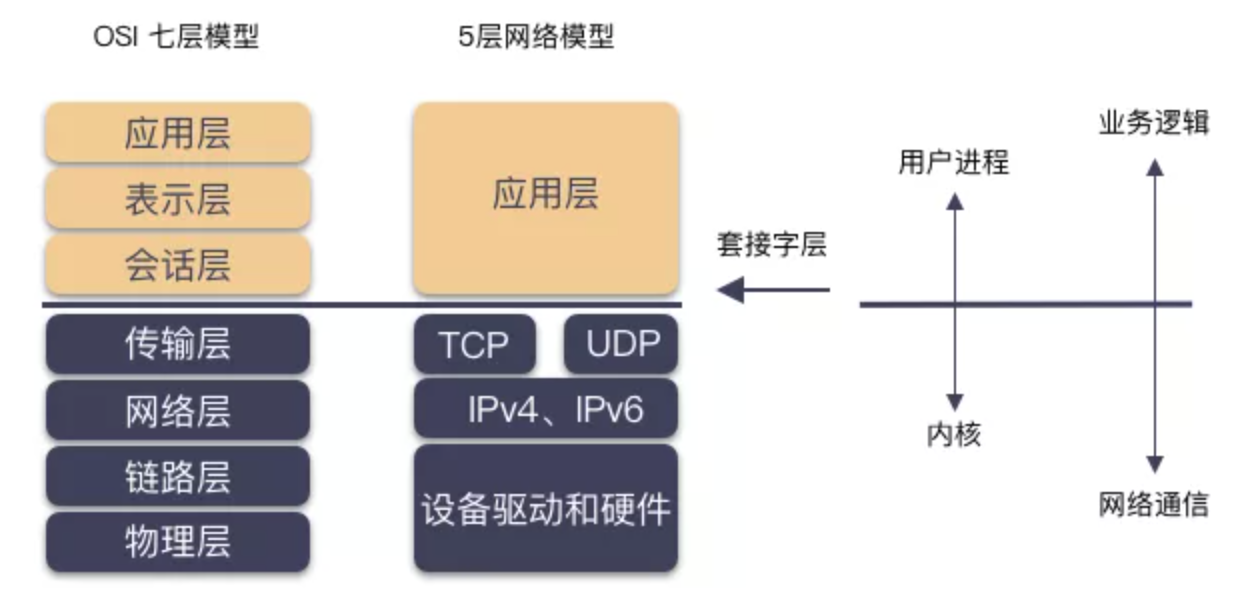
什么是 socket fd ？粗糙的来讲，就是网络 fd，比如我们最常见的 C/S 客户端服务端的编程模式，就是网络通信的一种方式。撇开底层和协议细节，**网络通信**和**文件读写**从接口上有本质区别吗？

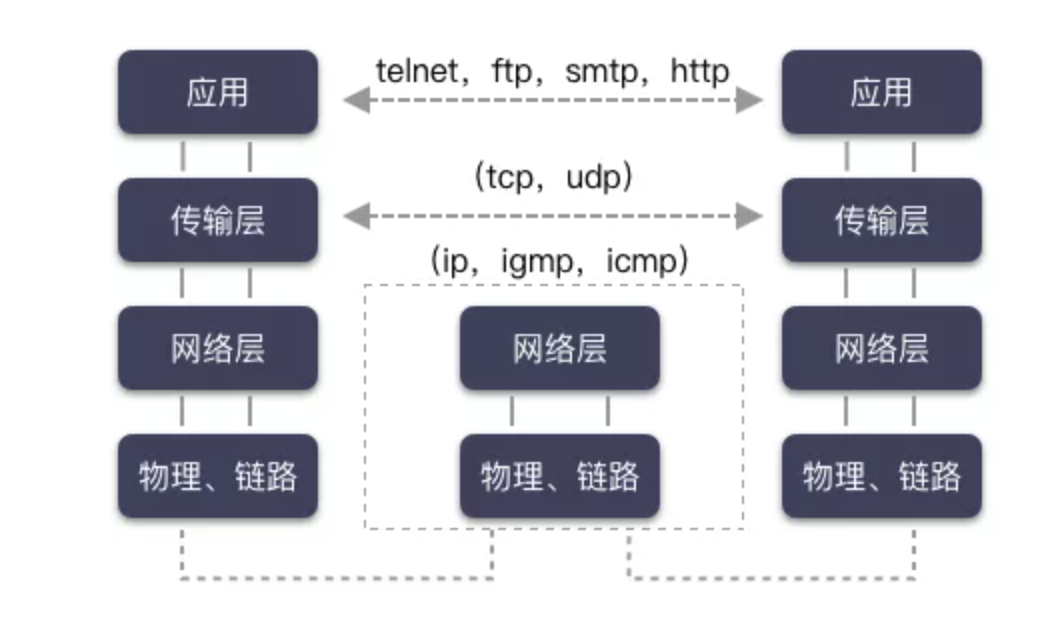
其实没啥区别，不就是读过来和写过去嘛，简称 IO 。

网络模型一般会对应到两种：

* 完美理论的 **OSI 七层模型**；
* 现实应用的 5 层模型；

对应关系如下图（取自 Unix 套接字编程）





套接字，是内核对贼复杂的网络协议栈的 API 封装，使得程序猿能够用极简的姿势进行网络编程。比如写一个基于 Tcp 的 C/S 的网络程序，需要用到啥？我们大概畅想下：

