io_uring

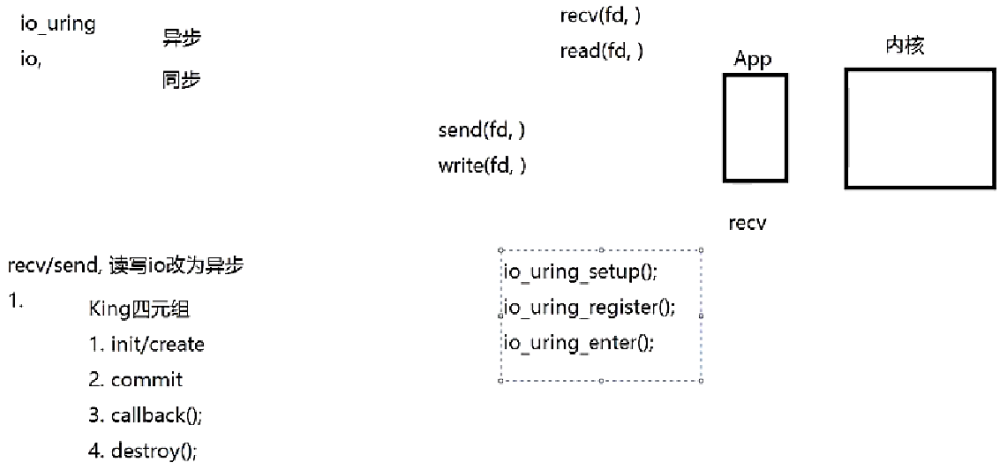
1. io_uring 首次引入 Linux 内核主线的版本是 **5.1**，但 5.4 版本才是首个生产级可用版本。
2. Linux 5.10（LTS）是 io_uring 功能成熟的标志性版本，被主流发行版广泛采用。
3. Ubuntu 20.04 LTS（5.4 内核）及以上版本均原生支持 io_uring，无需额外配置。

1.同步与异步

send\write;recv,read：同步，不返回无法向下面走

协程实现异步：不可读，我就去处理其他的事情，可读了我再切换回来

异步：发送请求和数据处理分开处理



使用fio测试：

posix sync：

```
// 测试命令
fio --name=test_sync \
    --filename=./fio_data_file \
    --size=1G \
    --time_based \
    --runtime=10 \
    --ioengine=psync \
    --direct=1 \
    --rw=randread \
    --bs=4k \
    --numjobs=1 \
    --iodepth=1 \
    --group_reporting
```

```
fio-3.36
Starting 1 process
test_sync: Laying out IO file (1 file / 1024MiB)
Jobs: 1 (f=1): [r(1)][100.0%][r=53.5MiB/s][r=13.7k IOPS][eta 00m:00s]
test_sync: (groupid=0, jobs=1): err= 0: pid=25293: Wed Dec 24 21:33:58 2025
  read: IOPS=13.6k, BW=53.2MiB/s (55.8MB/s)(532MiB/10001msec)
    clat (usec): min=34, max=1745, avg=72.88, stdev=45.43
     lat (usec): min=34, max=1745, avg=72.91, stdev=45.44
    clat percentiles (usec):
     |  1.00th=[   60],  5.00th=[   61], 10.00th=[   61], 20.00th=[   62],
     | 30.00th=[   62], 40.00th=[   63], 50.00th=[   64], 60.00th=[   65],
     | 70.00th=[   68], 80.00th=[   72], 90.00th=[   84], 95.00th=[  102],
     | 99.00th=[  258], 99.50th=[  404], 99.90th=[  668], 99.95th=[  775],
     | 99.99th=[ 1106]
   bw (  KiB/s): min=51232, max=56312, per=100.00%, avg=54514.11, stdev=1389.19, samples=
19
   iops        : min=12808, max=14078, avg=13628.53, stdev=347.30, samples=19
  lat (usec)   : 50=0.01%, 100=94.59%, 250=4.37%, 500=0.72%, 750=0.26%
  lat (usec)   : 1000=0.04%
```

io_uring:

```
fio --name=test_uring_sqpoll \
    --filename=./fio_data_file \
    --size=1G \
    --time_based \
    --runtime=10 \
    --ioengine=io_uring \
    --direct=1 \
    --rw=randread \
    --bs=4k \
    --numjobs=1 \
    --iodepth=32 \
    --sqthread_poll=1 \
    --group_reporting
```

```
fio-3.36
Starting 1 process
Jobs: 1 (f=1): [r(1)][100.0%][r=191MiB/s][r=49.0k IOPS][eta 00m:00s]
test_uring_sqpoll: (groupid=0, jobs=1): err= 0: pid=25502: Wed Dec 24 21:47:45 2025
  read: IOPS=49.2k, BW=192MiB/s (202MB/s)(1922MiB/10001msec)
    clat (usec): min=31, max=6342, avg=650.16, stdev=184.63
     lat (usec): min=32, max=6342, avg=650.20, stdev=184.63
    clat percentiles (usec):
     |  1.00th=[  178],  5.00th=[  482], 10.00th=[  545], 20.00th=[  578],
     | 30.00th=[  586], 40.00th=[  603], 50.00th=[  611], 60.00th=[  627],
     | 70.00th=[  644], 80.00th=[  676], 90.00th=[  824], 95.00th=[ 1074],
     | 99.00th=[ 1287], 99.50th=[ 1418], 99.90th=[ 1729], 99.95th=[ 1942],
     | 99.99th=[ 2343]
   bw (  KiB/s): min=184704, max=208176, per=100.00%, avg=197012.37, stdev=4178.62, sampl
es=19
   iops        : min=46176, max=52044, avg=49253.00, stdev=1044.66, samples=19
  lat (usec)   : 50=0.03%, 100=0.34%, 250=1.10%, 500=4.60%, 750=80.95%
  lat (usec)   : 1000=6.41%
```

49.2/13.6 = 3.6

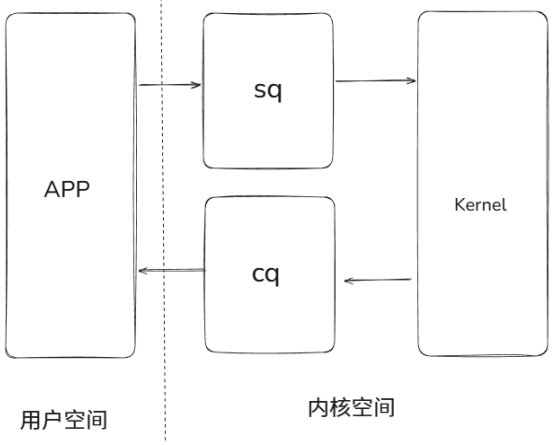
libaio:

```
fio --name=test_libaio \
    --filename=./fio_data_file \
    --size=1G \
    --time_based \
    --runtime=10 \
    --ioengine=libaio \
    --direct=1 \
    --rw=randread \
    --bs=4k \
    --numjobs=1 \
    --iodepth=32 \
    --group_reporting
```

```
engine=libaio, iodepth=32
fio-3.36
Starting 1 process
Jobs: 1 (f=1): [r(1)][100.0%][r=157MiB/s][r=40.2k IOPS][eta 00m:00s]
test_libaio: (groupid=0, jobs=1): err= 0: pid=25489: Wed Dec 24 21:45:15 2025
  read: IOPS=40.4k, BW=158MiB/s (166MB/s)(1579MiB/10001msec)
    slat (usec): min=15, max=2126, avg=23.26, stdev=11.67
    clat (usec): min=23, max=2893, avg=767.99, stdev=82.83
     lat (usec): min=54, max=2918, avg=791.25, stdev=84.16
    clat percentiles (usec):
     |  1.00th=[  693],  5.00th=[  709], 10.00th=[  717], 20.00th=[  725],
     | 30.00th=[  734], 40.00th=[  742], 50.00th=[  750], 60.00th=[  758],
     | 70.00th=[  775], 80.00th=[  791], 90.00th=[  832], 95.00th=[  889],
     | 99.00th=[ 1123], 99.50th=[ 1254], 99.90th=[ 1598], 99.95th=[ 1680],
     | 99.99th=[ 2180]
   bw (  KiB/s): min=159416, max=166235, per=100.00%, avg=161782.63, stdev=1561.92, sam
```

