C++ 网络编程

从 Socket 到高性能服务器

核心内容

TCP/IP 协议栈 · Socket 编程 · I/O 模型

epoll 详解 · Reactor 模式 · 高并发优化

网络库设计 · 实战案例 · 面试题解析

面试准备 · 项目实战 · 技术进阶

作者：Aweo

2025 年 10 月

Contents

1 TCP/IP 协议基础

1.1 TCP/IP 四层模型

网络编程的基础是理解 TCP/IP 协议栈。TCP/IP 分为四层模型：

应用层：为应用程序提供网络服务（HTTP、FTP、DNS、SMTP 等）

传输层：提供端到端的数据传输服务（TCP、UDP）

网络层：负责数据包的路由和转发（IP、ICMP、ARP）

链路层：负责物理网络的数据传输（Ethernet、WiFi）

1.2 TCP vs UDP

| 特性 | TCP | UDP | |——|——|——| | 连接性 | 面向连接 | 无连接 | | 可靠性 | 可靠（确认、重传） | 不可靠 | | 顺序性 | 保证顺序 | 不保证顺序 | | 速度 | 较慢（开销大） | 较快（开销小） | | 应用场景 | 文件传输、HTTP、邮件 | 视频流、游戏、DNS |

1.3 TCP 三次握手（连接建立）

为什么需要三次握手？

- 确认双方的发送和接收能力都正常
- 同步序列号（seq）和确认号（ack）
- 防止已失效的连接请求突然又传到服务器

握手过程：

```C++
// 第一次握手：客户端发送SYN
Client -> Server: SYN=1, seq=x

// 第二次握手：服务器回复SYN+ACK
Server -> Client: SYN=1, ACK=1, seq=y, ack=x+1

// 第三次握手：客户端发送ACK
Client -> Server: ACK=1, seq=x+1, ack=y+1

// 连接建立，可以传输数据
```

面试重点：

1. 为什么不是两次握手？
   - 两次握手无法确认客户端的接收能力
   - 无法防止旧的连接请求突然到达服务器
2. 第三次握手可以携带数据吗？
   - 可以！第三次握手时连接已建立，客户端可以发送数据
   - 前两次不能携带数据（防止 SYN flood 攻击）

1.4 TCP 四次挥手（连接断开）

为什么需要四次挥手？

- TCP 是全双工通信，双方都需要关闭连接
- 一方关闭发送不代表另一方也关闭发送

挥手过程：

```C++
// 第一次挥手：客户端发送FIN
Client -> Server: FIN=1, seq=u

```

```
// 第二次挥手：服务器回复ACK
Server -> Client: ACK=1, ack=u+1

// 此时客户端->服务器方向关闭，但服务器->客户端方向仍可发送数据

// 第三次挥手：服务器发送FIN
Server -> Client: FIN=1, seq=w

// 第四次挥手：客户端回复ACK
Client -> Server: ACK=1, ack=w+1

// 客户端进入TIME_WAIT状态（2MSL）
// 服务器收到ACK后关闭连接
```

面试重点：

1. 为什么不是三次挥手？
   - 服务器收到 FIN 后，可能还有数据要发送
   - ACK 和 FIN 必须分开发送
2. 什么是 **TIME_WAIT** 状态？为什么需要 **2MSL**？
   - 主动关闭方进入 TIME_WAIT 状态
   - 2MSL（Maximum Segment Lifetime）= 2 × 报文最大生存时间
   - 原因 1：确保最后一个 ACK 能到达对方（如果丢失，对方会重传 FIN）
   - 原因 2：确保旧连接的所有报文都消失
3. **TIME_WAIT** 过多怎么办？
   - 调整 `net.ipv4.tcp_tw_reuse` 和 `net.ipv4.tcp_tw_recycle`
   - 使用 `SO_REUSEADDR` 套接字选项
   - 让客户端主动关闭连接（TIME_WAIT 在客户端）

1.5 TCP 状态转换图

常见状态：

- `LISTEN`：服务器等待连接
- `SYN_SENT`：客户端发送 SYN 后等待
- `SYN_RCVD`：服务器收到 SYN，发送 SYN+ACK 后等待
- `ESTABLISHED`：连接建立，可以传输数据
- `FIN_WAIT_1`：主动关闭方发送 FIN 后
- `FIN_WAIT_2`：收到对方 ACK 后
- `TIME_WAIT`：收到对方 FIN 并发送 ACK 后（2MSL）
- `CLOSE_WAIT`：被动关闭方收到 FIN 后
- `LAST_ACK`：被动关闭方发送 FIN 后

面试常考：`CLOSE_WAIT` 过多的原因？

```C++
// 问题代码：服务器收到客户端FIN后，没有调用close()
void handle_client(int fd) {
    char buf[1024];
    int n = read(fd, buf, sizeof(buf));
    if (n == 0) {
        // 客户端关闭连接
        // 如果这里不调用close(fd)，服务器会一直处于CLOSE_WAIT状态
        return;  // ❌ 错误：没有关闭fd
    }
```

```
    // ...
}

// 正确做法：
void handle_client_correct(int fd) {
    char buf[1024];
    int n = read(fd, buf, sizeof(buf));
    if (n == 0) {
        close(fd);  // ✅ 正确：关闭文件描述符
        return;
    }
    // ...
}
```

1.6 TCP 可靠性保证机制

1.6.1 1. 序列号和确认号

每个字节都有序列号，接收方通过 ACK 确认收到的数据。

```
// 发送1000字节数据
Client: seq=100, len=1000, data=[100-1099]
Server: ACK, ack=1100  // 确认收到，期望下一个字节是1100
```

1.6.2 2. 超时重传

发送方设置重传定时器，超时未收到 ACK 则重传。

```
// 简化的重传机制
struct Packet {
    uint32_t seq;
    char data[1024];
    std::chrono::time_point<std::chrono::steady_clock> send_time;
};

void send_with_retransmit(int sockfd, Packet& pkt) {
    pkt.send_time = std::chrono::steady_clock::now();
    send(sockfd, &pkt, sizeof(pkt), 0);

    // 设置超时重传（实际TCP使用更复杂的RTO算法）
    auto timeout = std::chrono::milliseconds(200);
    while (!received_ack(pkt.seq)) {
        auto now = std::chrono::steady_clock::now();
        if (now - pkt.send_time > timeout) {
            // 超时重传
            pkt.send_time = now;
            send(sockfd, &pkt, sizeof(pkt), 0);
            timeout *= 2;  // 指数退避
        }
    }
}
```