1. 客户端和服务端都用 socket 调用创建套接字；
2. 服务端用 bind 绑定监听地址，用 listen 把套接字转化为监听套接字，用 accept 捞取一个客户端来的连接；
3. 客户端用 connect 进行建连，用 write/read 进行网络 IO；

套接字由 socket() 系统调用创建，但你可知套接字其实可分为两种类型，监听套接字和普通套接字。而监听套接字是由 listen() 把 socket fd 转化而成。

**监听套接字**

对于监听套接字，不走数据流，只管理连接的建立。accept 将从全连接队列获取一个创建好的 socket（ 3 次握手完成），对于监听套接字的可读事件就是全连接队列非空。对于监听套接字，我们只在乎可读事件。

**普通套接字**

普通套接字就是走数据流的，也就是网络 IO，针对普通套接字我们关注可读可写事件。在说 socket 的可读可写事件之前，我们先捋顺套接字的读写大概是什么样子吧。

套接字层是内核提供给程序员用来网络编程的，程序猿读写都是针对套接字而言，那么 write( socketfd, /\* 参数 \*/) 和 read( socketfd, /\* 参数 \*/) 都会发生什么呢？

* write 数据到 socketfd，大部分情况下，数据写到 socket 的内存 buffer，就结束了，并没有发送到对端网络（异步发送）；
* read socketfd 的数据，也只是从 socket 的 内存 buffer 里读数据而已，而不是从网卡读（虽然数据是从网卡一层层递上来的）；

也就是说，程序猿而言，是跟 socket 打交道，内核屏蔽了底层的细节。

那说回来 socket 的可读可写事件就很容易理解了。

* socketfd 可读：其实就是 socket buffer 内有数据（超过阈值 SO\_RCLOWAT ）；
* socketfd 可写：就是 socket buffer 还有空间让你写（阈值 SO\_SNDLOWAT ）；

socket fd 为什么能具备“文件”的语义，从而和 eventfd，ext2 fd 这样的句柄一样，统一提供对外 io 的样子？

核心就是：sockfs ，这也是个文件系统，只不过普通用户看不见，这是只由内核管理的文件系统，位于 vfs 之下，为了封装 socket 对上的文件语义。

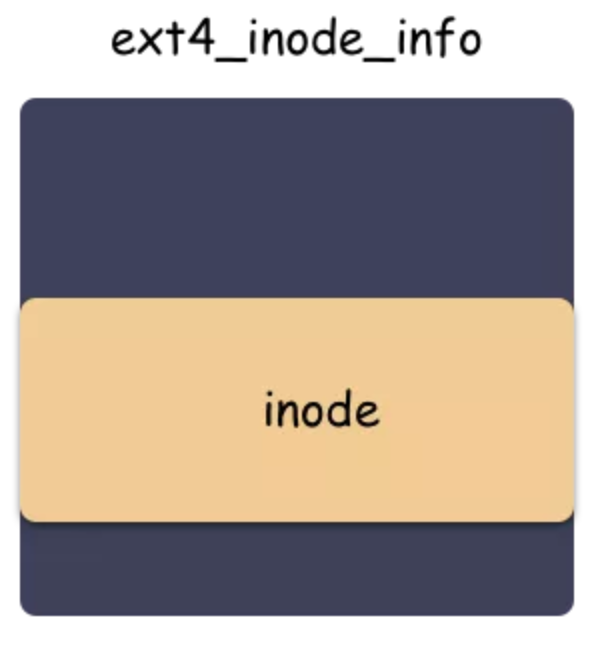
// net/socket.c  
static int \_\_init sock\_init(void)  
{  
    // 注册 sockfs 文件系统  
    err = register\_filesystem(&sock\_fs\_type);  
    // 内核挂载  
    sock\_mnt = kern\_mount(&sock\_fs\_type);  
}

其中最关键的是 sock\_mnt 这个全局变量里面的超级块的操作表 sockfs\_ops 。

// net/socket.c  
static const struct super\_operations sockfs\_ops = {  
    .alloc\_inode    = sock\_alloc\_inode,  
    .destroy\_inode  = sock\_destroy\_inode,  
    .statfs     = simple\_statfs,  
};

这个是每个文件系统的核心函数表，如上指明了 inode 的分配规则（这里又将体现依次结构体内嵌组合+类型强转的应用）。

读者朋友还记得 inode 和 ext4\_inode\_info 的关系吗？在 [Linux 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 "/Users/zpw/Documents\\x/_blank)一文中有提到这个：



