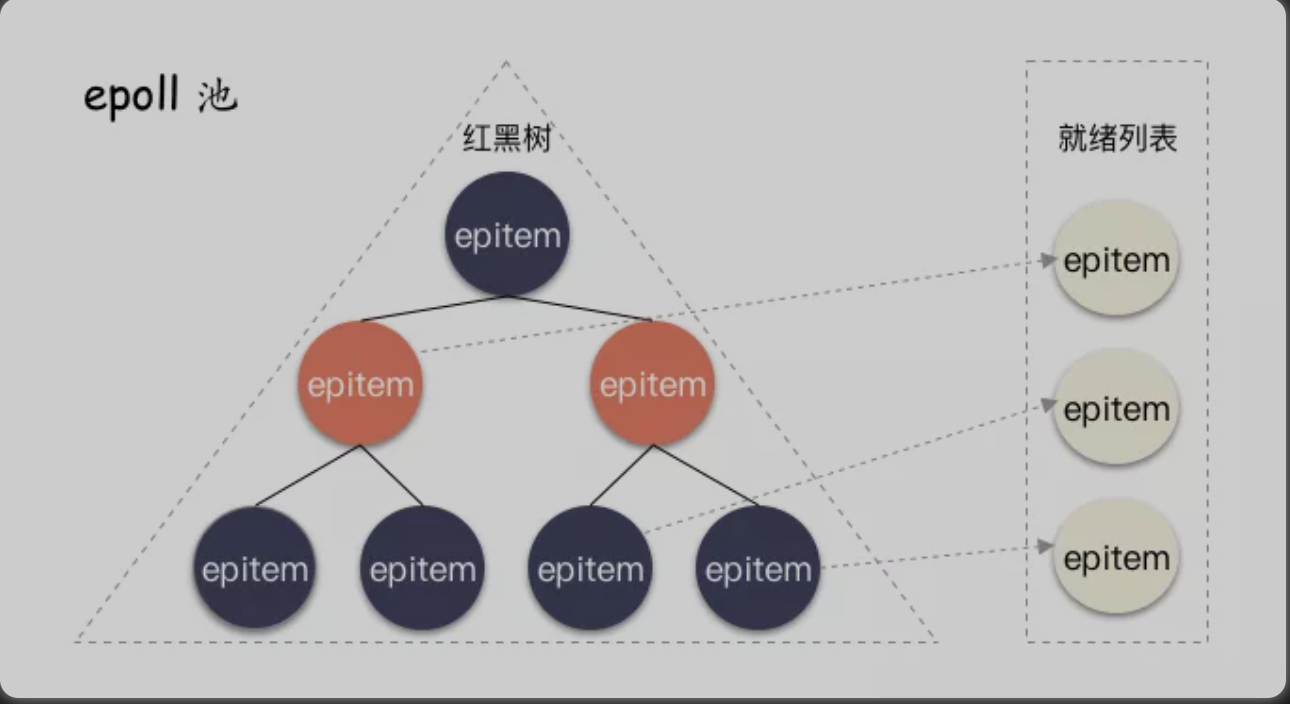
就绪队列就简单了，因为没有查找的需求了呀，只要是在就绪队列中的 epitem ，都是事件就绪的，必须处理的。所以就绪队列就是一个最简单的双指针链表。

小结下：epoll 之所以做到了高效，最关键的两点：

内部管理 fd 使用了高效的红黑树结构管理，做到了增删改之后性能的优化和平衡；

epoll 池添加 fd 的时候，调用 file\_operations->poll ，把这个 fd 就绪之后的回调路径安排好。通过事件通知的形式，做到最高效的运行；

epoll 池核心的两个数据结构：红黑树和就绪列表。红黑树是为了应对用户的增删改需求，就绪列表是 fd 事件就绪之后放置的特殊地点，epoll 池只需要遍历这个就绪链表，就能给用户返回所有已经就绪的 fd 数组；



再来思考另外一个问题：由于并不是所有的 fd 对应的文件系统都实现了 poll 接口，所以自然并不是所有的 fd 都可以放进 epoll 池，那么有哪些文件系统的 file\_operations 实现了 poll 接口？

首先说，类似 ext2，ext4，xfs 这种常规的文件系统是没有实现的，换句话说，这些你最常见的、真的是文件的文件系统反倒是用不了 epoll 机制的。

那谁支持呢？

最常见的就是网络套接字：socket 。网络也是 epoll 池最常见的应用地点。Linux 下万物皆文件，socket 实现了一套 socket\_file\_operations 的逻辑（ net/socket.c ）：

static const struct file\_operations socket\_file\_ops = {  
    .read\_iter =    sock\_read\_iter,  
    .write\_iter =   sock\_write\_iter,  
    .poll =     sock\_poll,  
    // ...  
};

