计算机网络面试题

> 基于简历：扎实的计算机网络（TCP/IP协议栈）基础，项目中使用epoll实现网络编程

一、TCP/IP协议栈

Q1: OSI七层模型和TCP/IP四层模型的对应关系？

OSI七层模型：

1. 应用层（Application）
2. 表示层（Presentation）
3. 会话层（Session）
4. 传输层（Transport）
5. 网络层（Network）
6. 数据链路层（Data Link）
7. 物理层（Physical）

TCP/IP四层模型：

1. 应用层（HTTP、FTP、DNS等）
2. 传输层（TCP、UDP）
3. 网络层（IP、ICMP、ARP）
4. 网络接口层（以太网、WiFi）

对应关系：

- 应用层 = OSI的应用层+表示层+会话层
- 传输层 = OSI的传输层
- 网络层 = OSI的网络层
- 网络接口层 = OSI的数据链路层+物理层

与你项目的联系： "在文件传输系统中，应用层使用HTTP协议，传输层使用TCP保证可靠性，网络层使用IP寻址路由。"

Q2: TCP和UDP的区别？各自的应用场景？

特性	TCP	UDP
连接	面向连接	无连接
可靠性	可靠传输	不可靠
顺序	保证顺序	不保证
速度	较慢	较快
头部开销	20字节	8字节

特性	TCP	UDP
流量控制	有（滑动窗口）	无
拥塞控制	有	无

TCP应用场景：

- 文件传输（FTP、HTTP）
- 邮件（SMTP）
- 远程登录（SSH）
- **你的项目：文件传输需要可靠性**

UDP应用场景：

- 视频直播（丢几帧无所谓）
- DNS查询（单个数据包）
- 游戏（实时性重要）

二、TCP核心机制

Q3: TCP三次握手的详细过程？为什么需要三次？

三次握手过程：

```
Client                    Server
  |                         |
  |-----SYN, seq=x---------->|  第一次握手
  |                         |  (SYN_SENT)
  |                         |
  |<---SYN+ACK, seq=y,-------|  第二次握手
  |    ack=x+1              |  (SYN_RCVD)
  |                         |
  |-----ACK, ack=y+1-------->|  第三次握手
  |                         |  (ESTABLISHED)
  |                         |
```

详细说明：

1. **第一次握手**：客户端发送SYN，序列号seq=x
    - 客户端状态：SYN_SENT
    - 证明：客户端发送能力OK
2. **第二次握手**：服务器发送SYN+ACK，seq=y, ack=x+1
    - 服务器状态：SYN_RCVD
    - 证明：服务器接收和发送能力OK
3. **第三次握手**：客户端发送ACK，ack=y+1