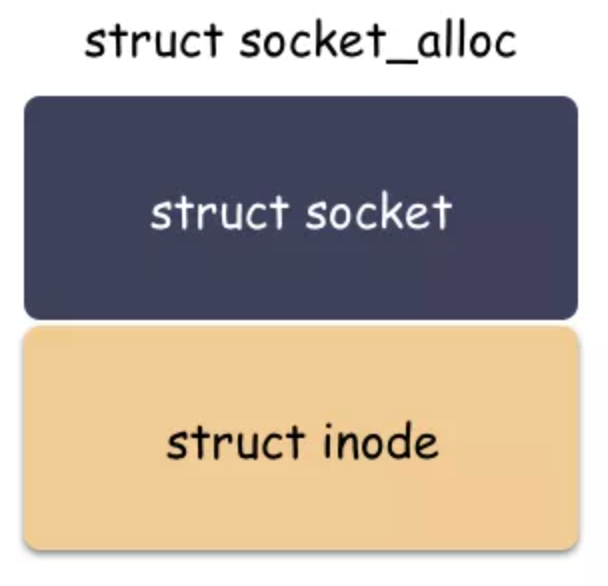
inode 是 vfs 抽象的适配所有文件系统的结构体，但分配其实是有下层具体文件系统分配出来的，以 ext4 文件系统来说，使用 ext4\_alloc\_inode 函数分配出 ext4\_inode\_info 这个大结构体，然后返回的是 inode 的地址而已。

**划重点：**struct inode**内嵌于具体文件系统的 “inode” 里，vfs 层使用的是 inode，ext4 层使用的是**ext4\_inode\_info**，不同层次通过地址的强制转化类型来切换结构体。**

那么类似，sockfs 也是如此，sockfs 作为文件系统，也有自己特色的 “inode”，这个类型就是 struct socket\_alloc ，如下：

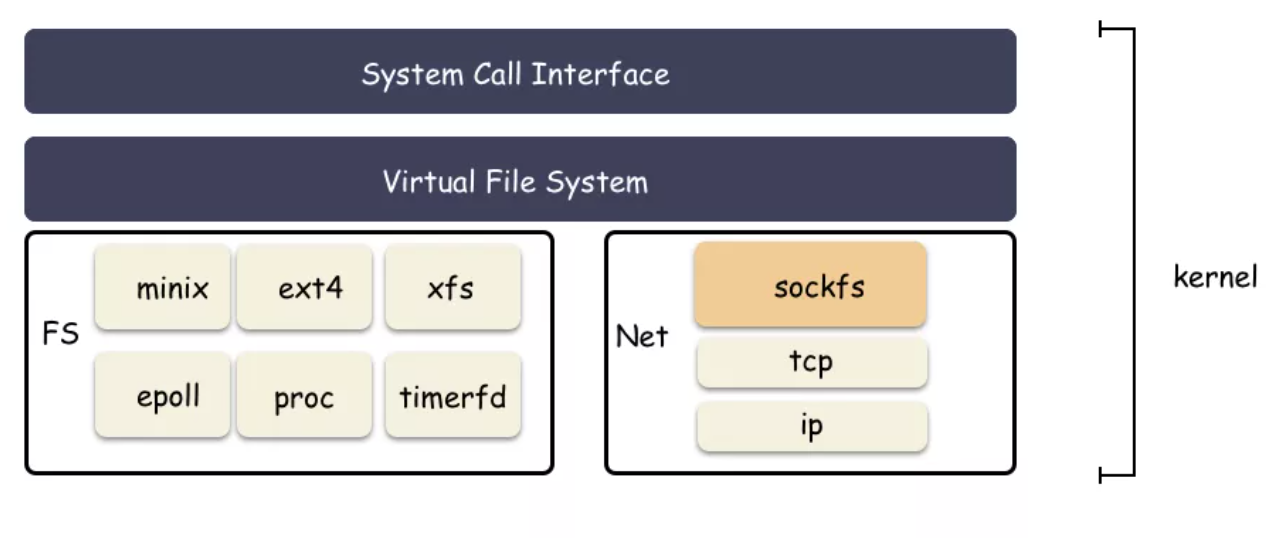
struct socket\_alloc {  
    struct socket socket;  
    struct inode vfs\_inode;  
};

这个结构体关联 socket 和 inode 两个角色，是“文件”抽象的核心之一。分配 struct socket 结构体其实是分配了 struct socket\_alloc 结构体，然后返回了 socket\_alloc->socket  字段的地址而已。



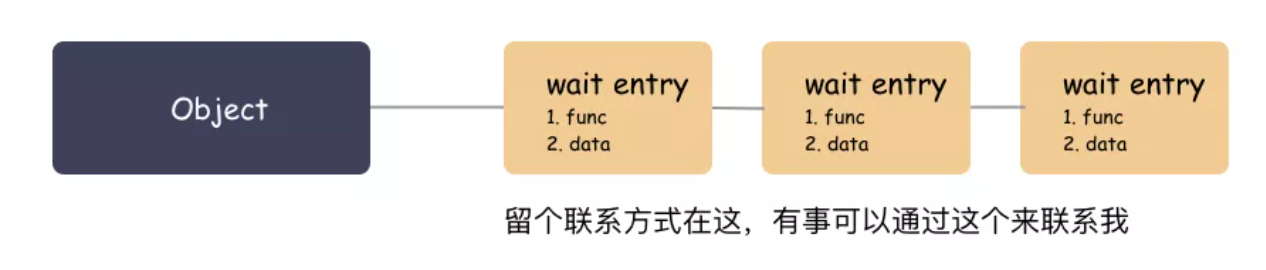
**划重点：vfs 层用的时候给 inode 字段的地址，socket 层的时候给 socket 字段的地址。不同抽象层面对于同一个内存块的理解不同，强制转化类型，然后各自使用**