我们看到 socket 实现了 poll 调用，所以 socket fd 是天然可以放到 epoll 池管理的。

还有支持的吗？

有的，很多。其实 Linux 下还有两个很典型的 fd ，常常也会放到 epoll 池里。

eventfd：eventfd 实现非常简单，故名思义就是专门用来做事件通知用的。使用系统调用 eventfd 创建，这种文件 fd 无法传输数据，只用来传输事件，常常用于生产消费者模式的事件实现；

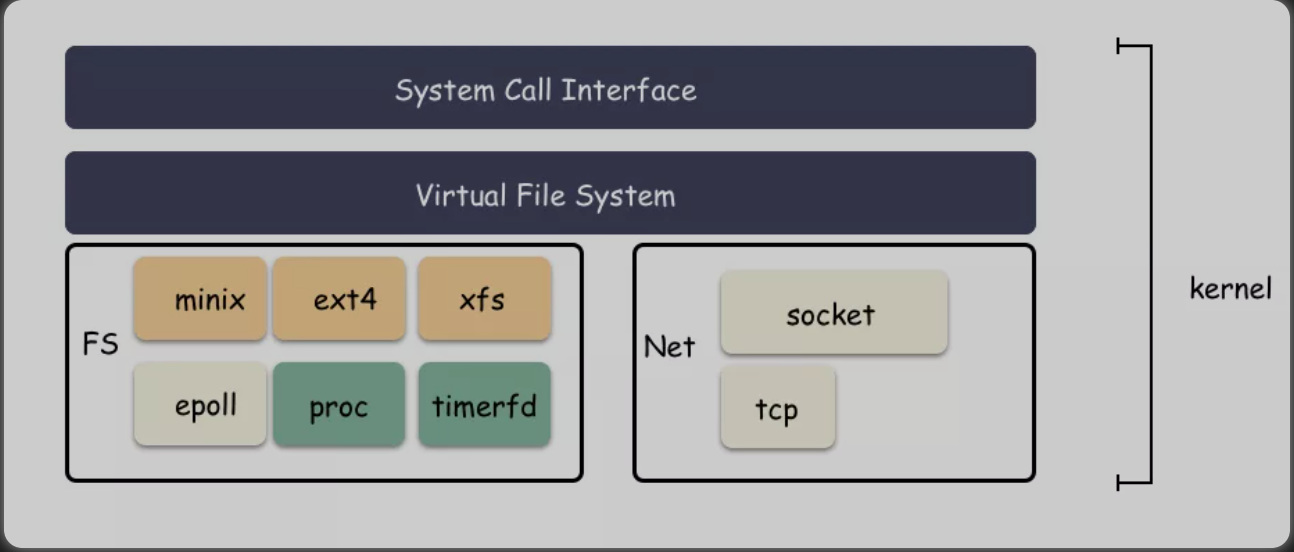
timerfd：这是一种定时器 fd，使用 timerfd\_create 创建，到时间点触发可读事件；

小结一下：

ext2，ext4，xfs 等这种真正的文件系统的 fd ，无法使用 epoll 管理；

socket fd，eventfd，timerfd 这些实现了 poll 调用的可以放到 epoll 池进行管理；

其实，在 Linux 的模块划分中，eventfd，timerfd，epoll 池都是文件系统的一种模块实现。



IO 多路复用的原始实现很简单，就是一个 1 对多的服务模式，一个 loop 对应处理多个 fd ；

IO 多路复用想要做到真正的高效，必须要内核机制提供。因为 IO 的处理和完成是在内核，如果内核不帮忙，用户态的程序根本无法精确的抓到处理时机；

fd 记得要设置成非阻塞的哦，切记；

epoll 池通过高效的内部管理结构，并且结合操作系统提供的 poll 事件注册机制，实现了高效的 fd 事件管理，为高并发的 IO 处理提供了前提条件；

epoll 全名 eventpoll，在 Linux 内核下以一个文件系统模块的形式实现，所以有人常说 epoll 其实本身就是文件系统也是对的；

socketfd，eventfd，timerfd 这三种”文件“fd 实现了 poll 接口，所以网络 fd，事件fd，定时器fd 都可以使用 epoll\_ctl 注册到池子里。我们最常见的就是网络fd的多路复用；

ext2，ext4，xfs 这种真正意义的文件系统反倒没有提供 poll 接口实现，所以不能用 epoll 池来管理其句柄。那文件就无法使用 epoll 机制了吗？不是的，有一个库叫做 libaio ，通过这个库我们可以间接的让文件使用 epoll 通知事件，以后详说，此处不表；