1.6.3 3. 滑动窗口（流量控制）

接收方通过窗口大小限制发送方的发送速率。

```
// TCP头部包含窗口大小字段
struct TCPHeader {
```

```
    uint16_t window_size;  // 接收方剩余缓冲区大小
    // ...
};

// 发送方维护滑动窗口
class SendWindow {
    uint32_t base;       // 最早未确认的字节
    uint32_t next_seq;   // 下一个要发送的字节
    uint32_t window_size;// 接收方通告的窗口大小

public:
    bool can_send() {
        return next_seq - base < window_size;
    }

    void send_data(int sockfd, const char* data, size_t len) {
        if (can_send()) {
            send(sockfd, data, len, 0);
            next_seq += len;
        }
    }

    void on_ack(uint32_t ack_seq) {
        base = ack_seq;  // 滑动窗口向前移动
    }
};
```

1.6.4 4. 拥塞控制

通过慢启动、拥塞避免、快速重传、快速恢复等算法控制网络拥塞。

慢启动：拥塞窗口从 1 开始指数增长

拥塞避免：窗口达到阈值后线性增长

快速重传：收到 3 个重复 ACK 立即重传

快速恢复：快速重传后进入拥塞避免而非慢启动

2 Socket 编程基础

2.1 Socket 是什么？

Socket（套接字）是应用层与传输层之间的接口，提供网络通信的编程接口。

Socket 的本质：

- 在内核中是一个文件描述符（fd）
- 关联了五元组：{协议, 本地 IP, 本地端口, 远程 IP, 远程端口}
- 维护了接收/发送缓冲区

2.2 Socket 基本 API

2.2.1 服务器端 API

```cpp
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

// 1. 创建socket
int socket(int domain, int type, int protocol);
// domain: AF_INET(IPv4), AF_INET6(IPv6)
// type: SOCK_STREAM(TCP), SOCK_DGRAM(UDP)
// protocol: 通常为0
// 返回：socket文件描述符，失败返回-1

// 2. 绑定地址
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
// 将socket绑定到指定的IP地址和端口

// 3. 监听连接
int listen(int sockfd, int backlog);
// backlog: 全连接队列的最大长度

// 4. 接受连接
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
// 从全连接队列中取出一个连接
// 返回：新的socket fd用于与客户端通信

// 5. 发送/接收数据
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);

// 6. 关闭连接
int close(int sockfd);
```

2.2.2 客户端 API

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

// 2. 连接服务器
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

// 3. 发送/接收数据
send(sockfd, buf, len, 0);
```

```
recv(sockfd, buf, len, 0);

// 4. 关闭连接
close(sockfd);
```

2.3 TCP 服务器完整示例

2.3.1 简单的 Echo 服务器（单线程阻塞版本）

C++

```cpp
#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    // 1. 创建监听socket
    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd < 0) {
        perror("socket failed");
        return 1;
    }

    // 设置SO_REUSEADDR，允许地址重用（解决TIME_WAIT问题）
    int opt = 1;
    setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    // 2. 绑定地址
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;  // 0.0.0.0
    server_addr.sin_port = htons(8080);         // 端口8080

    if (bind(listen_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind failed");
        close(listen_fd);
        return 1;
    }

    // 3. 开始监听
    if (listen(listen_fd, 128) < 0) {
        perror("listen failed");
        close(listen_fd);
        return 1;
    }

    std::cout << "Server listening on port 8080..." << std::endl;

    // 4. 循环接受客户端连接
    while (true) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);

```

```
47          // 接受连接（阻塞等待）
48          int client_fd = accept(listen_fd, (struct sockaddr*)&client_addr, &client_len);
49          if (client_fd < 0) {
50              perror("accept failed");
51              continue;
52          }
53
54          char client_ip[INET_ADDRSTRLEN];
55          inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, sizeof(client_ip));
56          std::cout << "Client connected: " << client_ip
57                    << ":" << ntohs(client_addr.sin_port) << std::endl;
58
59          // 5. 处理客户端请求（Echo）
60          char buffer[1024];
61          while (true) {
62              memset(buffer, 0, sizeof(buffer));
63              ssize_t n = recv(client_fd, buffer, sizeof(buffer) - 1, 0);
64
65              if (n <= 0) {
66                  // n == 0: 客户端关闭连接
67                  // n < 0: 读取错误
68                  if (n == 0) {
69                      std::cout << "Client disconnected" << std::endl;
70                  } else {
71                      perror("recv failed");
72                  }
73                  break;
74              }
75
76              std::cout << "Received: " << buffer;
77
78              // Echo回客户端
79              send(client_fd, buffer, n, 0);
80          }
81
82          // 6. 关闭客户端连接
83          close(client_fd);
84      }
85
86      close(listen_fd);
87      return 0;
88  }
```

2.3.2 TCP 客户端示例

C++

```
1   #include <iostream>
2   #include <cstring>
3   #include <sys/socket.h>
4   #include <netinet/in.h>
5   #include <arpa/inet.h>
6   #include <unistd.h>
7
8   int main() {
9       // 1. 创建socket
10      int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
11      if (sockfd < 0) {
12          perror("socket failed");
13          return 1;
14      }
15
16      // 2. 设置服务器地址
17      struct sockaddr_in server_addr;
18      memset(&server_addr, 0, sizeof(server_addr));
19      server_addr.sin_family = AF_INET;
20      server_addr.sin_port = htons(8080);
21
22      // 将IP地址从字符串转换为网络字节序
23      if (inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr) <= 0) {
24          perror("invalid address");
25          close(sockfd);
26          return 1;
27      }
28
29      // 3. 连接服务器
30      if (connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
31          perror("connect failed");
32          close(sockfd);
33          return 1;
34      }
35
36      std::cout << "Connected to server" << std::endl;
37
38      // 4. 发送和接收数据
39      char send_buf[1024];
40      char recv_buf[1024];
41
42      while (std::cin.getline(send_buf, sizeof(send_buf))) {
43          // 发送数据
44          ssize_t n = send(sockfd, send_buf, strlen(send_buf), 0);
45          if (n < 0) {
46              perror("send failed");
47              break;
48          }
49
50          // 接收回复
51          memset(recv_buf, 0, sizeof(recv_buf));
52          n = recv(sockfd, recv_buf, sizeof(recv_buf) - 1, 0);
53          if (n <= 0) {
54              std::cout << "Server closed connection" << std::endl;
55              break;
56          }
57
58          std::cout << "Server reply: " << recv_buf << std::endl;
59      }
60
61      // 5. 关闭连接
62      close(sockfd);
63      return 0;
64  }
```

2.4 重要的 Socket 选项

2.4.1 SO_REUSEADDR

作用：允许端口重用，解决 TIME_WAIT 状态占用端口的问题。

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

使用场景：

- 服务器重启时，旧连接还在 TIME_WAIT 状态
- 没有 SO_REUSEADDR 会导致 bind 失败（Address already in use）

2.4.2 SO_REUSEPORT（Linux 3.9+）

作用：允许多个 socket 绑定到同一个 IP 和端口。

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

使用场景：

- 多进程/多线程服务器，每个进程/线程有自己的 listen socket
- 内核负载均衡，避免惊群问题

2.4.3 TCP_NODELAY

作用：禁用 Nagle 算法，立即发送小包。

```cpp
int opt = 1;
setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt));
```

Nagle 算法：

- 将多个小包合并成一个大包发送，减少网络开销
- 但会增加延迟（等待更多数据）

使用场景：

- 对延迟敏感的应用（游戏、实时通信）
- 发送小而频繁的数据包

2.4.4 SO_KEEPALIVE

作用：启用 TCP keepalive 机制，检测死连接。

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

// 可以进一步配置keepalive参数
int keepidle = 60;    // 空闲60秒后开始发送探测包
int keepinterval = 5; // 探测包间隔5秒
int keepcount = 3;    // 探测3次失败则认为连接断开

setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPIDLE, &keepidle, sizeof(keepidle));
setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPINTVL, &keepinterval, sizeof(keepinterval));
setsockopt(sockfd, IPPROTO_TCP, TCP_KEEPCNT, &keepcount, sizeof(keepcount));
```

使用场景：

- 长连接应用（聊天服务器、推送服务）
- 检测客户端异常断开（断电、拔网线）

2.4.5 SO_RCVBUF / SO_SNDBUF

作用：设置接收/发送缓冲区大小。

```cpp
int rcvbuf = 1024 * 1024;  // 1MB接收缓冲区
int sndbuf = 1024 * 1024;  // 1MB发送缓冲区
setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &rcvbuf, sizeof(rcvbuf));
setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &sndbuf, sizeof(sndbuf));
```

使用场景：

- 高吞吐量应用（文件传输）
- 调整缓冲区大小优化性能

2.5 网络字节序

字节序问题：

- 小端序（Little-Endian）：低字节存储在低地址（x86）
- 大端序（Big-Endian）：高字节存储在低地址（网络字节序）

转换函数：

```cpp
#include <arpa/inet.h>

// 主机字节序 -> 网络字节序
uint32_t htonl(uint32_t hostlong);   // long (32位)
uint16_t htons(uint16_t hostshort);  // short (16位)

// 网络字节序 -> 主机字节序
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

// IP地址转换
// 字符串 -> 网络字节序（新版，推荐）
int inet_pton(int af, const char *src, void *dst);

// 网络字节序 -> 字符串（新版，推荐）
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);

// 旧版函数（不推荐，仅支持IPv4）
in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

使用示例：

```cpp
// 设置端口（必须转换为网络字节序）
server_addr.sin_port = htons(8080);

// 读取端口（转换为主机字节序）
uint16_t port = ntohs(server_addr.sin_port);

// IP地址转换
struct in_addr ip_addr;
inet_pton(AF_INET, "192.168.1.1", &ip_addr);

char ip_str[INET_ADDRSTRLEN];
inet_ntop(AF_INET, &ip_addr, ip_str, sizeof(ip_str));
```

面试重点：为什么需要字节序转换？

- 不同 CPU 架构的字节序可能不同
- 网络协议统一使用大端序（网络字节序）

- 端口号和 IP 地址是多字节数据，必须转换

3 I/O 模型

3.1 阻塞 vs 非阻塞

阻塞 I/O（Blocking I/O）：

```cpp
// 阻塞读取
int sockfd = socket(...);
char buf[1024];
ssize_t n = recv(sockfd, buf, sizeof(buf), 0);  // 阻塞，直到有数据
```

- 调用 recv 时，如果没有数据，进程会阻塞等待
- 简单直观，但无法处理多个连接（一个连接阻塞时，其他连接无法处理）

非阻塞 I/O（Non-blocking I/O）：

```cpp
// 设置为非阻塞模式
#include <fcntl.h>

int sockfd = socket(...);
int flags = fcntl(sockfd, F_GETFL, 0);
fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);

// 非阻塞读取
char buf[1024];
ssize_t n = recv(sockfd, buf, sizeof(buf), 0);
if (n < 0) {
    if (errno == EAGAIN || errno == EWOULDBLOCK) {
        // 没有数据，立即返回（不阻塞）
        std::cout << "No data available" << std::endl;
    } else {
        // 真正的错误
        perror("recv failed");
    }
} else if (n == 0) {
    // 连接关闭
    std::cout << "Connection closed" << std::endl;
} else {
    // 读取到n字节数据
    std::cout << "Read " << n << " bytes" << std::endl;
}
```

- 调用 recv 时，立即返回（不等待）
- 如果没有数据，返回-1，errno 设置为 EAGAIN/EWOULDBLOCK
- 需要循环轮询（busy-wait），浪费 CPU 资源

3.2 Unix 五种 I/O 模型

3.2.1 1. 阻塞 I/O（Blocking I/O）

```cpp
// 伪代码
recvfrom(sockfd, buf, len, ...);
// 阻塞，直到数据到达
// 数据从内核空间复制到用户空间
// 返回
```

特点：

- 进程阻塞，直到数据准备好并复制到用户空间

- 简单但效率低

3.2.2 2. 非阻塞 I/O（Non-blocking I/O）

```cpp
// 伪代码
while (true) {
    int n = recvfrom(sockfd, buf, len, ...);  // 非阻塞
    if (n > 0) break;  // 数据准备好
    if (errno != EAGAIN) break;  // 真正的错误
    // 继续轮询
}
```

特点：
- 不阻塞，但需要循环轮询
- 浪费 CPU 资源

3.2.3 3. I/O 多路复用（I/O Multiplexing）

```cpp
// 伪代码
select/poll/epoll(fds, ...);  // 阻塞，等待任意fd就绪
// 某个fd就绪后返回
recvfrom(sockfd, buf, len, ...);  // 不会阻塞（数据已就绪）
```

特点：
- 可以同时监听多个 fd
- 阻塞在 select/poll/epoll 上，而不是每个 fd 上
- 最常用的高性能服务器模型

3.2.4 4. 信号驱动 I/O（Signal-driven I/O）