2.原理学习

io_uring的数据流向
1.APP的事件带着自己的身份证“私有数据”，扔到sq中，并通知内核可以取走了
2.Kernel取出sq的事件，并处理，处理完了就把该事件扔进cq
3.APP阻塞式等待cq中的事件，拿到就知道刚才的事件已经确实完成了，可以开始下面的业务了



手写一个echo的iouring程序：

- 基本数据结构
- **SQ (Submission Queue - 提交队列)**
    - **全称**：Submission Queue
    - **角色**：**生产者是用户程序，消费者是内核。**
    - **作用**：应用程序将想要执行的 I/O 请求（例如“读取文件 A 的前 1KB 数据”）封装成一个个 **SQE (Submission Queue Entry)**，放入这个队列中，告诉内核“我要做这些事”。

  CQ (Completion Queue - 完成队列)
    - **全称**：Completion Queue
    - **角色**：**生产者是内核，消费者是用户程序。**
    - **作用**：内核完成 I/O 操作后（无论成功还是失败），会生成一个 **CQE (Completion Queue Entry)** 放入这个队列。这个条目包含操作的结果（如读取到的字节数或错误码）。

```
// 管理结构体，保存两个缓冲区的相关信息
struct io_uring {
    struct io_uring_sq sq;
    struct io_uring_cq cq;
    unsigned flags;
    int ring_fd;

    unsigned features;
    int enter_ring_fd;
    __u8 int_flags;
    __u8 pad[3];
    unsigned pad2;

};
```

```
// 环形队列的相关信息
// 头指针、尾指针等
struct io_uring_sq {
    unsigned int *khead;
    unsigned int *ktail;
    unsigned int *kring_mask;
    unsigned int *kring_entries;
    unsigned int *kflags;
    unsigned int *kdropped;
    unsigned int *array;
    struct io_uring_sqe *sqes;

    unsigned int sqe_head;
    unsigned int sqe_tail;

    size_t ring_sz;
};

struct io_uring_cq {
    unsigned int *khead;
    unsigned int *ktail;
    unsigned int *kring_mask;
    unsigned int *kring_entries;
    unsigned int *koverflow;
    struct io_uring_cqe *cqes;

    size_t ring_sz;
};


struct io_uring_sqe {
    __u8    opcode;     /* type of operation for this sqe */
    __u8    flags;      /* IOSQE_ flags */
    __u16   ioprio;     /* ioprio for the request */
    __s32   fd;     /* file descriptor to do IO on */
    union {
        __u64   off;    /* offset into file */
        __u64   addr2;
        struct {
            __u32   cmd_op;
            __u32   __pad1;
        };
    };
    union {
        __u64   addr;   /* pointer to buffer or iovecs */
        __u64   splice_off_in;
        struct {
            __u32   level;
            __u32   optname;
```

```
        };
    };
    __u32   len;        /* buffer size or number of iovecs */
    union {
        __kernel_rwf_t  rw_flags;
        __u32       fsync_flags;
        __u16       poll_events;    /* compatibility */
        __u32       poll32_events;  /* word-reversed for BE */
        __u32       sync_range_flags;
        __u32       msg_flags;
        __u32       timeout_flags;
        __u32       accept_flags;
        __u32       cancel_flags;
        __u32       open_flags;
        __u32       statx_flags;
        __u32       fadvise_advice;
        __u32       splice_flags;
        __u32       rename_flags;
        __u32       unlink_flags;
        __u32       hardlink_flags;
        __u32       xattr_flags;
        __u32       msg_ring_flags;
        __u32       uring_cmd_flags;
        __u32       waitid_flags;
        __u32       futex_flags;
    };
    __u64   user_data;  /* data to be passed back at completion time
*/
    /* pack this to avoid bogus arm OABI complaints */
    union {
        /* index into fixed buffers, if used */
        __u16   buf_index;
        /* for grouped buffer selection */
        __u16   buf_group;
    } __attribute__((packed));
    /* personality to use, if used */
    __u16   personality;
    union {
        __s32   splice_fd_in;
        __u32   file_index;
        __u32   optlen;
        struct {
            __u16   addr_len;
            __u16   __pad3[1];
        };
    };
    union {
        struct {
            __u64   addr3;
```

```
            __u64   __pad2[1];
        };
        __u64   optval;
        /*
         * If the ring is initialized with IORING_SETUP_SQE128, then
         * this field is used for 80 bytes of arbitrary command data
         */
        __u8    cmd[0];
    };
};
```

这两个队列不仅是数据的容器，更是 `io_uring` 高性能的秘诀：

1. **零拷贝通信（共享内存）**： `sq` 和 `cq` 指向的环形缓冲区是通过 `mmap` 映射到内存中的。这意味着**用户态和内核态共享同一块物理内存**。应用程序写入 SQ 的数据，内核可以直接看到，不需要像传统 `read/write` 系统调用那样在用户态和内核态之间来回拷贝数据。
2. **批量处理**： 应用程序可以一次性向 SQ 放入多个请求（例如 100 个），然后只调用一次系统调用通知内核，内核处理完后批量填入 CQ。这极大地减少了上下文切换的开销。
3. **无锁（或低锁）机制**： 由于是环形缓冲区，通过巧妙的 Head（头）和 Tail（尾）指针设计，在单生产者/单消费者场景下可以实现无锁并发。

`sq` 和 `cq` 如何交互？（工作流程）

交互过程就像一个**餐厅的点单和出餐流程**：

1. **用户提交 (App -> SQ)**：
   - 用户程序获取一个空的 SQE（点单纸）。
   - 填入操作（如：`IORING_OP_READV`），设置文件描述符、缓冲区地址等。
   - 将 SQE 放入 **SQ 环形缓冲区**的尾部。
   - 更新 SQ 的 Tail 指针。
2. **通知内核 (System Call)**：
   - 用户程序调用 `io_uring_enter()` 系统调用。这相当于按了一下服务铃：“我有新订单了，请处理”。
   - *(注：在轮询模式下，甚至连这个系统调用都可以省去，内核会自动检查 SQ)*。
3. **内核处理 (Kernel Consumes SQ)**：
   - 内核读取 SQ 里的请求。
   - 内核执行实际的 I/O 操作（通常是异步的，交给硬件或后台线程）。
4. **内核回填 (Kernel -> CQ)**：
   - I/O 操作完成后，内核生成一个 CQE（包含结果，如 `res` 返回值）。
   - 内核将 CQE 放入 **CQ 环形缓冲区**的尾部。
   - 内核更新 CQ 的 Tail 指针。

5. **用户收割 (App Consumes CQ)**:
    - 用户程序检查 CQ 的 Head 和 Tail 指针。如果 `Head != Tail`，说明有新完成的任务。
    - 用户程序从 CQ 中读取 CQE，获取操作结果。
    - 用户更新 CQ 的 Head 指针，表示“这个结果我已经处理了”。

2.测试

测试环境：

```
Linux liyumin 6.14.0-37-generic #37~24.04.1-Ubuntu SMP PREEMPT_DYNAMIC
Thu Nov 20 10:25:38 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
```

io_uring

```
// 包大小为512bytes
liyumin@liyumin:~/Desktop/0voice/io_uring$ ./test_qps_tcpclient -s
127.0.0.1 -p 9999 -t 50 -c 100 -n 1000000
-s: 127.0.0.1
-p: 9999
-t: 50
-c: 100
-n: 1000000
success: 1000000, failed: 0, time_used: 7873, qps: 127016


// 包大小为256bytes
liyumin@liyumin:~/Desktop/0voice/io_uring$ ./test_qps_tcpclient -s
127.0.0.1 -p 9999 -t 50 -c 100 -n 1000000
-s: 127.0.0.1
-p: 9999
-t: 50
-c: 100
-n: 1000000
success: 1000000, failed: 0, time_used: 7900, qps: 126582

// 包大小为128bytes
liyumin@liyumin:~/Desktop/0voice/io_uring$ ./test_qps_tcpclient -s
127.0.0.1 -p 9999 -t 50 -c 100 -n 1000000
-s: 127.0.0.1
-p: 9999
-t: 50
-c: 100
-n: 1000000
success: 1000000, failed: 0, time_used: 7834, qps: 127648
```

```
liyumin@liyumin:~/Desktop/0voice/io_uring$ ./test_qps_tcpclient -s
127.0.0.1 -p 9999 -t 50 -c 100 -n 1000000
-s: 127.0.0.1
-p: 9999
-t: 50
-c: 100
-n: 1000000
success: 1000000, failed: 0, time_used: 7880, qps: 126903
```

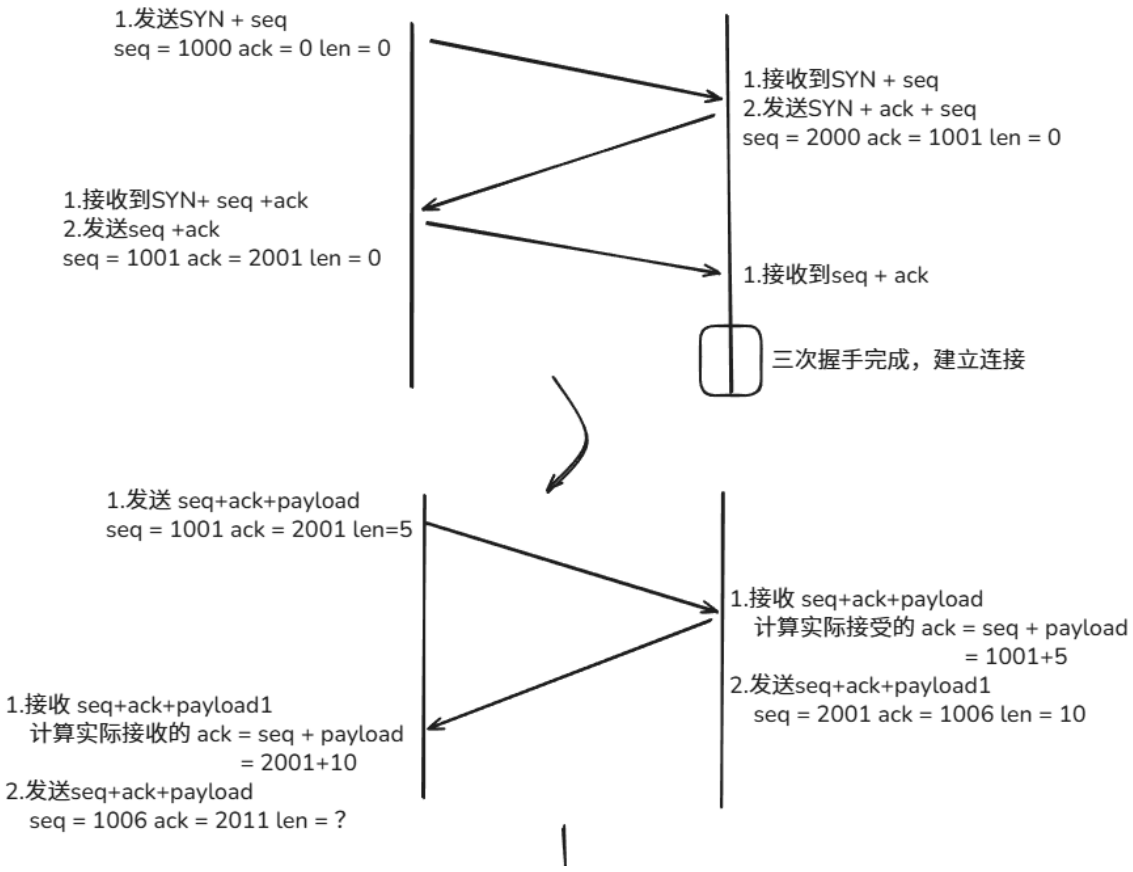
epoll

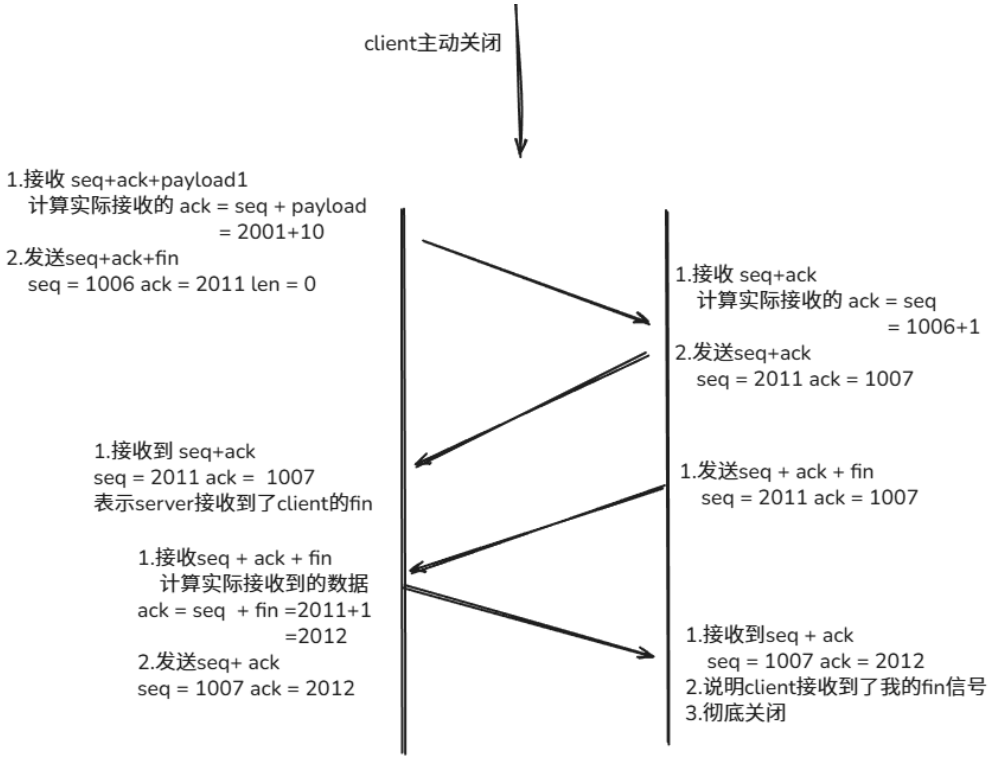
```
// 包大小为512bytes
liyumin@liyumin:~/Desktop/0voice/io_uring$ ./test_qps_tcpclient -s
127.0.0.1 -p 2048 -t 50 -c 100 -n 1000000
-s: 127.0.0.1
-p: 2048
-t: 50
-c: 100
-n: 1000000
success: 1000000, failed: 0, time_used: 9545, qps: 104766


// 包大小为256bytes
liyumin@liyumin:~/Desktop/0voice/io_uring$ ./test_qps_tcpclient -s
127.0.0.1 -p 2048 -t 50 -c 100 -n 1000000
-s: 127.0.0.1
-p: 2048
-t: 50
-c: 100
-n: 1000000
success: 1000000, failed: 0, time_used: 9460, qps: 105708


// 包大小为128bytes
liyumin@liyumin:~/Desktop/0voice/io_uring$ ./test_qps_tcpclient -s
127.0.0.1 -p 2048 -t 50 -c 100 -n 1000000
-s: 127.0.0.1
-p: 2048
-t: 50
-c: 100
-n: 1000000
success: 1000000, failed: 0, time_used: 9494, qps: 105329

// 包大小为64bytes
```

```
liyumin@liyumin:~/Desktop/0voice/io_uring$ ./test_qps_tcpclient -s
127.0.0.1 -p 2048 -t 50 -c 100 -n 1000000
-s: 127.0.0.1
-p: 2048
-t: 50
-c: 100
-n: 1000000
success: 1000000, failed: 0, time_used: 9459, qps: 105719
```

3.面试题目

1.三次握手、四次挥手





2.udp并发如何做?

UDP 的并发和 TCP 的并发有着**本质的区别**，因为 UDP 没有"连接"的概念，也没有 `accept` 返回的新 fd。

在 TCP 中，每个客户端都有一个独立的 `connfd`，你只要把这个 fd 扔给一个线程或者 epoll 去管就行了。

但在 UDP 中，通常**只有一个** `sockfd` 绑定在 9999 端口。成千上万个客户端发来的包，全部挤在这**唯一的一个**接收缓冲区里。

如果不做处理，单线程 recvfrom 肯定处理不过来。以下是实现 UDP 高并发的三种境界，从**入门**到**工业级**再到**核武级**：

第一境界：单 Socket + 多线程工作池 (Producer-Consumer)

这是最容易想到的模型，适合**业务逻辑重、但流量不是特别大**的场景。

原理

1. **IO 线程（1个）**：死循环调用 `recvfrom`。它只负责收包，不负责处理。
2. **队列**：IO 线程收到数据后，封装成一个任务对象（包含 `data` + `peer_addr`），扔到一个线程安全的队列里。
3. **工作线程池（N个）**：从队列里取数据，解析协议、计算、回包（`sendto`）。

局限性

- **瓶颈仍在 IO**：那个唯一的 IO 线程是瓶颈。如果每秒有 100万个包（1M PPS），一个线程光是 `recvfrom` 和 `memcpy` 就累死了，队列也会爆满。
- **锁竞争**：队列需要加锁，多线程竞争锁会消耗 CPU。

第二境界：多线程争抢同一个 Socket (Simple Multi-threading)

为了解决 IO 瓶颈，你可能会想：**能不能让多个线程同时去 `recvfrom` 同一个 fd?**

原理

1. 主线程创建 socket，bind 端口。
2. 创建 N 个线程。
3. **所有线程**都对这个**同一个 fd** 调用 `recvfrom`。

结果

Linux 内核允许多个线程等待同一个 socket。当一个包来的时候，内核会唤醒其中一个线程（惊群效应在现代内核中有所缓解）。

局限性

- **锁竞争移到了内核**：虽然用户态不用锁了，但内核里的 Socket 接收队列是受锁保护的。多个线程同时去抢内核缓冲区里的数据，内核层面的 `spinlock` 竞争会非常激烈。
- **乱序问题**：同一个客户端发来的包 1、2、3，可能被线程 A、B、C 分别拿走，处理顺序无法保证。

第三境界：工业级方案 `SO_REUSEPORT` (多队列/多端口)

这是 Nginx、QUIC、DNS 服务器等高性能 UDP 服务的标准做法。

核心痛点解决

既然一个 Socket 队列会导致锁竞争，那我就**搞 N 个 Socket，绑定到同一个端口上!**

实现步骤

1. **开启选项**：在 `bind` 之前，必须设置 `SO_REUSEPORT` 选项。

```
int opt = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt));
```

2. **创建多个 Socket**：

- 如果你有 4 个 CPU 核，你就创建 4 个线程。
- 每个线程里**都创建自己的 socket**。
- 每个线程**都 bind 到 0.0.0.0:9999** (是的，开启了 REUSEPORT 就不会报错 Address already in use)。

3. **独立循环**：
   - 每个线程只 `recvfrom` 自己的那个 fd。
   - 完全无锁，完全并行。

内核是如何分配流量的?

内核会根据数据包的 **(源IP, 源端口, 目的IP, 目的端口)** 进行 Hash 运算。

- **结果**：同一个客户端发来的包，始终会被哈希到**同一个线程**的 Socket 里。
- **优势**：
  1. **无锁**：线程之间互不干扰。
  2. **CPU 亲和性**：数据都在同一个 CPU 核上处理，缓存（L1/L2 Cache）命中率极高。
  3. **解决乱序**：同一个客户端由同一个线程处理，逻辑上容易保序。

第四境界：核武级优化 (Batching & io_uring)

即便用了 `SO_REUSEPORT`，如果你一次只读一个包，系统调用（System Call）的开销依然很大。

1. 使用 `recvmmsg` (注意是多了一个 m)

不要用 `recvfrom`，用 `recvmmsg`。

- `recvfrom`：一次系统调用，拿 1 个包。
- `recvmmsg`：一次系统调用，拿 **N 个包**（比如 32 个）。
- **效果**：系统调用次数减少 30 倍，吞吐量大幅提升。

2. 结合你的 `io_uring`

既然你在学 `io_uring`，它对 UDP 的提升比 TCP 更大!

- 你可以 `prep_recv` 100 个 buffer 下去。
- 当大量 UDP 包涌入时，内核直接把数据填满这些 buffer，然后通过 CQE 告诉你。
- **完全异步，批量收割。**

总结代码示例 (SO_REUSEPORT 伪代码)

这就是高性能 UDP Server 的骨架：

```
// 线程函数
void *worker_thread(void *arg) {
    int port = *(int*)arg;

    // 1. 每个线程创建自己的 Socket
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    // 2. 关键：开启 SO_REUSEPORT
    int opt = 1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt));

    // 3. 绑定同一个端口
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = INADDR_ANY;
    bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));

    // 4. 独立的 Event Loop (或者直接 while recvmmsg)
    struct mmsghdr msgs[32]; // 批量接收缓冲区
    while(1) {
        // 一次拿 32 个包
        int n = recvmmsg(sockfd, msgs, 32, 0, NULL);
        for(int i=0; i<n; i++) {
            process_packet(msgs[i]); // 处理数据
            // sendto 直接回包，因为 UDP 不需要建立连接
        }
    }
}

int main() {
    int port = 9999;
    // 根据 CPU 核数开启线程
    for(int i=0; i<4; i++) {
        pthread_create(&tid[i], NULL, worker_thread, &port);
    }
    pthread_join(...);
}
```

怎么区分“用户”?

既然没有 connection fd，你怎么知道谁是谁?

- **TCP**：靠 `fd` 区分用户。
- **UDP**：靠 `sockaddr` (IP + Port) 区分用户。
  - 你需要自己在内存里维护一个 **Hash Map** (也就是 Session 表)。
  - Key = `Client IP + Port`
  - Value = `User Context` (用户状态、登录信息等)
  - 收到包后，先查 Map，查到了就是老用户，查不到就是新用户。

总结： 想要 UDP 高并发，不要去想怎么多线程抢一个 fd，直接上 `SO_REUSEPORT` **+ 多线程 +** `recvmmsg`。这就是现代 QUIC/HTTP3 服务器的底层实现方式。

3.TCP与UDP区别?

TCP的确认机制、拥塞算法写的太全面了，导致了占用资源、太谨慎了，没法修改；

goole就看到了这个问题，底层还是基于UDP，把一些可靠性、流量控制的东西加到了应用层，大大提高了灵活性；

TCP 是**保姆**：它帮你搞定了一切（顺序、丢包、流控），你只要往里写数据就行，但它**重、慢、复杂**。

UDP 是**底座**：它只负责把你交给它的东西扔出去，剩下的（丢没丢、乱没乱）它不管。它**轻、快、自由**。

现在的趋势是：越来越多的高性能应用（游戏、流媒体、RPC）开始**抛弃 TCP**，转而使用 **UDP + 应用层可靠性封装 (如 QUIC, KCP, UDT)**，以此来获得对传输控制的绝对主导权。

特性	TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
连接性	**面向连接** (三次握手/四次挥手)	**无连接** (直接发，Bind 完即发)
可靠性	**强可靠** (不丢包、不乱序、不重复)	**不可靠** (尽最大努力交付，丢了不管)
数据形态	**字节流** (Byte Stream，无边界，有粘包)	**数据报** (Datagram，有边界，无粘包)
流量/拥塞控制	**有** (滑动窗口、慢启动、拥塞避免、快重传)	**无** (不管网络堵不堵，只要网卡能发就发)
头部开销	**大** (20 ~ 60 字节)	**小** (固定 8 字节)
传输速度	较慢 (受 RTT、拥塞控制影响)	极快 (受限于物理带宽和 CPU)
系统资源	高 (维护 TCB、定时器、缓冲区、重传队列)	低 (只维护简单的 Socket 结构)
应用场景	网页(HTTP), 文件(FTP), 邮件(SMTP), 数据库	视频/语音(RTP), DNS, 游戏, **QUIC/HTTP3**

二、 深度剖析（配合你的内核开发背景）

1. 数据的“边界感” (粘包问题的根源)

这是你之前问“粘包”时涉及的核心点。

- **TCP 是水流**：你调用 `send("hello")`，内核只是把它拷贝到发送缓冲区。发送时，内核可能把 "hel" 发在一个包，"lo" 发在下一个包；也可能把几次 send 的数据合在一起发（Nagle 算法）。**接收端必须自己切分数据（包头+包体）。**
- **UDP 是包裹**：UDP 头部有个 `Length` 字段。你 `sendto` 发送 100 字节，接收端 `recvfrom` 要么收到完整的 100 字节，要么什么都收不到（丢包）。**UDP 天然保留消息边界，不存在粘包。**

2. 内核中的“状态机” (TCP 的沉重)

- **TCP**：内核为每个 TCP 连接维护了一个庞大的数据结构 `tcp_sock` (TCB)。
  - 它要记录：发送窗口、接收窗口、拥塞窗口 (cwnd)、慢启动阈值 (ssthresh)、各种定时器 (重传、坚持、保活)、乱序队列 (OFO queue) 等。
  - 每一次收发包，内核都要更新这个状态机，计算 RTT，调整窗口。**这就是 TCP 耗 CPU 的原因。**
- **UDP**：内核几乎不维护状态。

- `sendto` 几乎是直通网卡驱动的队列。
- 这就是为什么 UDP 适合做高吞吐量的场景（如 DPDK 经常用来跑 UDP）。

3. 拥塞控制：君子 vs 流氓

- **TCP (君子)**：TCP 极度在意网络的“健康”。如果发现丢包（网络堵了），它会主动减小发送窗口（`cwnd` 减半或归一），避让网络。
- **UDP (流氓/自私)**：UDP 根本不管网络堵不堵。哪怕路由器已经在丢包了，UDP 依然会以 100% 的速度向外灌数据，甚至把 TCP 的流量挤死（导致 TCP 进一步退让）。
  - *注：现代基于 UDP 的可靠协议（如 KCP、QUIC）通常会在用户层自己实现拥塞控制，以防止打崩网络。*

4. 头部开销对比

- **UDP 头 (8字节)**：
  - 源端口 (2) + 目的端口 (2) + 长度 (2) + 校验和 (2)。
  - 非常精简，几乎全是干货。
- **TCP 头 (20字节 + Options)**：
  - 序列号 (Seq)、确认号 (Ack)、标识位 (SYN/ACK/FIN/RST)、窗口大小、校验和、紧急指针...
  - 如果是 HTTP 请求，TCP 头部可能比 HTTP 头部还要大。

三、 现代网络的演进：为什么 UDP 正在逆袭?

在通信工程领域，一定要关注 **QUIC (Quick UDP Internet Connections) / HTTP3**。

传统的观念是：“想要可靠就用 TCP”。 但在移动互联网和高延迟网络下，TCP 有**致命缺陷**：

1. **队头阻塞 (Head-of-Line Blocking)**：TCP 只要丢失一个包，后面所有收到的包都不能给应用层（必须等那个丢失的包重传回来）。这会导致视频卡顿。
2. **握手太慢**：TCP 握手 1 RTT + TLS 握手 1~2 RTT。建立连接太慢。

UDP 的逆袭 (QUIC)： Google 提出的 QUIC 协议，**底层使用 UDP**，但在**用户态 (User Space)** 实现了类似 TCP 的可靠传输、流量控制和加密。

- **0-RTT 建连。**
- **无队头阻塞**（基于 Stream 的多路复用）。
- **连接迁移**（手机从 WiFi 切到 4G，IP 变了，TCP 会断，但 QUIC 可以靠 Connection ID 保持不断）。

4.TCP粘包?

应用层的TCP包加长度；分隔符

出现的原因：

首先要纠正一个概念：**TCP 协议本身从来就没有“粘包”这个词，这是应用层开发中对一种现象的俗称。**

TCP 是**“流式协议”（Byte Stream）**，它的特点是像水流一样，没有边界。你发的时候是 `send("A")`，`send("B")`，TCP 只知道你发了一堆字节流。

粘包什么时候会出现？ 简单来说，就是**“发送太快”**或者**“接收太慢”**的时候。

以下是三种最典型会出现粘包的场景：

1. 发送方发得太快（Nagle 算法 + 缓冲区）

- **场景**：你的 Client 代码在一个 `for` 循环里疯狂调用 `send`，每次只发几个字节（比如 "Hello"）。
- **原理**：TCP 为了提高效率，默认开启了 **Nagle 算法**。它会想：“哎呀，你每次就发这么点数据，光 TCP 头都比数据大了，太浪费网费了。我先攒一攒，攒够了一大块再发给你。”
- **结果**：你调用了 10 次 `send("Hello")`，结果网线上只跑了一个大包 `"HelloHelloHello..."`。
- **接收端**：Server 一次 `recv`，直接收到一坨连在一起的字符，分不清哪个是哪个。

2. 接收方收得太慢（积压）

- **场景**：Server 正在忙着处理复杂的业务逻辑（比如写数据库），没空去调用 `recv`。
- **原理**：Client 继续在发数据。这些数据到了 Server 的网卡后，操作系统看应用程序没来取，就先把数据堆在 **内核的接收缓冲区（Recv Buffer）** 里。
- **结果**：等 Server 忙完了，终于调用了一次 `recv(fd, buf, 2048)`。
- **现象**：这时候缓冲区里可能已经积压了 Client 发来的 3 个请求。Server 这一铲子下去，直接把这 3 个请求全读到了一个 buffer 里。这就是粘包。

3. 数据包太大（拆包/半包）

虽然这叫“拆包”，但往往和粘包一起讨论。

- **场景**：你发了一个 4KB 的数据包，但 Server 的 `recv` buffer 只有 2KB，或者 TCP 这一段只传过来 1KB。
- **结果**：Server 第一次 `recv` 只读到了半截数据。剩下的半截数据，会在下一次 `recv` 时读出来，而下一次 `recv` 可能还会顺带读到下一个请求的开头。
- **现象**：**上一个包的尾巴 + 下一个包的头** 粘在了一起。

怎么解决粘包？（核心面试题）

既然 TCP 是流，没有边界，应用层就必须**自己画边界**。常用的三种方法：

1. **定长包（Fixed Length）**：
   - 规定每个包必须是 512 字节。不够就填 `0` 补齐。
   - *缺点*：浪费流量。
2. **特殊分隔符（Delimiter）**：
   - 像 HTTP 协议那样，用 `\r\n` 或者 `\r\n\r\n` 结尾。
   - 或者像 JSON 字符串用 `{}` 匹配。
   - *缺点*：内容里不能含有分隔符，由于需要扫描全文，效率稍低。
3. **包头+包体（Length Header - 推荐标准做法）**：
   - 在数据包的最前面加上 4 个字节的 `int`，存数据包的长度。
   - **解析逻辑**：
     1. 先读 4 个字节，解出来的数字是 `Len`。
     2. 再向后读 `Len` 个字节，这才是完整的数据。
   - *你的代码改进方向*：如果要上生产环境，必须加上这个“包头”。

4.补充网络知识

（1）mmap

`mmap` (Memory Map) 是 Linux 系统编程中一个非常基础且强大的概念。

用一句话概括：`mmap` **是一种让“文件”看起来像“内存数组”的技术。**

它打破了“文件读写”和“内存操作”之间的界限，是你理解 **零拷贝 (Zero Copy)**、**DPDK 底层原理** 以及 **io_uring** 的关键钥匙。

一、 传统读写 vs mmap

为了理解它，我们先看没有 `mmap` 时我们是怎么读文件的（比如用 `read` 系统调用）。

1. 传统的 `read()` 模式（两次拷贝）

假设你要把磁盘上的一个文件读到你的程序里：

1. **DMA 拷贝**：磁盘控制器把数据拷贝到 **内核缓冲区 (Page Cache)**。
2. **CPU 拷贝**：CPU 把数据从 **内核缓冲区** 拷贝到你的 **用户缓冲区 (User Buffer)**。

3. **结果**：数据在内存里存了两份（内核一份，你一份），且发生了一次系统调用。

2. `mmap` 模式（零拷贝）

当你对一个文件调用 `mmap` 时：

1. **建立映射**：操作系统仅仅是在你的 **进程虚拟地址空间** 中分配了一段地址，把它"映射"到内核的文件缓存（Page Cache）上。**此时并没有真的读数据。**
2. **缺页中断 (Page Fault)**：当你第一次去读写这段内存时，CPU 发现数据不在物理内存里，触发中断。
3. **DMA 拷贝**：内核被唤醒，指挥 DMA 直接把磁盘数据搬到 **内核缓冲区 (Page Cache)**。
4. **直接访问**：你的进程直接读写这段 **内核缓冲区**（因为映射关系已经建立了）。
5. **结果**：数据在内存里只有一份（就在内核里），你直接操作它，省去了从内核拷给你的那次 CPU 拷贝。

二、 mmap 的三个核心用途

你在高性能编程中会遇到 `mmap` 的三种"分身"：

1. 文件映射 (File-backed Mapping) —— 读写大文件

这是最常用的场景。比如 MongoDB 处理几个 GB 的数据文件，它不是用 `read/write`，而是直接 `mmap` 进内存。

- **好处**：像操作数组一样操作文件（`data[i] = 'a'`），操作系统会自动帮你把修改脏页写回磁盘。
- **场景**：数据库存储引擎、加载动态库（`.so` 文件）。

2. 匿名映射 (Anonymous Mapping) —— 分配大内存

当你调用 `malloc` 申请很小的内存（比如 10字节）时，它是从堆（Heap）里切一块给你。 但如果你申请 **大块内存**（比如 `malloc(2MB)`），`malloc` 底层就会自动切换成 `mmap`。

- **含义**：它不映射任何文件，只是向操作系统要一块纯净的物理内存。
- **场景**：`malloc` 的底层实现。

3. 共享内存 (Shared Memory) —— 进程间通信 (IPC)

既然 `mmap` 可以把物理内存映射到进程空间，那如果 **两个进程映射同一块物理内存** 会怎样？

- **结果**：进程 A 写的数据，进程 B 瞬间就能看到！
- **场景**：这是 Linux 下最快的进程间通信方式（因为根本不需要内核传递数据）。`io_uring` 就是靠这个让内核和用户态共享队列的。

三、 mmap 与你的 DPDK 项目有什么关系?

这一点至关重要，**DPDK 的底层基石就是 mmap。**

回顾你的 DPDK 初始化流程：

1. **UIO/VFIO 驱动**： 你把网卡从内核驱动里"抢"过来，绑定到 `uio_pci_generic` 或 `vfio-pci`。
2. **映射网卡寄存器 (BAR Space)**： DPDK 会通过 `mmap` 将网卡硬件上的 **寄存器（控制收发队列的门铃等）** 直接映射到用户态内存。
   - **效果**：你在用户态写一个内存地址，实际上是在直接指挥网卡硬件工作！
3. **Hugepages (大页内存)**： DPDK 启动时会申请几 GB 的大页内存。这本质上也是一种特殊的 `mmap`（映射 `/dev/hugepages` 文件）。
   - **效果**：DPDK 拿到了物理地址连续的内存，直接把这块内存的物理地址告诉网卡（DMA），网卡收到的包就直接往这里扔。

总结：如果没有 `mmap`，DPDK 就无法在用户态直接控制硬件，也无法实现零拷贝。

四、 mmap 的优缺点总结

优点：

1. **快**：减少了内核到用户的一次内存拷贝。
2. **省内存**：多个进程可以共享同一个文件的只读副本（比如 C 标准库 `libc.so` 在内存里只有一份，所有程序都 `mmap` 它）。
3. **方便**：不用在 `lseek` 来回跳转，像访问数组一样访问文件。

缺点：

1. **小文件不划算**：建立映射本身有开销（修改页表、TLB），如果文件很小（几 KB），直接 `read` 反而更快。
2. **缺页中断不可控**：读写 `mmap` 的内存时，如果数据不在物理内存，会触发缺页中断，这会导致 **微秒级的阻塞。**
   - *注：* 这就是为什么 DPDK 要用 **Hugepages** 并预先 `Touch` 一遍内存，就是为了防止运行时发生缺页中断导致抖动。

一句话概括

`mmap` 就是给操作系统开个后门，让你在用户态直接通过指针，去操作内核管理的内存（文件缓存）甚至硬件设备（网卡），从而消灭了中间商赚差价（拷贝开销）。

（2）io_uring

`io_uring` 是 Linux 内核在 5.1 版本（2019年由 Facebook 的 Jens Axboe 大神开发）引入的一套**全新的、高性能异步 I/O 接口。**

它不仅仅是用来替代 `epoll` 的，它更像是一次 Linux I/O 子系统的"重构"。为了让你彻底理解它，我们从**背景痛点**、**核心架构**、**三种模式**以及**实战对比**这四个维度来讲。

一、 为什么要搞个 io_uring？（Epoll 不够用吗？）

在 `io_uring` 出现之前，Linux 的高性能 I/O 主要靠 `epoll`。但随着硬件越来越快（NVMe SSD、100G 网卡），`epoll` 暴露出了一些"由于设计年代久远"带来的娘胎里的毛病：

1. **系统调用开销（Syscall Overhead）**：
   - `epoll` 只是通知你"有数据了"，你还得自己调 `read()` / `write()`。
   - 如果处理 100 万个小包，你就要调 100 万次 `read`。每一次系统调用（用户态切内核态）都要消耗 CPU，加上 Spectre/Meltdown 漏洞补丁的影响，这个开销变得昂贵。
2. **数据拷贝（Copy）**：
   - 虽然 `epoll_wait` 本身比较高效，但后续的数据读写依然需要在内核缓冲区和用户缓冲区之间拷贝。
3. **对磁盘 I/O 支持分裂**：
   - `epoll` 是为**网络**（Socket）生的。Linux 虽然有原生的 AIO（异步 I/O），但那个 AIO 极其难用（只支持 Direct I/O，不支持缓存 I/O），导致 Nginx、MySQL 等软件为了处理磁盘文件，还得搞线程池模拟异步。

io_uring 的目标就是：一套接口，统一网络和磁盘，把系统调用和内存拷贝降到最低。

二、 核心架构：双环形队列 + 共享内存

`io_uring` 的名字里，"ring" 是精髓。它在用户态和内核态之间建立了两条**环形队列（Ring Buffer）**，并且这两块内存是**用户和内核共享的。**

1. 两条队列

- **SQ (Submission Queue，提交队列)**：
  - 你是生产者，内核是消费者。
  - 你想读文件、发数据，就往 SQ 里扔一个请求（SQE - Submission Queue Entry）。
- **CQ (Completion Queue，完成队列)**：
  - 内核是生产者，你是消费者。
  - 内核处理完了，把结果（成功/失败、读了多少字节）写到 CQ 里（CQE - Completion Queue Entry）。

2. 为什么这样设计最快?

- **零拷贝指令传输**:
  - 因为 SQ 和 CQ 是映射到同一块物理内存的（mmap）。你往 SQ 写指令，内核直接就能看见，**不需要**像 `read/write` 系统调用那样把参数从用户栈拷贝到内核栈。
- **生产-消费模型**:
  - 这是一个标准的无锁（Lock-free）单生产者单消费者模型。只要按照顺序读写，连锁都不用加，效率极高。

三、 io_uring 的“三档变速”模式

这是 `io_uring` 最骚的操作，它允许你根据应用场景选择不同的“档位”:

档位 1：中断驱动模式 (默认)

- **玩法**：你把请求扔进 SQ，然后调用 `io_uring_enter()` 系统调用踢一脚内核：“干活了！”。内核干完活，发个中断或者信号通知你。
- **适用**：普通的高并发应用。
- **对比**：比 `epoll` 稍微快一点点，因为它是批量提交的（一次 syscall 提交 100 个请求）。

档位 2：轮询模式 (IOPOLL)

- **玩法**：主要针对磁盘 I/O。内核不再等待硬件中断，而是主动去轮询磁盘驱动器。
- **适用**：对延迟极其敏感的数据库系统。

档位 3：内核轮询模式 (SQPOLL) —— 真正的“大杀器”

- **玩法**：内核会启动一个**内核线程 (Kernel Thread)**，专门盯着 SQ 队列。
  - 你（用户态）：只管往 SQ 内存里写数据，**完全不需要**调用任何系统调用。
  - 内核线程：看到 SQ 有东西，立马拿走去执行。
- **效果**：**0 系统调用!**
- **对比**：这实际上就是**在内核层面实现了 DPDK 的轮询思想**。DPDK 是你在用户态死循环轮询网卡；`io_uring` 是内核线程死循环轮询你的请求队列。

四、 IO 模型本质区别：Reactor vs Proactor

面试或技术选型时，这是最本质的区别:

1. **Epoll 是 Reactor（反应堆）模式:**
   - **逻辑**：Epoll 告诉你“这个水管**有水了**（Readable）”。
   - **动作**：你自己拿桶去接水（调用 `read`，CPU 参与数据搬运）。

- **缺点**：如果水流很大，你接水的动作会占用大量 CPU 时间。

2. **io_uring 是 Proactor（前摄器）模式：**
   - **逻辑**：你给内核一个空桶，告诉内核“把这个桶**装满**”。
   - **动作**：内核自己去接水，接满了直接把桶给你。告诉你“**任务完成**（Complete）”。
   - **优点**：真正做到了应用层不需要关心数据拷贝的过程。

五、 总结与对比表

特性	Epoll	io_uring
设计思路	只要就绪就通知 (Reactor)	只要做完就通知 (Proactor)
系统调用	极多 (每次读写都要)	极少 (批量提交) 或 **0** (SQPOLL模式)
适用范围	主要是网络	**网络 + 磁盘** (全能)
内存交互	拷贝 (Copy)	共享内存 (Shared Memory)
编程难度	中等	较难 (通常使用 `liburing` 库封装)
性能极限	很高	**极高** (接近硬件极限)

给你的建议

既然你在做 DPDK，你现在的技术栈处于 **Linux 网络栈的两个极端**：

1. **DPDK**：彻底抛弃内核，自己管硬件。
2. **io_uring**：最大程度利用内核，让内核跑得像 DPDK 一样快。

如果你的业务场景允许独占网卡，DPDK 依然是王（因为它连内核网络协议栈都省了）。

但如果你的业务需要通用的 Linux 功能（防火墙、容器），或者需要同时处理极高并发的磁盘 I/O，io_uring 是未来的唯一标准。现在像 Nginx (实验性支持)、Redis 6.0、Netty、RocksDB 都在全面拥抱 io_uring。

（3）页表

既然你刚才问到了 `mmap` 和 DPDK，**“页表” (Page Table)** 其实是这一切背后的“幕后黑手”。

如果不理解页表，你就无法真正理解为什么 DPDK 要用“大页内存 (Hugepages)”，也无法理解为什么内存访问会有快慢之分。

用最通俗的话来说：**页表就是操作系统手里的“房号对照本”。**

一、 为什么要有个“对照本”？（虚拟 vs 物理）

想象一下，你（作为一个程序进程）入住了一家无限大的“虚拟酒店”。

1. **你的视角（虚拟地址）**：
    - 前台给你一把钥匙，上面写着“101号房”。
    - 你觉得自己拥有从 1号 到 无穷大号 所有的房间。哪怕你只存了一个整数，你也觉得它是连续存放在“101号”的。
    - **这就是虚拟内存**：每个程序都觉得自己独占了整个 CPU 和内存，拥有连续的地址空间。
2. **操作系统的视角（物理地址）**：
    - 实际上，物理内存（RAM条）是有限的，而且非常碎。
    - “101号房”里的数据，实际上可能被存在了物理内存的角落里（比如物理地址 0x9999）。
    - “102号房”的数据，可能被扔到了十万八千里外的物理地址 0x1111。

矛盾来了：

你（CPU 执行指令时）发出的命令是：“去读 虚拟地址 101 的数据”。

内存条（硬件）只认：“给我 物理地址，否则免谈”。

解决办法：

中间需要一个翻译。这个翻译手里拿着一本**“对照本”**：

> “查一下，虚拟地址 101 对应 物理地址 0x9999。好，去 0x9999 拿数据。”

这个“对照本”，就是页表。

二、 页表是怎么工作的？

如果让 CPU 每读一个字节都去查一次表，那太慢了。所以 Linux 采用了一种**“打包映射”**的策略。

1. 也是“页” (Page)

Linux 不会一个字节一个字节地记录，而是把内存切成一块一块的，每一块叫一个**“页” (Page)**。

- **标准大小**：通常是 **4KB** (4096 字节)。

这就像搬家，你不会一双筷子一双筷子地搬，而是先把东西装进**箱子**里。

- **页表记录的是**：“虚拟箱子号” -> “物理箱子号”。
- 箱子内部的偏移量（第几双筷子）是不变的。

2. MMU (内存管理单元)

CPU 里面有个专门的硬件芯片叫 **MMU**。

- 它的工作就是**查页表**。
- 当你的 C 代码写 `int a = *p;` 时，`p` 是虚拟地址。CPU 把 `p` 扔给 MMU，MMU 疯狂查表，算出物理地址，然后发给内存条。这个过程对程序员是**透明**的。

三、 为什么 Linux 的页表那么复杂？（多级页表）

你可能会想：*“搞个大数组，第 1 项对应物理页 1，第 2 项对应物理页 2，不就行了吗？”*

不行，因为太大了。

- 算笔账：

  在 64 位系统下，虚拟地址空间非常大。如果用一个单层的大数组来记录所有映射关系，光是存放这个“对照本”本身，就需要几百 GB 的内存！这显然不可能。

- Linux 的方案：多级页表 (Multi-level Page Table)

  Linux 采用了一种“树状结构”或者说“多级目录”的方式。就像查字典：

  1. 先查**拼音首字母**（PGD - 全局页目录）
  2. 再查**韵母**（PUD - 上层页目录）
  3. 再查**具体页码**（PMD - 中间页目录）
  4. 最后查**那个字**（PTE - 页表项）

这样，如果一大块内存你根本没用到，那对应的中间目录就不需要创建。大大节省了“对照本”占用的内存。

（目前的 Linux 通常是 4 级或 5 级页表）。

四、 页表与性能的核心冲突：TLB

这就引出了你 DPDK 学习中最重要的概念。

问题：

查多级页表虽然省内存，但是太慢了！

每次读一个数据，MMU 都要去内存里查 4 次目录（PGD->PUD->PMD->PTE），最后才能拿到物理地址去读数据。

这意味着：访问一次内存，实际变成了访问 5 次内存！性能暴跌。

救星：TLB (Translation Lookaside Buffer)

CPU 里有个专门的高速缓存叫 TLB。

- 它是一张**“小抄”**。

- 当你查过一次“虚拟页 101 -> 物理页 0x9999”后，TLB 会把这个结果**记下来**。
- 下次再访问虚拟页 101，MMU 直接看 TLB，**瞬间**拿到物理地址，不用再去查那 4 级页表了。

五、 回到 DPDK：为什么要用 Hugepages（大页）？

理解了页表和 TLB，你就懂了 DPDK 的良苦用心。

- 普通页表 (4KB)：

  如果你的 DPDK 程序占用了 1GB 内存。

  1GB / 4KB = 262,144 个页。

  这意味着需要 26 万个页表项映射关系。

  但是，CPU 的 TLB 只有几千个位置（比如 1024 个）。

  后果：TLB 根本存不下！你的程序运行时，TLB 会频繁失效（TLB Miss），CPU 被迫不停地去查 4 级页表，性能极其低下。

- 大页 (Hugepages, 2MB 或 1GB)：

  如果你开启了 1GB 的大页（Hugepages）。

  1GB / 1GB = 1 个页。

  后果：只需要 1 个 TLB 表项就能覆盖你所有的内存！TLB 命中率 100%，CPU 再也不用去查页表了。

总结

1. **页表**是虚拟地址到物理地址的**字典**。
2. **MMU** 是查字典的**硬件**。
3. **TLB** 是字典查询结果的**缓存**（非常快但容量小）。
4. **普通页 (4KB)** 会导致 TLB 存不下，频繁查表导致慢。
5. **DPDK 用大页 (2MB/1GB)** 就是为了减少页的数量，让 TLB 能装下所有的映射关系，从而实现**极致的内存访问速度**。

（4）全连接队列和半连接队列

在 Linux 内核中，当一个 socket 调用 `listen` 变成监听状态后，内核会为它维护两个队列：

1. **半连接队列 (SYN Queue)**：
   - 存放状态为 `SYN_RCVD` 的连接。
   - 也就是客户端发来了 `SYN` 包，服务器回了 `SYN+ACK`，但还没收到客户端的最后一次 `ACK`。
   - 这些还在“握手过程中”的连接放在这里。

2. **全连接队列 (Accept Queue)**：
    - 存放状态为 `ESTABLISHED` 的连接。
    - 也就是**三次握手已经完全成功**，连接已经建立好了，正坐在那里等着你的程序调用 `accept()` 把它领走。
    - **你代码里的 `listen(sockfd, 10)` 中的 `10`，控制的就是这个队列的长度。**

（5）socket与ip、port的关系

socket在app层就是个fd【app通过fd找到内核中给该socket分配的缓冲区】；

在内核层是一个缓冲区【内核通过ip + port确定数据存在哪个缓冲区】；

```
// accept前的准备工作
int init_server(unsigned short port) {

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);// ipv4\tcp
    // 造表
    struct sockaddr_in serveraddr;
    memset(&serveraddr, 0, sizeof(struct sockaddr_in));
    // 填表
    serveraddr.sin_family = AF_INET;        //ipv4
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); // 本机的任意IP
    serveraddr.sin_port = htons(port);      //端口号

    // 给光杆司令sockfd一个身份：可以处理ip+prot的信息
    // 凡是给本ip+port的信息内核都会放到sockfd这个缓冲区中
    // “认亲，告诉内核sockfd的亲戚是谁”
    if (-1 == bind(sockfd, (struct sockaddr*)&serveraddr,
sizeof(struct sockaddr))) {
        perror("bind");
        return -1;
    }
    // 监听这个socket(ip + port);来了连接就进行三次握手
    // 三次握手成功但没被accept的最大数量为10
    listen(sockfd, 10);

    return sockfd;
}
```

（6）socket缓冲区大小

最小：4KB；默认：16KB；最大：4MB

```
liyumin@liyumin:~$ sysctl net.ipv4.tcp_wmem
net.ipv4.tcp_wmem = 4096    16384    4194304
```

所以我们去写2048个byte = 2KB；很easy

(7) pthread

```
#include <pthread.h>

int pthread_create(pthread_t *thread, // tid，线程的唯一ID号
                   const pthread_attr_t *attr, // 线程的属性：栈大小，调度优先级、是都分离等
                   void *(*start_routine) (void *), // 现成函数的指针
                   void *arg);                      // 传递给线程函数的参数
```