从文件的角度来看 socket，模块如下：



**先铺垫一个小知识点：内核里面有回调唤醒的实现，里面有用到一种 wait queue 的做法，其实很简单的原理。**

大白话原理：你要走可以，把联系方式留下，我搞好之后通知你（调用你留下的函数，传入你留下的参数）。



拿 socket 来说，struct sock 里面就有个字段 sk\_wq ，这是个表头，就是用来挂接等待对象的。

就以 epoll 池来说，epoll\_ctl 注册 socket fd 的时候，就会挂一个 wait 对象到 sk->sk\_wq 里。回调参数为 ep\_poll\_callback ，参数为 epitem 。

这样 epoll 给 socket 留下联系方式了（ wait 对象 ），socket 有啥事就可以随时通知到 epoll 池了。

能有什么事？

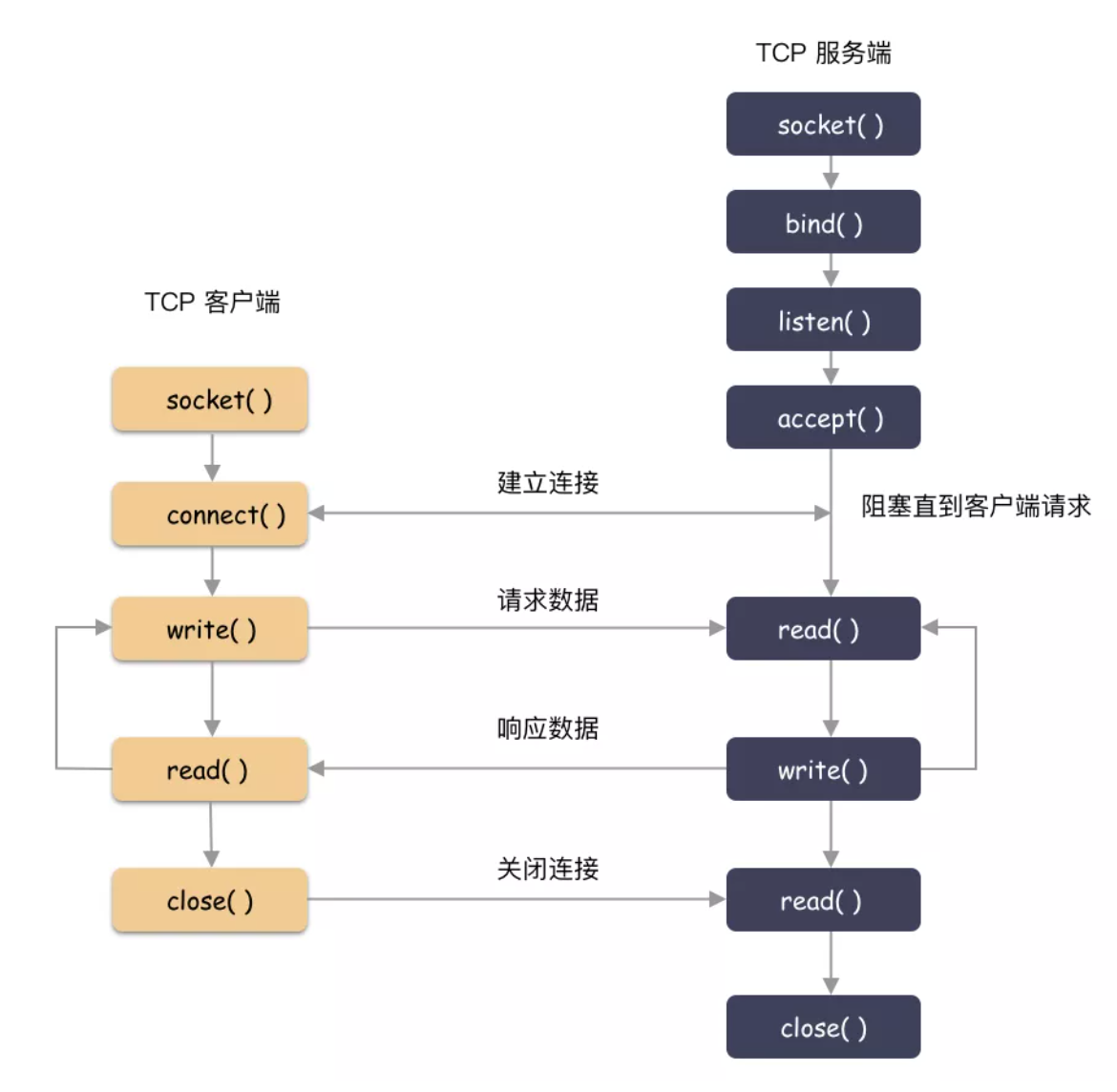
socket 可读可写了呗。sk buffer 里面有数据可以读，或者有空间可以写了呗。对于监听类型的 socket，有新的连接了呗。epoll 监听的不就是这个嘛。

**服务端：**

1. socket( ) 创建出 socketfd；
2. bind( )  绑定一个端口（和客户端约定好的知名端口号）；
3. listen( ) 讲套接字转化成监听套接字；
4. accept( ) 等待客户端的建连请求；
5. 建连之后 read/write 处理数据即可（一般和监听线程并发）；

**客户端：**

1. socket( ) 创建出 socketfd；
2. connect( ) 向指定机器、端口发起建连请求；
3. 建连之后，read/write 处理数据；



**socket 函数**

#include<sys/socket.h>  
int socket(int family, int type, int protocol)

socket 系统调用对应了 \_\_sys\_socket 这个函数。这个函数主要做两件事情：

1. 第一件事：调用 socket\_create 函数创建好 socket 相关的结构体，主要是 struct socket ，还有与之关联的 socket sock 结构，再往下就是具体网络协议对应的结构体；
2. 第二件事：调用 sock\_map\_fd 函数创建好 struct file 这个结构体，并与第一步创建出的 struct socket 关联起来；

**涉及的一些函数调用**：

\_\_sys\_socket  
    // 创建 struct socket 结构体  
    -> sock\_create  
            // 创建 struct socket 结构，并且关联特殊 inode  
            -> sock\_alloc  
            // pf 是根据 family 从 net\_families 这个全局表中取出的操作函数表，用来创建具体网络协议结构的;    
            // 比如 IPv4 对应的 family 就是 AF\_INET ，对应的函数是 inet\_create  
            // 在这里面会赋值 sock->ops 为对应协议族的操作函数表（比如 inet\_stream\_ops）  
            -> pf->create  
                    // struct sock 结构体的创建（sk->sk\_prot 的赋值就在这里，比如 tcp\_prot ）  
                    -> sk\_alloc  
                    // struct sock 结构体的初始化(比如 sk\_receive\_queue, sk\_write\_queue, sk\_error\_queue 就是在这里初始化的)  
                    // 可读写的关键函数 sock\_def\_readable，sock\_def\_write\_space 也是在这里赋值的  
                    -> sock\_init\_data  
    // 创建 struct file 结构体，并且关联 struct socket  
    -> sock\_map\_fd

**先说 socket 函数：**：

1. socket( ) 函数只负责创建出适配具体网络协议的资源（内存、结构体、队列等），并**没有和具体地址绑定**；
2. socket( ) 返回的是非负整数的 fd，与 struct file 对应，而 struct file 则与具体的 struct socket 关联，从而实现**一切皆文件**的封装的一部分（另一部分 inode 的创建处理在 sock\_alloc 的函数里体现）；

**再简要说下内部细节：**

sock\_create 函数里，会根据协议族查找对应的操作表，以 AF\_INET 协议族举例，pf->create 是 inet\_create ，主要做两件事：

1. 把 sock->ops 按照协议类型赋值成**具体的函数操作表**，比如 tcp 的就是 inet\_stream\_ops ；
2. 创建了 struct sock 对象，并且把 struct sock 初始化，并和 struct socket 进行关联；

着重提一点，sock\_init\_data  函数（ net/core/sock.c ）主要是初始化 struct sock 结构体的，提两点最关键的：

**第一点：接收队列和发送队列在这里初始化**；

* sk\_receive\_queue：套接字接收到的数据（sk\_buff 里面是纯粹的用户数据哦，没有 header 啥信息）；
* sk\_write\_queue：套接字要发送的数据；
* sk\_error\_queue：挂接一些 pengding 的 error 信息；

**第二点：socket 的唤醒回调在这个地方设置**；

sk->sk\_data\_ready   =   sock\_def\_readable;  
sk->sk\_write\_space  =   sock\_def\_write\_space;

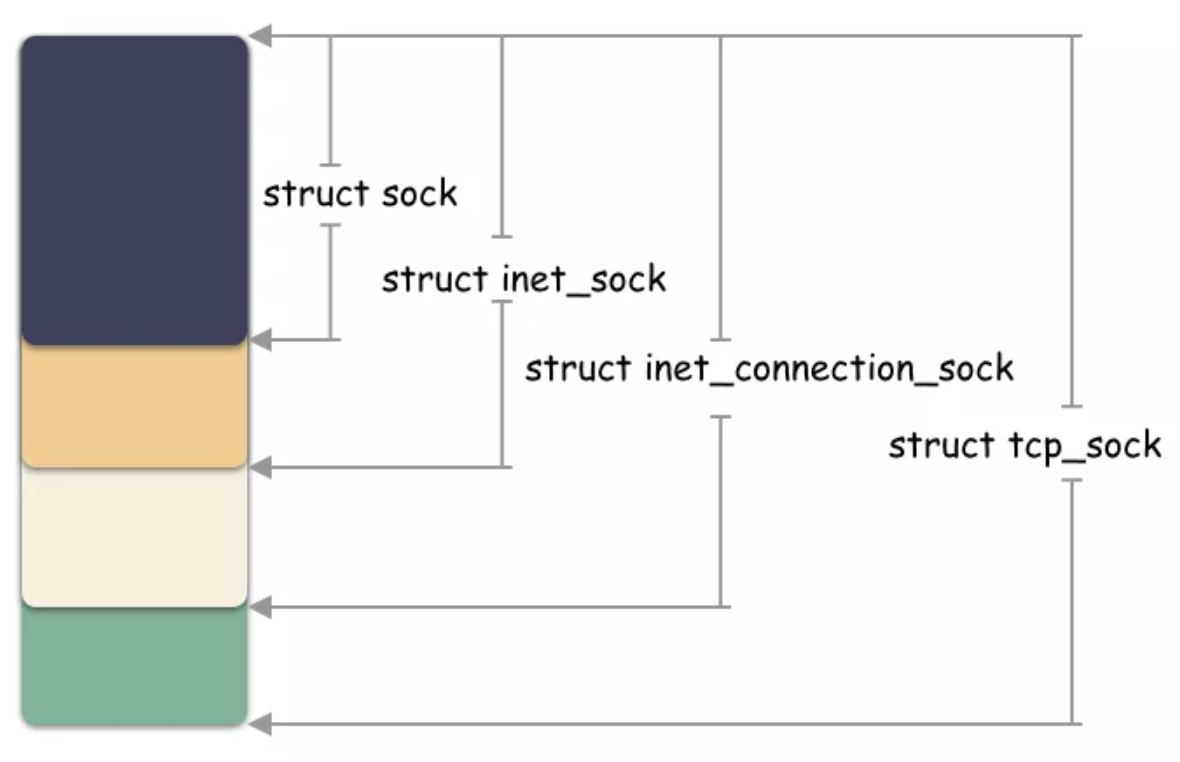
为什么这里很重要，因为这个跟 socket fd 可读可写的判断逻辑，数据到了之后的唤醒路径息息相关。简述下回调链路（以套接字层为主干，其他的流程简略描述）：

sk->sk\_data\_ready（数据到了，该通知留下过联系方式的人了）  
tcp\_v4\_rcv（具体协议栈处理函数）  
软中断  
硬中断

**再说下结构体：**

继续说 struct sock ，这个对象有意思了，这个也是以组合的方式往下兼容的，同一个地址强转类型得到不同层面的结构体。原理就在于：他们是一块连续的内存空间，起始地址相同。

sock -> inet\_sock -> inet\_connection\_sock-> tcp\_sock



**小思考：**struct socket**和**struct sock**是两个不同的结构体？**

是的。这两个是不同的结构体。属于**套接字层**的两个维度的描述，一个面向上层，一个面向下层。

struct socket 在内核的注释为：

struct socket - general BSD socket

struct sock 在内核的注释为：

struct sock\_common - minimal network layer representation of sockets

struct socket 是内核抽象出的一个通用结构体，主要作用是放置了一些跟 fs 相关的字段，而真正跟网络通信相关的字段结构体是 struct sock 。它们内部有相互的指针，可以获取到对方的地址。

struct socket 这个字段出生的时候其实就和一个 inode 结构体伴生出来的，由 socketfs  的 sock\_alloc\_inode  函数分配。

struct sock 这个结构体是 socket 套阶字核心的结构（注意，还有个结构是 struct socket，这两个是不同的结构体哦）。这个是对底下具体协议做的一层抽象封装，比如在分配 struct sock 的时候，如果是 tcp 协议，那么 sk->sk\_prot 会赋值为 tcp\_prot ，udp 协议赋值的是 udp\_prot ，之后的一系列协议解析和处理就是调用到对应协议的回调函数。

**小思考：socket fd 可以和文件一样用**write(fd, /\*xxxx\*/ )**这行的调用，为什么？**

write(fd, /\*xxxx\*/) 进到内核首先是到 vfs 层，也就是调用到 vfs\_write ，在这个里面首先获取到 file 这个结构体，然后调用下层注册的回调，比如 file->f\_op->write\_iter ，file->f\_op->write ，所以，关键在 file->f\_op 这个字段，对吧？

**现在的问题是，这个字段是啥呢？**

这个字段在 file 结构体生成的时候，根据你的“文件”类型赋值的，这个在之前文件系统章节提过这个，比如 ext2 的文件，那么就是 ext2\_file\_operations ，socketfd 是 socket\_file\_ops。

vfs\_write    =>    
                -> socket\_file\_ops （sockfs）  
                -> ext2\_file\_operations （ext2）  
                -> ext4\_file\_operations （ext4）  
                -> eventfd\_fops

可以看下 socket\_file\_ops  的定义：

static const struct file\_operations socket\_file\_ops = {  
    .llseek =   no\_llseek,  
    .read\_iter =    sock\_read\_iter,  
    .write\_iter =   sock\_write\_iter,  
    .poll =     sock\_poll,  
    // ...  
}

所以，vfs\_write 调用到的将是 sock\_write\_iter，而这个里面就是调用到 sock\_sendmsg ，从而走到网络相关的处理流程。

// sock\_sendmsg 实际调用；  
static inline int sock\_sendmsg\_nosec(struct socket \*sock, struct msghdr \*msg)  
{  
    int ret = sock->ops->sendmsg(sock, msg, msg\_data\_left(msg));  
    return ret;  
}

还记得上面在 socket 初始化的时候 socket->ops 和 sock->sk\_prot 两个回调函数操作表的赋值吗（ tcp ）：

* socket->ops => inet\_stream\_ops
* sock->sk\_prot => tcp\_prot

这样从 vfs 进来，转接到具体的协议处理模块去了。

**bind 函数**

对应内核 \_\_sys\_bind 函数，做的事情很简单：

1. 先通过 fd 找到对应的 struct socket 结构体；
2. 然后把 address 和 socket 绑定对应起来（调用 sock->ops->bind 函数）；

tcp 连接的对应的 bind 函数是 inet\_bind，里面做的事情很简单，就是简单的查一下端口有没有被占用，没有被占用的话端口就赋值给 inet\_sock->inet\_sport 这个字段。

inet\_sock 则是由 sk 强转类型得到。

思考个小问题：在上面的图中，bind 这个函数只在服务端用到？

**为啥客户端没用这个函数呢？**

其实，客户端也是可以用 bind 这个函数，**但是没必要**。

理解下 bind 函数的作用：给这个 socketfd 绑定地址（IP:Port）用的。客户端不需要是因为：如果没设置，内核在建连的时候会自动选一个临时的端口号作为本次 TCP 连接的地址。一般客户端也不在意端口号，只要能和服务端正常通信就好，所以客户端一般没有 bind 调用。

服务端必须要用这个是因为服务端必须提前明确指定监听的 IP 和 Port （不然谁知道向哪里发起连接呢）。

**listen 函数**

其实 socket( ) 创建出来的套接字并无客户端和服务端之分，是 listen 函数让 socket 有了不一样的属性，成为监听套接字。

listen 系统调用主要做两件事：

1. 通过 fd 找到 struct socket 结构体；
2. 调用 sock->ops->listen 函数（对应 inet\_listen ）；

inet\_listen  做啥了？内核注释：

Move a socket into listening state.

简单看下 inet\_listen 的实现功能：

1. 检查 socket 状态，类型，必须为流式套接字才能转化成监听套接字；
2. 调用 inet\_csk\_listen\_start ；

inet\_csk\_listen\_start 做啥了？

1. 初始化请求队列 icsk->icsk\_accept\_queue ；
2. 套接字状态设置成 TCP\_LISTEN；
3. 获取到之前 bind 的端口，如果没有设置，那么就会用个临时的端口；
4. 把监听套接字加入到全局 hash 表中；

**划重点：套接字的转变就在于此。**

**accept 函数**

inet\_accept （ net/ipv4/af\_inet.c ）注释：

Accept a pending connection. The TCP layer now gives BSD semantics.

这个主要是从队列 icsk->icsk\_accept\_queue 中取请求，如果队列为空，就看 socket 是否设置了非阻塞标识，非阻塞的就直接报错 EAGAIN，否则阻塞线程等待。

所以，监听套接字的可读事件是啥？

icsk\_accept\_queue 队列非空。

这个队列什么时候被填充的？

tcp\_child\_process  
    -> tcp\_rcv\_state\_process

这个也是底层网络协议回调往上调用的，tcp 三次握手之后，建立好的连接就在一个队列中 accept\_queue ，队列非空则为只读。由 tcp 的协议栈往上调用，对应到 socket 层，还是会调用到 sk->sk\_data\_ready 。

这里还是以 epoll 管理监听套接字来举例。这个跟上面讲的数据来了一样，都是把挂接在 socket 本身上的 wait 对象进行唤醒（调用回调），这样就会到 ep\_poll\_callback ，ep\_poll\_callback 就会把监听套接字对应的 ep\_item 挂到 epoll 的 ready 队列中，并且唤醒阻塞在 epoll\_wait 的线程，从而实现了监听套接字的读事件的触发的流程。

**connect 函数**

这个没啥讲的，就是由客户端向服务端发起连接的时候调用，一般也和 epoll 配合不起来，略过。

**句柄事件**

epoll 池可以管理 socket fd ，用于监听 socket fd 的可读，可写事件。那么问题来了，socket fd 的可读可写事件分别是啥？代表了什么含义？

这个要把服务端的监听类型的 socket fd 和传输数据的 socket fd 分开来说。

**监听类型的 fd：**

1. 有 client 建连，则触发可读事件；
2. 句柄被 close ，则触发可读事件；

**数据类型的 fd：**

1. sk buffer 有可读的数据，触发可读事件；
2. sk buffer 有可写的空间，触发可写事件；
3. 句柄杯 close，连接关闭的时候，也是可读的；

还有，如果 socket 之上有 pending 的 error 待处理，那么也会触发可读事件。

**epoll 池怎么配合？**

socket fd 数据就绪之后，怎么能及时的唤醒被阻塞在 epoll\_wait 的线程？

还记得套接字 buffer 数据来了的时候的回调吗？

调用的是 sk->sk\_data\_ready 这个函数指针，这个字段在 socket 初始化的时候被赋值为  sock\_def\_readable ，这个函数里面会**依次调用**所有挂接到 socket 的 wait 队列的对象（ 表头：sk->sk\_wq ），在这个 wait 队列中存在和 epoll 关联的秘密。

回忆下，在  深入剖析 epoll 篇 提到，epoll\_ctl 的时候，在把 socket fd 注册进 epoll 池的时候，会把一个 wait 对象挂接到这个 socket 的 sk->sk\_wq 中 ，回调函数就是 ep\_poll\_callback 。

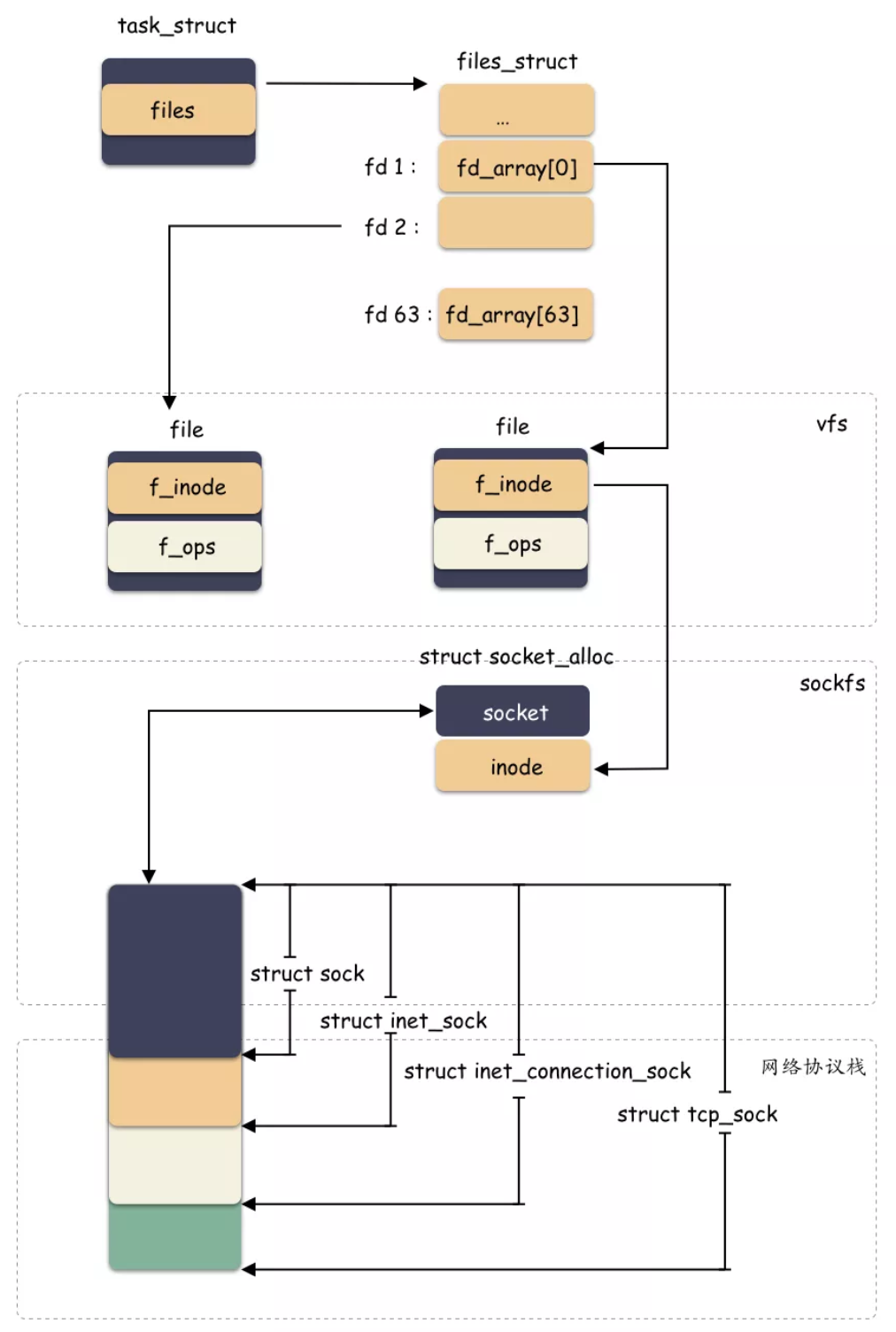
这个wait 对象就是数据就绪时候的联系方式，这样把 socket 数据就绪的流程和 epoll 关联上了。

也就是说，sk->sk\_data\_ready 会调用到 ep\_poll\_callback ，ep\_poll\_callback 这个函数处理很简单，做两件事情：

1. 把 socket 对应的 ep\_item 挂接到就绪队列中；
2. 把阻塞在 epoll\_wait 的线程（Linux 进程和线程本质无区别）投递到就绪队列中，等待内核调度（也就是所谓的唤醒，实现机制很简单，就是 epoll\_wait 阻塞切走之前，会创建出一个 wait 对象，挂到 epoll 池上，后续唤醒就能以此为依据）；

ep\_poll\_callback  
 sk->sk\_data\_ready  
 tcp\_v4\_rcv（具体协议栈处理函数）  
 软中断  
 硬中断  
  
 数据来了

## ****最后用一张简要的图展示结构体之间的关系：****



**总结**

1. vfs 下有一个**sockfs 的抽象层，是把 socket 抽象成“文件” fd 的关键之一**；
2. socket fd 能够和文件 IO 一样，使用 write/read 等系统调用，就得益于 vfs 帮你做的转接。那 socket() 函数调用是不是就和 open 文件 fd 的效果是一样的呀？是的，都是构建并关联各种内核结构体；
3. epoll 池能管理 socketfd，因为 socket fd 实现 poll 接口；
4. epoll\_ctl 注册 socket fd 的时候，挂了个 wait 对象在 socket 的 sk\_wq 里，所以数据就绪的时候，socket 才能通知到 epoll；
5. epoll\_wait 切走的时候挂了个 wait 对象在 epoll 上，所以 epoll 就绪的时候，才能有机会唤醒阻塞的线程；
6. **套接字由**socket()**创建出来**，客户端和服务端都是，listen() 调用可以把套接字转化成监听套接字；
7. 监听套接字一般只监听可读事件，关注连接的建立，普通套接字走数据流，关注数据的读写事件；