- 双方状态：ESTABLISHED
- 证明：客户端接收能力OK

为什么需要三次？

❌ **两次不够：** 假设网络延迟，客户端重传了SYN。第一个SYN延迟后到达，服务器会建立两个连接，浪费资源。三次握手时，客户端可以拒绝旧的SYN。

✅ **三次刚好：**

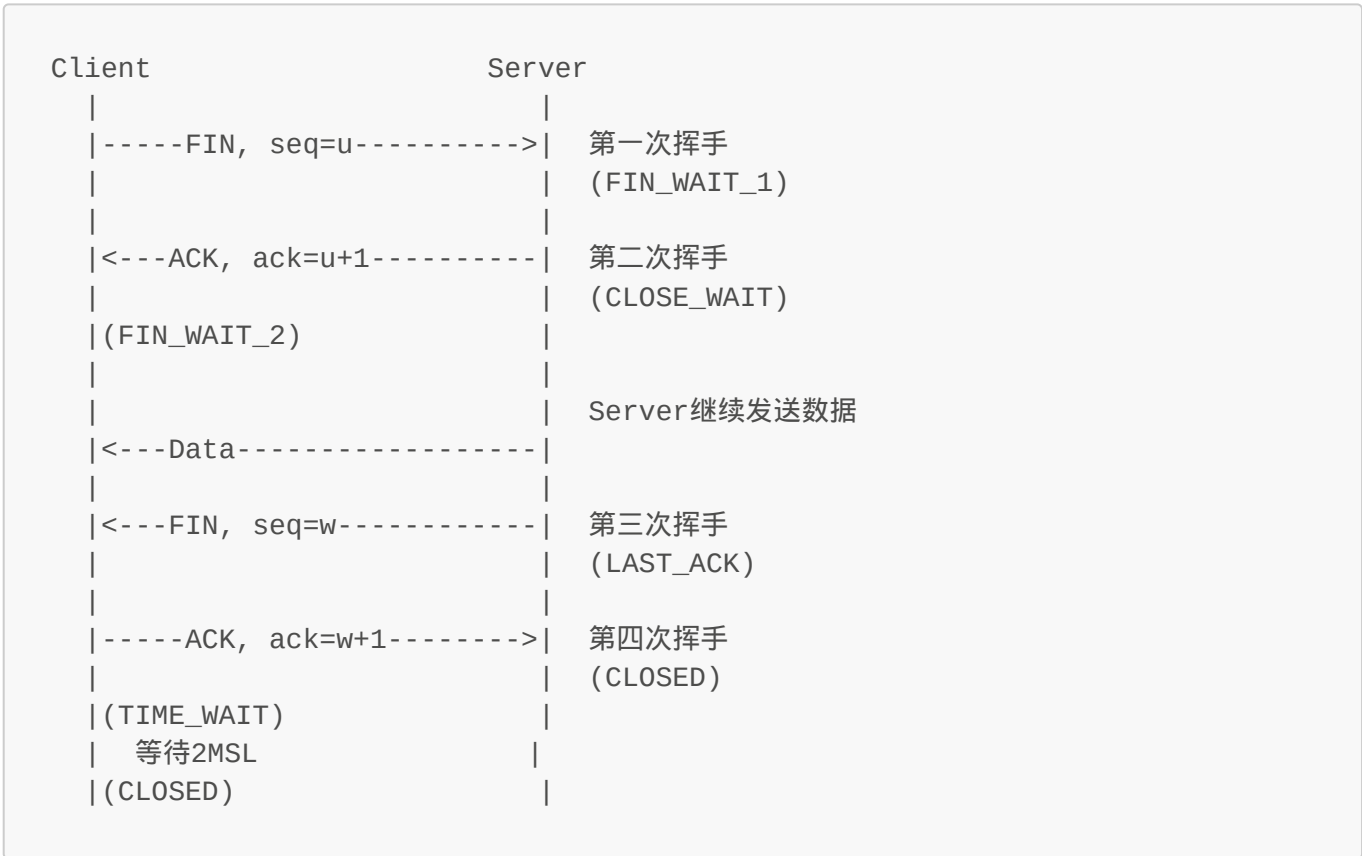
- 确认双方的发送和接收能力
- 同步双方的初始序列号
- 防止旧的重复连接

面试追问： "第三次握手失败会怎样？"

答案： 服务器会重传SYN+ACK，超时后关闭连接。客户端认为连接已建立，发送数据时收到RST。

Q4: TCP四次挥手的详细过程？为什么需要四次？

四次挥手过程：



为什么需要四次？

TCP是**全双工**通信，两个方向需要分别关闭：

1. 第一次：客户端告诉服务器"我不再发送数据了"
2. 第二次：服务器确认"我知道你不发了"
3. 第三次：服务器告诉客户端"我也不发了"（可能还有数据在发送）

4. 第四次：客户端确认"我知道你不发了"

TIME_WAIT状态：

- 持续时间：2MSL（Maximum Segment Lifetime，一般2分钟）
- 作用：
    1. 确保最后的ACK能到达服务器
    2. 让旧的数据包在网络中消失

与你项目的联系： "文件传输系统中，客户端下载完文件后主动关闭连接。服务器端需要处理大量TIME_WAIT状态的连接，可以通过SO_REUSEADDR选项复用端口。"

Q5: TCP如何保证可靠传输?

五大机制：

1. 序列号和确认应答（ACK）

```
发送方: [1][2][3][4][5]
接收方:  ACK1  ACK2  ACK3  ACK4  ACK5
```

- 每个字节都有序列号
- 接收方返回ACK确认收到

2. 超时重传

```
发送方: [3] ----X  (丢失)
        等待超时...
        [3] ----→  (重传)
接收方:        ACK3 ←----
```

- 发送后启动定时器
- 超时未收到ACK，重传

3. 流量控制（滑动窗口）

```
接收窗口：[     可接收空间     ]
发送窗口：[已发送未确认][可发送]
```

- 接收方告诉发送方自己的缓冲区大小
- 发送方控制发送速率

4. 拥塞控制

- 慢启动
- 拥塞避免

- 快速重传
- 快速恢复

5. 校验和

- TCP头部和数据计算校验和
- 接收方验证，发现错误则丢弃

与你项目的联系： "文件传输系统使用TCP，数据的可靠性由TCP保证。应用层只需要处理业务逻辑，不用担心丢包重传。"

Q6: TCP流量控制和拥塞控制的区别?

流量控制（Flow Control）：

- **目的**：防止发送方发送过快，接收方来不及处理
- **机制**：滑动窗口
- **控制者**：接收方通过窗口大小控制

拥塞控制（Congestion Control）：

- **目的**：防止网络拥塞，整体性能下降
- **机制**：慢启动、拥塞避免、快速重传、快速恢复
- **控制者**：发送方感知网络状况主动控制

形象比喻：

- 流量控制：你家水管太细，让自来水厂少送点水（点对点）
- 拥塞控制：整个城市水管堵塞，自来水厂主动减少供水（全局）

Q7: TCP拥塞控制的四个算法详细说明?

1. 慢启动（Slow Start）

```
拥塞窗口(cwnd):
 1 → 2 → 4 → 8 → 16 → ...
```

- 初始cwnd=1 MSS
- 每收到一个ACK，cwnd翻倍（指数增长）
- 达到慢启动阈值(ssthresh)后，进入拥塞避免

2. 拥塞避免（Congestion Avoidance）

```
拥塞窗口(cwnd):
16 → 17 → 18 → 19 → ...
```

- 每个RTT，cwnd加1（线性增长）

- 继续直到发生拥塞

3. 快速重传（Fast Retransmit）

```
发送: 1 2 3 4 5
接收: ACK1 ACK2 ACK2 ACK2 ACK2
                ↑收到3个重复ACK
发送: 立即重传3，不等超时
```

- 收到3个重复ACK，立即重传
- 不等超时，提高效率

4. 快速恢复（Fast Recovery）

- 发生快速重传后：
  - ssthresh = cwnd / 2
  - cwnd = ssthresh
  - 进入拥塞避免（不回到慢启动）

状态转换图：

```
[慢启动] --达到ssthresh--> [拥塞避免]
    ↑                          ↓
    |                       发生拥塞
    |                          ↓
    ←---超时---  [快速恢复] ←--3个重复ACK
```

Q8: 粘包问题是什么？如何解决？

粘包问题： TCP是字节流协议，没有消息边界。

示例：

```
发送方:  Send("Hello") Send("World")
接收方可能收到:
  - "HelloWorld"    (粘包)
  - "Hel" + "loWorld"  (拆包)
```

解决方案：

1. 固定长度

```
// 每条消息10字节
char buf[10];
recv(fd, buf, 10, 0);
```

2. 分隔符

```
// 以'\n'分隔
std::string readLine(int fd) {
    std::string line;
    char c;
    while (recv(fd, &c, 1, 0) > 0) {
        if (c == '\n') break;
        line += c;
    }
    return line;
}
```

3. 长度前缀（推荐）

```
// 消息格式: [4字节长度][数据]
struct Message {
    uint32_t length;
    char data[MAX_SIZE];
};

// 发送
uint32_t len = htonl(dataLen);
send(fd, &len, 4, 0);
send(fd, data, dataLen, 0);

// 接收
uint32_t len;
recv(fd, &len, 4, 0);
len = ntohl(len);
char* buf = new char[len];
recv(fd, buf, len, 0);
```

与你项目的联系： "文件传输系统使用HTTP协议，HTTP头部用`\r\n\r\n`分隔，`Content-Length`字段指定body长度，解决粘包问题。"

三、HTTP协议

Q9: HTTP请求和响应的格式?

HTTP请求格式：

```
GET /index.html HTTP/1.1\r\n
Host: www.example.com\r\n
User-Agent: Mozilla/5.0\r\n
```

```
Accept: text/html\r\n
\r\n
[请求体]
```

HTTP响应格式：

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
Content-Length: 1234\r\n
\r\n
[响应体]
```

关键点：

- 请求行/状态行
- 头部字段
- 空行（`\r\n\r\n`）分隔头部和body
- 消息体

Q10: HTTP状态码分类及常见状态码？

1xx（信息）

- 100 Continue

2xx（成功）

- 200 OK
- 201 Created
- 204 No Content

3xx（重定向）

- 301 Moved Permanently（永久重定向）
- 302 Found（临时重定向）
- 304 Not Modified（缓存有效）

4xx（客户端错误）

- 400 Bad Request
- 401 Unauthorized（未认证）
- 403 Forbidden（无权限）
- 404 Not Found

5xx（服务器错误）

- 500 Internal Server Error
- 502 Bad Gateway
- 503 Service Unavailable

Q11: HTTP/1.0、HTTP/1.1、HTTP/2的区别?

HTTP/1.0：

- 短连接：每次请求都要建立TCP连接
- 无Host头（不支持虚拟主机）

HTTP/1.1：

- 长连接：`Connection: keep-alive`，复用TCP连接
- 管道化：可以连续发送多个请求
- Host头：支持虚拟主机
- 缓存控制：`Cache-Control`

HTTP/2：

- 二进制协议（HTTP/1.x是文本）
- 多路复用：一个TCP连接并行处理多个请求
- 头部压缩：HPACK算法
- 服务器推送：主动推送资源

性能对比：

```
HTTP/1.0:  [请求1] → 等待 → [请求2] → 等待
HTTP/1.1:  [请求1] [请求2] [请求3] (管道化，但有队头阻塞)
HTTP/2:    [请求1][请求2][请求3] (并行，无队头阻塞)
```

Q12: GET和POST的区别?

表面区别：

特性	GET	POST
参数位置	URL	Body
缓存	可以	不可以
历史记录	保留	不保留
长度限制	有（URL限制）	无
安全性	参数可见	参数在body
幂等性	是	否

本质区别：

- GET：获取资源（幂等、安全）
- POST：提交数据（非幂等）

幂等性：

- 幂等：多次请求结果相同（GET、PUT、DELETE）
- 非幂等：多次请求结果不同（POST）

四、Socket编程

Q13: socket编程的基本流程?

服务器端：

```
// 1. 创建socket
int listenfd = socket(AF_INET, SOCK_STREAM, 0);

// 2. 绑定地址
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
addr.sin_addr.s_addr = INADDR_ANY;
bind(listenfd, (struct sockaddr*)&addr, sizeof(addr));

// 3. 监听
listen(listenfd, 128);

// 4. 接受连接
int connfd = accept(listenfd, NULL, NULL);

// 5. 读写数据
char buf[1024];
int n = read(connfd, buf, sizeof(buf));
write(connfd, "Hello", 5);

// 6. 关闭连接
close(connfd);
close(listenfd);
```

客户端：

```
// 1. 创建socket
int sockfd = socket(AF_INET, SOCK_STREAM, 0);

// 2. 连接服务器
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr);
connect(sockfd, (struct sockaddr*)&addr, sizeof(addr));

// 3. 读写数据
```

```
write(sockfd, "Hello", 5);
char buf[1024];
read(sockfd, buf, sizeof(buf));

// 4. 关闭连接
close(sockfd);
```

Q14: 阻塞I/O和非阻塞I/O的区别?

阻塞I/O：

```
int n = read(fd, buf, sizeof(buf));
// read会阻塞，直到有数据到来
```

- 调用会阻塞，直到操作完成
- 简单，但效率低

非阻塞I/O：

```
fcntl(fd, F_SETFL, O_NONBLOCK);
int n = read(fd, buf, sizeof(buf));
if (n < 0 && errno == EAGAIN) {
    // 没有数据，稍后再试
}
```

- 立即返回，不阻塞
- 需要轮询或配合I/O多路复用

与你项目的联系： "文件传输系统使用epoll配合非阻塞I/O，避免单个连接阻塞整个服务器。"

Q15: select、poll、epoll的区别?

select：

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(fd, &readfds);
select(maxfd+1, &readfds, NULL, NULL, &timeout);
if (FD_ISSET(fd, &readfds)) {
    // fd可读
}
```

- 最大文件描述符限制（一般1024）
- 每次调用需要拷贝fd_set

- 时间复杂度O(n)

poll：

```
struct pollfd fds[MAX_FDS];
fds[0].fd = fd;
fds[0].events = POLLIN;
poll(fds, nfds, timeout);
if (fds[0].revents & POLLIN) {
    // fd可读
}
```

- 无最大文件描述符限制
- 仍需要遍历所有fd
- 时间复杂度O(n)

epoll：

```
int epfd = epoll_create(1);
struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.fd = fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);

struct epoll_event events[MAX_EVENTS];
int n = epoll_wait(epfd, events, MAX_EVENTS, -1);
for (int i = 0; i < n; i++) {
    // events[i].data.fd 可读
}
```

- 无最大文件描述符限制
- 只返回就绪的fd
- 时间复杂度O(1)
- 支持ET和LT模式

对比：

特性	select	poll	epoll
性能	O(n)	O(n)	O(1)
fd限制	1024	无限制	无限制
跨平台	是	是	Linux only

与你项目的联系： "文件传输系统使用epoll，在高并发场景下性能远优于select/poll。"

Q16: epoll的LT和ET模式详细区别?

LT（Level Triggered，水平触发）：

```
缓冲区：[data data data]
epoll_wait: 通知 → 读一半 → epoll_wait: 继续通知
```

- 只要缓冲区有数据就通知
- 类似电平触发
- 不容易遗漏数据

ET（Edge Triggered，边缘触发）：

```
缓冲区：[data data data]
epoll_wait: 通知 → 读一半 → epoll_wait: 不通知（除非新数据到来）
```

- 只在状态变化时通知一次
- 类似边沿触发
- 必须一次性读完所有数据

ET模式代码示例：

```
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);  // 不加EPOLLET就是LT

// ET模式，必须循环读取
while (true) {
    int n = read(fd, buf, sizeof(buf));
    if (n < 0) {
        if (errno == EAGAIN) break;  // 读完了
        // 错误处理
    }
    if (n == 0) break;  // 连接关闭
    // 处理数据
}
```

选择建议：

- 初学者：LT模式，简单不易出错
- 性能要求高：ET模式，减少系统调用

与你项目的联系： "我的项目使用LT模式，因为实现简单，性能瓶颈不在epoll通知次数。"

五、综合应用

Q17: 从浏览器输入URL到页面显示，经历了哪些过程？

完整流程：

1. DNS解析

```
www.example.com → 93.184.216.34
```

- 浏览器缓存 → 系统缓存 → 路由器缓存 → ISP DNS → 根DNS

2. TCP连接（三次握手）

```
Client → SYN → Server
Client ← SYN+ACK ← Server
Client → ACK → Server
```

3. 发送HTTP请求

```
GET /index.html HTTP/1.1
Host: www.example.com
...
```

4. 服务器处理请求

- 解析请求
- 查找资源
- 生成响应

5. 返回HTTP响应

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234
...
[HTML内容]
```

6. 浏览器渲染

- 解析HTML构建DOM树
- 解析CSS构建CSSOM树
- 合并成渲染树
- 布局计算位置
- 绘制到屏幕

7. 连接关闭（四次挥手）

Q18: TIME_WAIT状态过多如何解决?

问题： 主动关闭连接的一方会进入TIME_WAIT，持续2MSL（约2分钟）。

危害：

- 占用端口资源
- 高并发下端口耗尽

解决方案：

1. 调整内核参数

```
# 减少TIME_WAIT时间
net.ipv4.tcp_fin_timeout = 30

# 允许TIME_WAIT重用
net.ipv4.tcp_tw_reuse = 1

# 快速回收
net.ipv4.tcp_tw_recycle = 1  # (不建议，可能导致问题)
```

2. SO_REUSEADDR选项

```
int reuse = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
```

3. 让客户端主动关闭

- 服务器不主动关闭连接
- 由客户端关闭，TIME_WAIT在客户端

与你项目的联系： "文件传输系统中，服务器使用SO_REUSEADDR选项，重启时可以立即绑定端口。"

六、面试技巧

回答问题的思路

1. 分层回答

```
问题：TCP如何保证可靠性？
回答：
- 首先列举：序列号、ACK、超时重传、流量控制、拥塞控制
- 然后展开：选2-3个重点讲细节
- 最后联系：在我的项目中...
```

2. 画图说明

- 三次握手、四次挥手：画时序图
- 拥塞控制：画cwnd变化图
- epoll：画架构图

3. 举例说明

- 不要只说概念
- 结合实际场景
- 联系自己的项目

常见追问

Q: "三次握手"后追问 → "第三次握手失败会怎样？" → "为什么不是两次或四次？" → "SYN洪水攻击如何防御？"

Q: "epoll"后追问 → "LT和ET的区别？" → "epoll底层数据结构？" → "为什么epoll比select快？"

加分回答

✅ "在我的文件传输项目中..."（联系实践） ✅ "可以画个图说明一下"（主动展示） ✅ "这有几种解决方案..."（展现思考） ✅ "这样做有个问题..."（指出局限）

减分回答

❌ "这个我背过，是..."（死记硬背） ❌ "就是...反正就是那样"（模糊不清） ❌ "我没用过"（缺少实践）

快速复习清单

必须掌握

- ☐ TCP三次握手、四次挥手
- ☐ TCP可靠性机制（5个）
- ☐ 流量控制vs拥塞控制
- ☐ select/poll/epoll区别
- ☐ HTTP状态码
- ☐ GET vs POST

重点理解

- ☐ TIME_WAIT状态
- ☐ 粘包问题
- ☐ 拥塞控制算法
- ☐ epoll LT/ET模式

结合项目

- ☐ 为什么用TCP?
- ☐ 为什么用epoll?
- ☐ 如何处理并发连接?
- ☐ 如何解决粘包?

记住：理论 + 实践 + 项目经验 = 高分回答