```cpp
// 伪代码
sigaction(SIGIO, ...);  // 注册信号处理函数
// 数据就绪时，内核发送SIGIO信号
// 信号处理函数中调用recvfrom()
```

特点：
- 不需要轮询，由内核通知
- 很少使用（信号处理复杂）

3.2.5 5. 异步 I/O（Asynchronous I/O）

```cpp
// 伪代码
aio_read(sockfd, buf, len, ...);  // 立即返回
// 内核负责数据准备和复制
// 完成后通知应用程序（数据已在用户空间）
```

特点：
- 真正的异步：内核负责整个 I/O 过程
- Linux 下需要使用 libaio 或 io_uring
- Windows 的 IOCP（I/O Completion Port）是真正的异步 I/O

3.3 同步 vs 异步

同步 I/O：应用程序负责数据复制（前 4 种模型）
- 阻塞 I/O
- 非阻塞 I/O
- I/O 多路复用
- 信号驱动 I/O

异步 **I/O**：内核负责数据复制（第 5 种模型）

- 异步 I/O（AIO）

面试重点：I/O 多路复用是同步还是异步？

- 同步！虽然 select/epoll 可以监听多个 fd，但数据复制（recvfrom）仍由应用程序完成
- epoll_wait 只是告诉你"数据准备好了"，你还需要自己调用 recv 读取数据

4 I/O 多路复用

4.1 select

4.1.1 select 基本用法

```cpp
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

// nfds: 最大fd + 1
// readfds: 监听读事件的fd集合
// writefds: 监听写事件的fd集合
// exceptfds: 监听异常事件的fd集合
// timeout: 超时时间
// 返回：就绪的fd数量，0表示超时，-1表示错误

// fd_set操作宏
void FD_ZERO(fd_set *set);          // 清空集合
void FD_SET(int fd, fd_set *set);   // 添加fd
void FD_CLR(int fd, fd_set *set);   // 移除fd
int  FD_ISSET(int fd, fd_set *set); // 检查fd是否在集合中
```

4.1.2 select 服务器示例

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <sys/select.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>

int main() {
    // 创建监听socket
    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    int opt = 1;
    setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(8080);

    bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr));
    listen(listen_fd, 128);

    // 维护所有客户端fd
    std::vector<int> client_fds;

    while (true) {
        fd_set readfds;
        FD_ZERO(&readfds);
```

```
31          FD_SET(listen_fd, &readfds);
32
33          int max_fd = listen_fd;
34          for (int fd : client_fds) {
35              FD_SET(fd, &readfds);
36              max_fd = std::max(max_fd, fd);
37          }
38
39          // 等待事件（阻塞）
40          int nready = select(max_fd + 1, &readfds, nullptr, nullptr, nullptr);
41          if (nready < 0) {
42              perror("select failed");
43              break;
44          }
45
46          // 检查监听socket是否就绪（有新连接）
47          if (FD_ISSET(listen_fd, &readfds)) {
48              int client_fd = accept(listen_fd, nullptr, nullptr);
49              if (client_fd >= 0) {
50                  client_fds.push_back(client_fd);
51                  std::cout << "New client: fd=" << client_fd << std::endl;
52              }
53          }
54
55          // 检查每个客户端socket是否就绪（有数据）
56          for (auto it = client_fds.begin(); it != client_fds.end(); ) {
57              int fd = *it;
58              if (FD_ISSET(fd, &readfds)) {
59                  char buf[1024];
60                  ssize_t n = recv(fd, buf, sizeof(buf), 0);
61                  if (n <= 0) {
62                      // 客户端断开
63                      std::cout << "Client disconnected: fd=" << fd << std::endl;
64                      close(fd);
65                      it = client_fds.erase(it);
66                      continue;
67                  }
68                  // Echo
69                  send(fd, buf, n, 0);
70              }
71              ++it;
72          }
73      }
74
75      close(listen_fd);
76      return 0;
77  }
```

4.1.3 select 的限制

1. **fd** 数量限制：默认 1024（FD_SETSIZE）
2. 线性扫描：每次都要遍历所有 fd 检查是否就绪
3. **fd_set** 复制：每次调用 select 都需要将 fd_set 从用户空间复制到内核空间
4. 不可移植性：fd_set 大小固定

4.2 poll

4.2.1 poll 基本用法

```cpp
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
    int   fd;         // 文件描述符
    short events;     // 监听的事件（输入）
    short revents;    // 实际发生的事件（输出）
};

// events 和 revents 可以是以下值的组合：
// POLLIN   : 有数据可读
// POLLOUT  : 可以写数据
// POLLERR  : 发生错误
// POLLHUP  : 挂断
// POLLNVAL : fd未打开
```

4.2.2 poll 服务器示例

```cpp
#include <iostream>
#include <vector>
#include <poll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>

int main() {
    // 创建监听socket
    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    int opt = 1;
    setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(8080);

    bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr));
    listen(listen_fd, 128);

    // poll fd数组
    std::vector<struct pollfd> fds;
    fds.push_back({listen_fd, POLLIN, 0});

    while (true) {
        // 等待事件
        int nready = poll(fds.data(), fds.size(), -1);
        if (nready < 0) {
            perror("poll failed");
            break;
```

```
        }

        // 检查监听socket
        if (fds[0].revents & POLLIN) {
            int client_fd = accept(listen_fd, nullptr, nullptr);
            if (client_fd >= 0) {
                fds.push_back({client_fd, POLLIN, 0});
                std::cout << "New client: fd=" << client_fd << std::endl;
            }
        }

        // 检查客户端socket
        for (size_t i = 1; i < fds.size(); ) {
            if (fds[i].revents & POLLIN) {
                char buf[1024];
                ssize_t n = recv(fds[i].fd, buf, sizeof(buf), 0);
                if (n <= 0) {
                    std::cout << "Client disconnected: fd=" << fds[i].fd << std::endl;
                    close(fds[i].fd);
                    fds.erase(fds.begin() + i);
                    continue;
                }
                send(fds[i].fd, buf, n, 0);
            }
            ++i;
        }
    }

    close(listen_fd);
    return 0;
}
```

4.2.3 poll vs select

| 特性 | select | poll | |——|———|——| | fd 数量限制 | 1024（FD_SETSIZE） | 无限制（受系统限制） | | 数据结构 | fd_set（位图） | pollfd 数组 | | 性能 | O(n)线性扫描 | O(n)线性扫描 | | 可移植性 | POSIX 标准 | POSIX 标准 |

优势：poll 没有 fd 数量限制

劣势：仍然是 O(n)线性扫描

4.3 epoll（Linux 专有）

4.3.1 为什么需要 epoll？

select 和 poll 的性能瓶颈：

1. 每次调用都要复制 **fd** 集合（用户空间→内核空间）
2. 线性扫描所有 **fd**，O(n)复杂度
3. 内核不记录状态，每次都要重新注册

epoll 的优化：

1. 在内核维护红黑树，只需注册一次
2. 使用事件驱动，就绪的 fd 放入就绪队列
3. 只返回就绪的 **fd**，不需要遍历

4.3.2 epoll 基本 API

```
1  #include <sys/epoll.h>
```

C++

```

// 1. 创建epoll实例
int epoll_create1(int flags);
// flags: 0 或 EPOLL_CLOEXEC
// 返回：epoll文件描述符

// 2. 添加/修改/删除监听的fd
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
// op: EPOLL_CTL_ADD（添加）、EPOLL_CTL_MOD（修改）、EPOLL_CTL_DEL（删除）

struct epoll_event {
    uint32_t     events;  // 监听的事件
    epoll_data_t data;    // 用户数据
};

union epoll_data {
    void    *ptr;
    int      fd;
    uint32_t u32;
    uint64_t u64;
};

// events 可以是以下值的组合：
// EPOLLIN   : 可读
// EPOLLOUT  : 可写
// EPOLLERR  : 错误
// EPOLLHUP  : 挂断
// EPOLLET   : 边缘触发模式（Edge Triggered）
// EPOLLONESHOT: 一次性事件

// 3. 等待事件
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
// events: 用于接收就绪事件的数组
// maxevents: events数组的大小
// timeout: 超时时间（毫秒），-1表示永久等待
// 返回：就绪的fd数量
```

4.3.3 epoll 服务器示例（基础版）

```cpp
#include <iostream>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <cstring>
#include <fcntl.h>

// 设置非阻塞
void set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    fcntl(fd, F_SETFL, flags | O_NONBLOCK);
}

int main() {
```

```
    // 1. 创建监听socket
    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    int opt = 1;
    setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
    set_nonblocking(listen_fd);

    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(8080);

    bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr));
    listen(listen_fd, 128);

    // 2. 创建epoll实例
    int epfd = epoll_create1(0);
    if (epfd < 0) {
        perror("epoll_create1 failed");
        return 1;
    }

    // 3. 将监听socket添加到epoll
    struct epoll_event ev;
    ev.events = EPOLLIN;
    ev.data.fd = listen_fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, &ev);

    std::cout << "Server started on port 8080" << std::endl;

    // 4. 事件循环
    const int MAX_EVENTS = 64;
    struct epoll_event events[MAX_EVENTS];

    while (true) {
        int nready = epoll_wait(epfd, events, MAX_EVENTS, -1);
        if (nready < 0) {
            perror("epoll_wait failed");
            break;
        }

        // 5. 处理就绪的fd
        for (int i = 0; i < nready; ++i) {
            int fd = events[i].data.fd;

            if (fd == listen_fd) {
                // 新连接
                while (true) {
                    int client_fd = accept(listen_fd, nullptr, nullptr);
                    if (client_fd < 0) break;  // 没有更多连接

                    set_nonblocking(client_fd);

                    struct epoll_event client_ev;
```

```
                    client_ev.events = EPOLLIN;
                    client_ev.data.fd = client_fd;
                    epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &client_ev);

                    std::cout << "New client: fd=" << client_fd << std::endl;
                }
            } else {
                // 客户端数据
                char buf[1024];
                ssize_t n = recv(fd, buf, sizeof(buf), 0);

                if (n <= 0) {
                    // 客户端断开
                    std::cout << "Client disconnected: fd=" << fd << std::endl;
                    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, nullptr);
                    close(fd);
                } else {
                    // Echo
                    send(fd, buf, n, 0);
                }
            }
        }
    }

    close(epfd);
    close(listen_fd);
    return 0;
}
```

4.4 epoll 两种触发模式

4.4.1 水平触发（Level Triggered, LT）

默认模式，类似 select/poll 的行为。

特点：

- 只要 fd 上有未处理的事件，epoll_wait 就会一直通知
- 可以不一次性读完所有数据
- 更安全，不容易丢失事件

```cpp
// LT模式示例
struct epoll_event ev;
ev.events = EPOLLIN;  // 默认是LT模式
ev.data.fd = fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);

// epoll_wait返回，表示fd可读
// 读取部分数据
char buf[100];
recv(fd, buf, sizeof(buf), 0);  // 只读100字节

// 下次epoll_wait仍会返回（因为还有数据未读）
```

4.4.2 边缘触发（Edge Triggered, ET）

高性能模式，但需要小心处理。

特点：

- 只在状态变化时通知一次
- 必须一次性读完所有数据（循环读取直到 EAGAIN）
- 更高效，但容易出错

```cpp
// ET模式示例
struct epoll_event ev;
ev.events = EPOLLIN | EPOLLET;  // 开启ET模式
ev.data.fd = fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);

// epoll_wait返回，表示fd可读
// 必须循环读取所有数据
while (true) {
    char buf[1024];
    ssize_t n = recv(fd, buf, sizeof(buf), 0);
    if (n < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            // 数据读完了
            break;
        }
        // 真正的错误
        perror("recv failed");
        break;
    } else if (n == 0) {
        // 连接关闭
        break;
    }
    // 处理数据...
}
```

4.4.3 LT vs ET 对比

特性	LT（水平触发）	ET（边缘触发）
触发时机	只要有事件就触发	状态变化时触发一次
数据读取	可以分次读取	必须一次读完
性能	一般	更高（减少系统调用）
难度	简单	复杂（容易丢事件）
阻塞/非阻塞	都支持	必须非阻塞

面试重点：为什么 ET 模式必须配合非阻塞 I/O？

- ET 模式下必须循环读取直到 EAGAIN
- 如果是阻塞 I/O，最后一次 read 会阻塞（因为已经没有数据）
- 非阻塞 I/O 会立即返回 EAGAIN，告诉你数据读完了

4.4.4 ET 模式完整示例

```cpp
#include <iostream>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <fcntl.h>
#include <cstring>
#include <errno.h>

void set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    fcntl(fd, F_SETFL, flags | O_NONBLOCK);
}
```

```
14
15  void handle_client(int fd) {
16      // ET模式：必须循环读取直到EAGAIN
17      while (true) {
18          char buf[1024];
19          ssize_t n = recv(fd, buf, sizeof(buf), 0);
20
21          if (n > 0) {
22              // 处理数据（这里是Echo）
23              send(fd, buf, n, 0);
24          } else if (n == 0) {
25              // 连接关闭
26              std::cout << "Client closed: fd=" << fd << std::endl;
27              close(fd);
28              return;
29          } else {
30              if (errno == EAGAIN || errno == EWOULDBLOCK) {
31                  // 数据读完了（正常情况）
32                  break;
33              } else {
34                  // 真正的错误
35                  perror("recv failed");
36                  close(fd);
37                  return;
38              }
39          }
40      }
41  }
42
43  int main() {
44      int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
45      int opt = 1;
46      setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
47      set_nonblocking(listen_fd);
48
49      struct sockaddr_in addr;
50      memset(&addr, 0, sizeof(addr));
51      addr.sin_family = AF_INET;
52      addr.sin_addr.s_addr = INADDR_ANY;
53      addr.sin_port = htons(8080);
54
55      bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr));
56      listen(listen_fd, 128);
57
58      int epfd = epoll_create1(0);
59
60      struct epoll_event ev;
61      ev.events = EPOLLIN | EPOLLET;  // ET模式
62      ev.data.fd = listen_fd;
63      epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, &ev);
64
65      std::cout << "Server started (ET mode)" << std::endl;
66
67      const int MAX_EVENTS = 64;
```

```
68      struct epoll_event events[MAX_EVENTS];
69
70      while (true) {
71          int nready = epoll_wait(epfd, events, MAX_EVENTS, -1);
72
73          for (int i = 0; i < nready; ++i) {
74              int fd = events[i].data.fd;
75
76              if (fd == listen_fd) {
77                  // ET模式：必须循环accept直到EAGAIN
78                  while (true) {
79                      int client_fd = accept(listen_fd, nullptr, nullptr);
80                      if (client_fd < 0) {
81                          if (errno != EAGAIN && errno != EWOULDBLOCK) {
82                              perror("accept failed");
83                          }
84                          break;
85                      }
86
87                      set_nonblocking(client_fd);
88
89                      struct epoll_event client_ev;
90                      client_ev.events = EPOLLIN | EPOLLET;
91                      client_ev.data.fd = client_fd;
92                      epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &client_ev);
93
94                      std::cout << "New client: fd=" << client_fd << std::endl;
95                  }
96              } else {
97                  // 处理客户端数据
98                  handle_client(fd);
99              }
100         }
101     }
102
103     close(epfd);
104     close(listen_fd);
105     return 0;
106 }
```

4.5 select / poll / epoll 性能对比

| 特性 | select | poll | epoll | |——|———|——|———-| | fd 数量限制 | 1024 | 无限制 | 无限制 | | 数据结构 | 位图 | 数组 | 红黑树+链表 | | fd 复制 | 每次复制 | 每次复制 | 不需要 | | 查找就绪 fd | O(n)扫描 | O(n)扫描 | O(1)直接返回 | | 性能 | O(n) | O(n) | O(1) | | 平台 | 跨平台 | 跨平台 | 仅 Linux |

性能对比（连接数 **vs** 性能）：

- 100 连接：三者性能相近
- 1000 连接：epoll 明显优于 select/poll
- 10000 连接：epoll 性能碾压（select/poll 几乎不可用）

选择建议：

- 少量连接（< 1000）：select/poll 足够
- 大量连接（> 10000）：必须使用 epoll
- 跨平台需求：select（或者使用 libevent/libev 等封装库）

5 Reactor 模式

5.1 什么是 Reactor 模式？

Reactor 模式是一种事件驱动的设计模式，用于处理并发 I/O。

核心思想：

- 将 I/O 事件的等待和处理分离
- 主线程负责监听 I/O 事件（epoll_wait）
- 工作线程负责处理业务逻辑

Reactor 模式的组件：

1. **Reactor**：事件循环，监听 I/O 事件（epoll）
2. **Event Handler**：事件处理器，定义回调函数
3. **Demultiplexer**：I/O 多路复用器（epoll_wait）
4. **Dispatcher**：事件分发器，将事件分发给对应的 Handler

5.2 Single Reactor 单线程模式

结构：

- 1 个 Reactor 线程
- 负责 accept、read、处理业务、write

```cpp
// 简化的Single Reactor模型
class SingleReactor {
    int epfd_;
    int listen_fd_;

public:
    void run() {
        epfd_ = epoll_create1(0);

        // 创建并监听listen socket
        listen_fd_ = create_listen_socket(8080);
        add_event(listen_fd_, EPOLLIN);

        struct epoll_event events[64];
        while (true) {
            int nready = epoll_wait(epfd_, events, 64, -1);
            for (int i = 0; i < nready; ++i) {
                int fd = events[i].data.fd;

                if (fd == listen_fd_) {
                    handle_accept();
                } else {
                    handle_client(fd);
                }
            }
        }
    }

    void handle_accept() {
        int client_fd = accept(listen_fd_, nullptr, nullptr);
        set_nonblocking(client_fd);
        add_event(client_fd, EPOLLIN | EPOLLET);
    }

```

```
    void handle_client(int fd) {
        // 读取数据
        char buf[1024];
        ssize_t n = recv(fd, buf, sizeof(buf), 0);
        if (n <= 0) {
            close(fd);
            return;
        }

        // 处理业务逻辑（在同一线程）
        process_data(buf, n);

        // 发送响应
        send(fd, buf, n, 0);
    }
};
```

优点：

- 简单，易于实现
- 无需考虑线程安全

缺点：

- 单线程，无法利用多核 CPU
- 业务逻辑阻塞会影响其他连接

适用场景：

- 连接数少（< 100）
- 业务逻辑简单（如 Echo）

5.3 Single Reactor + 线程池模式

结构：

- 1 个 Reactor 线程：负责 accept、read、write
- N 个 Worker 线程：负责处理业务逻辑

这是你简历中高并发文件传输系统的架构！

C++

```
#include <iostream>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <vector>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <fcntl.h>
#include <cstring>

// 线程池
class ThreadPool {
    std::vector<std::thread> threads_;
    std::queue<std::function<void()>> tasks_;
    std::mutex mutex_;
    std::condition_variable cv_;
```

```
21      bool stop_ = false;
22
23  public:
24      ThreadPool(size_t num_threads) {
25          for (size_t i = 0; i < num_threads; ++i) {
26              threads_.emplace_back([this] {
27                  while (true) {
28                      std::function<void()> task;
29                      {
30                          std::unique_lock<std::mutex> lock(mutex_);
31                          cv_.wait(lock, [this] { return stop_ || !tasks_.empty(); });
32                          if (stop_ && tasks_.empty()) return;
33                          task = std::move(tasks_.front());
34                          tasks_.pop();
35                      }
36                      task();
37                  }
38              });
39          }
40      }
41
42      ~ThreadPool() {
43          {
44              std::unique_lock<std::mutex> lock(mutex_);
45              stop_ = true;
46          }
47          cv_.notify_all();
48          for (auto& t : threads_) {
49              t.join();
50          }
51      }
52
53      void submit(std::function<void()> task) {
54          {
55              std::unique_lock<std::mutex> lock(mutex_);
56              tasks_.push(std::move(task));
57          }
58          cv_.notify_one();
59      }
60  };
61
62  // Reactor + 线程池服务器
63  class ReactorServer {
64      int epfd_;
65      int listen_fd_;
66      ThreadPool pool_;
67
68  public:
69      ReactorServer(int num_threads) : pool_(num_threads) {}
70
71      void start(uint16_t port) {
72          // 创建epoll
73          epfd_ = epoll_create1(0);
74
75          // 创建监听socket
```

```
76          listen_fd_ = socket(AF_INET, SOCK_STREAM, 0);
77          int opt = 1;
78          setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
79          set_nonblocking(listen_fd_);
80
81          struct sockaddr_in addr;
82          memset(&addr, 0, sizeof(addr));
83          addr.sin_family = AF_INET;
84          addr.sin_addr.s_addr = INADDR_ANY;
85          addr.sin_port = htons(port);
86
87          bind(listen_fd_, (struct sockaddr*)&addr, sizeof(addr));
88          listen(listen_fd_, 128);
89
90          // 添加到epoll
91          struct epoll_event ev;
92          ev.events = EPOLLIN | EPOLLET;
93          ev.data.fd = listen_fd_;
94          epoll_ctl(epfd_, EPOLL_CTL_ADD, listen_fd_, &ev);
95
96          std::cout << "Reactor server started on port " << port << std::endl;
97
98          // 事件循环（主线程）
99          run();
100     }
101
102 private:
103     void run() {
104         const int MAX_EVENTS = 64;
105         struct epoll_event events[MAX_EVENTS];
106
107         while (true) {
108             int nready = epoll_wait(epfd_, events, MAX_EVENTS, -1);
109
110             for (int i = 0; i < nready; ++i) {
111                 int fd = events[i].data.fd;
112
113                 if (fd == listen_fd_) {
114                     handle_accept();
115                 } else if (events[i].events & EPOLLIN) {
116                     handle_read(fd);
117                 }
118             }
119         }
120     }
121
122     void handle_accept() {
123         while (true) {
124             int client_fd = accept(listen_fd_, nullptr, nullptr);
125             if (client_fd < 0) break;
126
127             set_nonblocking(client_fd);
128
129             struct epoll_event ev;
130             ev.events = EPOLLIN | EPOLLET;
```

```
131             ev.data.fd = client_fd;
132             epoll_ctl(epfd_, EPOLL_CTL_ADD, client_fd, &ev);
133 
134             std::cout << "New connection: fd=" << client_fd << std::endl;
135         }
136     }
137 
138     void handle_read(int fd) {
139         // 读取数据（主线程）
140         char buf[4096];
141         ssize_t n = recv(fd, buf, sizeof(buf), 0);
142 
143         if (n <= 0) {
144             close(fd);
145             return;
146         }
147 
148         // 将业务逻辑提交给线程池（工作线程）
149         std::string data(buf, n);
150         pool_.submit([this, fd, data] {
151             // 处理业务逻辑（工作线程）
152             std::string response = process_request(data);
153 
154             // 发送响应（需要线程安全）
155             send(fd, response.c_str(), response.size(), 0);
156         });
157     }
158 
159     std::string process_request(const std::string& data) {
160         // 模拟耗时的业务逻辑
161         std::this_thread::sleep_for(std::chrono::milliseconds(10));
162         return data;  // Echo
163     }
164 
165     void set_nonblocking(int fd) {
166         int flags = fcntl(fd, F_GETFL, 0);
167         fcntl(fd, F_SETFL, flags | O_NONBLOCK);
168     }
169 };
170 
171 int main() {
172     ReactorServer server(4);  // 4个工作线程
173     server.start(8080);
174     return 0;
175 }
```

优点：

- 充分利用多核 CPU
- I/O 操作和业务逻辑分离
- 主线程专注于 I/O，不会被业务逻辑阻塞

缺点：

- 单 Reactor 可能成为瓶颈（高并发连接）
- 需要考虑线程安全（send 操作）

性能指标：

- 可以达到 10000+ QPS
- 适合中等规模的服务（< 10000 连接）

这正是你简历中的架构：

```
1  主线程负责连接管理，工作线程池处理文件传输
```

5.4 Multi Reactor + 线程池模式（主从 Reactor）

结构：

- 1 个 Main Reactor：负责 accept（主线程）
- N 个 Sub Reactor：负责 I/O 处理（子线程）
- M 个 Worker 线程：负责业务逻辑（线程池）

这是 **Muduo/Netty** 等高性能网络库的架构

```cpp
// 简化的Multi Reactor架构
class MainReactor {
    int epfd_;
    int listen_fd_;
    std::vector<SubReactor*> sub_reactors_;
    size_t next_sub_ = 0;

public:
    void add_sub_reactor(SubReactor* sub) {
        sub_reactors_.push_back(sub);
    }

    void run() {
        epfd_ = epoll_create1(0);
        listen_fd_ = create_listen_socket(8080);

        struct epoll_event ev;
        ev.events = EPOLLIN;
        ev.data.fd = listen_fd_;
        epoll_ctl(epfd_, EPOLL_CTL_ADD, listen_fd_, &ev);

        while (true) {
            struct epoll_event events[64];
            int nready = epoll_wait(epfd_, events, 64, -1);

            for (int i = 0; i < nready; ++i) {
                if (events[i].data.fd == listen_fd_) {
                    int client_fd = accept(listen_fd_, nullptr, nullptr);

                    // 轮询分配给Sub Reactor
                    SubReactor* sub = sub_reactors_[next_sub_];
                    next_sub_ = (next_sub_ + 1) % sub_reactors_.size();

                    sub->add_connection(client_fd);
                }
            }
        }
    }
};

```

```
class SubReactor {
    int epfd_;
    std::thread thread_;
    ThreadPool* pool_;

public:
    SubReactor(ThreadPool* pool) : pool_(pool) {
        epfd_ = epoll_create1(0);
        thread_ = std::thread([this] { run(); });
    }

    void add_connection(int fd) {
        set_nonblocking(fd);

        struct epoll_event ev;
        ev.events = EPOLLIN | EPOLLET;
        ev.data.fd = fd;
        epoll_ctl(epfd_, EPOLL_CTL_ADD, fd, &ev);
    }

private:
    void run() {
        while (true) {
            struct epoll_event events[64];
            int nready = epoll_wait(epfd_, events, 64, -1);

            for (int i = 0; i < nready; ++i) {
                int fd = events[i].data.fd;

                // 读取数据
                char buf[4096];
                ssize_t n = recv(fd, buf, sizeof(buf), 0);
                if (n <= 0) {
                    close(fd);
                    continue;
                }

                // 提交给线程池处理
                std::string data(buf, n);
                pool_->submit([fd, data] {
                    // 业务逻辑
                    process_and_send(fd, data);
                });
            }
        }
    }
};
```

优点：

- Main Reactor 专注于 accept，不会成为瓶颈
- 多个 Sub Reactor 负载均衡，充分利用多核
- 最高性能的架构

缺点：

- 实现复杂

- 需要仔细设计线程间通信

性能指标：
- 可以达到 100000+ QPS
- 适合超高并发场景（> 100000 连接）

典型应用：
- Nginx：Multi-Process Reactor（多进程版本）
- Muduo/Netty：Multi-Reactor + 线程池
- Redis 6.0+：Multi-Reactor（I/O 线程）

5.5 Reactor 模式总结

| 模式 | 线程数 | QPS | 适用场景 | |——|———-|——|———| | Single Reactor | 1 | < 10K | 简单服务、连接少 | | Reactor + 线程池 | 1 + N | 10K - 50K | 中等并发、业务复杂 | | Multi Reactor + 线程池 | 1 + N + M | > 100K | 超高并发、生产环境 |

面试重点：

1. 为什么需要 **Reactor** 模式？
   - 传统阻塞 I/O 无法处理高并发
   - 多线程（每连接一线程）资源消耗大
   - Reactor 利用 I/O 多路复用，一个线程处理多个连接
2. **Reactor vs Proactor**？
   - Reactor：同步非阻塞，应用程序负责读写数据
   - Proactor：异步 I/O，内核负责读写数据（Windows IOCP、Linux io_uring）
3. 你的项目用的哪种 **Reactor**？
   - **Single Reactor +** 线程池
   - 主线程负责连接管理，工作线程处理文件传输
   - 适合中等并发场景（QPS 6000+）

6 高性能网络编程技术

6.1 TCP 粘包问题

6.1.1 什么是粘包？

TCP 是字节流协议，没有消息边界。发送方连续发送多个数据包，接收方可能一次性收到。

```cpp
// 发送方
send(fd, "Hello", 5);
send(fd, "World", 5);

// 接收方可能收到：
// 1. "Hello" + "World" (粘包)
// 2. "Hel" + "loWorld" (拆包)
// 3. "Hello"，然后"World" (正常)
```

6.1.2 为什么会粘包？

1. **Nagle** 算法：将小包合并成大包发送
2. **TCP** 缓冲区：多次写入的数据可能一次性发送
3. 接收缓冲区：多个包可能一次性读取

6.1.3 解决方案

6.1.3.1 方案 1：固定长度

每个消息固定 N 字节。

```cpp
const int MSG_LEN = 1024;

// 发送
void send_fixed(int fd, const std::string& msg) {
    char buf[MSG_LEN] = {0};
    memcpy(buf, msg.c_str(), std::min(msg.size(), sizeof(buf)));
    send(fd, buf, MSG_LEN, 0);
}

// 接收
std::string recv_fixed(int fd) {
    char buf[MSG_LEN];
    int total = 0;
    while (total < MSG_LEN) {
        int n = recv(fd, buf + total, MSG_LEN - total, 0);
        if (n <= 0) break;
        total += n;
    }
    return std::string(buf, MSG_LEN);
}
```

缺点：浪费空间（短消息也要占用固定长度）

6.1.3.2 方案 2：分隔符

使用特殊字符分隔消息（如\\n、\\0）。

```cpp
// 发送
void send_delimited(int fd, const std::string& msg) {
    std::string data = msg + "\n";
    send(fd, data.c_str(), data.size(), 0);
```

```
}

// 接收（需要维护缓冲区）
class MessageReader {
    std::string buffer_;

public:
    std::vector<std::string> read_messages(int fd) {
        char buf[1024];
        ssize_t n = recv(fd, buf, sizeof(buf), 0);
        if (n > 0) {
            buffer_.append(buf, n);
        }

        std::vector<std::string> messages;
        size_t pos;
        while ((pos = buffer_.find('\n')) != std::string::npos) {
            messages.push_back(buffer_.substr(0, pos));
            buffer_.erase(0, pos + 1);
        }
        return messages;
    }
};
```

缺点：需要转义分隔符，不适合二进制数据

6.1.3.3 方案 3：消息长度前缀（最常用）

在消息前加上长度字段。

```
// 消息格式：[4字节长度][消息内容]
struct Message {
    uint32_t length;  // 网络字节序
    char data[0];     // 柔性数组
};

// 发送
void send_message(int fd, const std::string& msg) {
    uint32_t len = htonl(msg.size());
    send(fd, &len, sizeof(len), 0);
    send(fd, msg.c_str(), msg.size(), 0);
}

// 接收
class MessageReader {
    std::string buffer_;

public:
    std::vector<std::string> read_messages(int fd) {
        char buf[4096];
        ssize_t n = recv(fd, buf, sizeof(buf), 0);
        if (n > 0) {
            buffer_.append(buf, n);
        }

        std::vector<std::string> messages;
```

```
27          while (buffer_.size() >= 4) {
28              // 读取长度
29              uint32_t len;
30              memcpy(&len, buffer_.data(), 4);
31              len = ntohl(len);
32
33              // 检查是否有完整消息
34              if (buffer_.size() < 4 + len) {
35                  break;  // 数据不完整
36              }
37
38              // 提取消息
39              messages.push_back(buffer_.substr(4, len));
40              buffer_.erase(0, 4 + len);
41          }
42          return messages;
43      }
44  };
```

优点：

- 支持任意长度和类型的数据
- 不需要转义
- **HTTP**、**RPC** 等协议都使用此方案

6.1.3.4 方案 4：应用层协议（如 HTTP、protobuf）

使用成熟的协议格式。

```
1  // HTTP示例
2  GET /index.html HTTP/1.1\r\n
3  Host: example.com\r\n
4  Content-Length: 123\r\n
5  \r\n
6  [123 bytes data]
7
8  // