## 网络结构总结

epoll 水平触发+ oneshot ≈ epoll 边缘触发

水平触发+ oneshot 处理接收采用多线程

epoll 边缘触发 采用单线程

## 获取接收数据的就绪数据字节数

使用 ioctl 函数：

int nBytesAvailable = 0;

ioctl(sockfd, FIONREAD, &nBytesAvailable);

比如在 Linux 系统中，获取套接字缓冲区可读取数据大小的 ioctl 操作类型为 FIONREAD，定义在 sys/ioctl.h 中。

在阻塞模式下，调用 ioctl(sockfd, FIONREAD, &nread) 将会一直等待直到有数据可读，并返回可读取的数据字节数。如果在超时时间内无法读取到数据，则会返回相应的错误码和标识符。可以配合 epoll 在数据就绪时调用，获得就绪的字节数

## C++ recv 怎么实现读取数据但是不从缓冲区移除

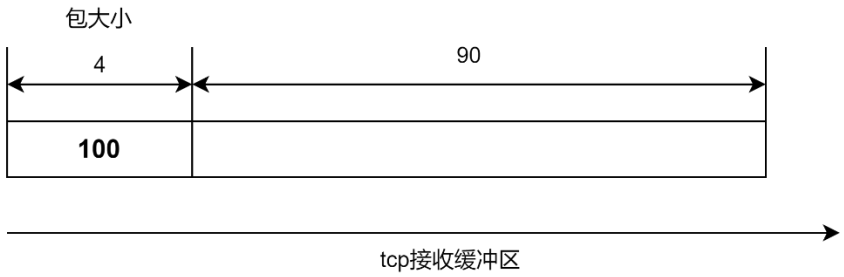
MSG_PEEK 标志表示仅从缓冲区中读取数据，而不将其删除。在 recv 函数中使用 MSG_PEEK 标志后，可以读取指定长度但不移除缓冲区中的数据，下次读取该套接字时，仍然可以继续读取到这个数据段。代码如下

```
char buf[1024] = {0};

int n = recv(sockfd, buf, sizeof(buf), MSG_PEEK);

// 这里 buf 中的数据仍然会保留在缓冲区中
```

为什么用它？比如现在是先发包大小再发数据包，那么首先应该先读 4 个字节，如果通过 ioctl 判断出有 4 个字节，可以处理数据了，比如读出包大小是 100 字节，有可能就绪的字节数不足 100 个，




那么这 4 个字节应该保存在接收缓冲区里面不读出来，然后退出，等待下次就绪数据有包大小那么多数据再读，然后处理，此种处理办法，适合阻塞或非阻塞接收不加缓冲区的方式.

## 阻塞套接字的非阻塞接收

```
char buffer[MAX_SIZE];
int nread = recv(sockfd, buffer, MAX_SIZE, MSG_DONTWAIT);
```

MSG_DONTWAIT 套接字临时按照非阻塞进行接收 ，没有可读数据会立即返回.

## 阻塞套接字接收超时

```
struct timeval timeout = {3, 0}; // 设置超时时间为 3 秒
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout, sizeof(timeout));
char buf[1024] = {0};
int n = recv(sockfd, buf, sizeof(buf), 0);
if(n < 0 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
    // 超时或者没有足够的数据可读
} else {
    // 成功读取到数据
}
```

此接收办法，当接收数据不足 sizeof(buf)时，达到了超时时间，会退出函数，返回接收数据的长度，或错误. 可以避免恶意攻击发送一个字节 ，占用服务器资源. 不过建议配合接收缓冲区使用，不然数据

就丢失了

## 服务器的优化处理

### 方案 1(不使用用户缓冲区)

采用 epoll 水平触发 + oneshot 多线程接收，非阻塞接收(临时)，阻塞发送(套接字设置为阻塞)

流程

epollwait 返回，判断为 EPOLLIN 事件，处理时，

步骤 1:

判断就绪数据是否够 4 个字节，如果够，peek 方式读出包大小，在查看就绪数据是否是包大小+4

步骤 2:

如果不是返回(不够一个包)，如果是，读出包大小，然后读出数据包，抛线程池执行函数，循环上面的步骤 1

发送是阻塞发送

优缺点：接收由于没有接收缓冲区，采用查看就绪字节数确认有足够包，采取读取的方式，会降低读写效率，不如有接收缓冲区，直接都读取出来．发送是阻塞的，好处在于设计简单，缺点是效率低，毕竟阻塞要先就绪然后运行，响应会慢一些，不如非阻塞高效．

### 方案 2(加接收缓冲区)

采用 epoll 水平触发 + oneshot 多线程接收，非阻塞接收(临时)，阻塞发送(套接字设置为阻塞)

定义接收缓冲区，结构体 RecvBuff，成员 char *buff , int pos , int len

流程

epollwait 返回，判断为 EPOLLIN 事件，处理时，

步骤 1:

首先判断 RecvBuff 是否申请，没有申请 ，判断就绪数据是否够 4 个字节，如果够，读取出包大小，然

后根据包大小给 RecvBuff 申请堆空间, pos = 0 , len = 包大小, 然后按照接收向 RecvBuff->buff+pos 位置接收数据, recv ( len - pos ) 的长度.

步骤 2:

读够一个包, 抛线程池执行数据包, 然后循环步骤 1 , 直到 EAGIAN 或 EWOULDBLOCK

发送是阻塞的.

## 方案 3: (et + 非阻塞接收 + 用户缓冲区)

采用 epoll 边沿触发单线程接收, 非阻塞接收(临时) , 阻塞发送(套接字设置为阻塞)

流程与方案 2 基本相同.

## 方案 4: (et 单线程接收+ 非阻塞接收 + 用户缓冲区 + 非阻塞发送)

接收与上面流程相同, 发送采取发送缓冲区, 发送时将数据都填充到队列中, 然后注册 EPOLLOUT 事件, 事件到来时, 取出队列中数据, 发送直到 EAGAIN 或 EWOULDBLOCK , 如果都发送完成, 取消 EPOLLOUT 注册 .

考虑读写队列涉及多线程操作, 那么, 需要线程同步来解决, 解决办法, 写一个线程安全的队列, 底层可以使用 deque 实现. 队列中添加的是一个一个发送数据包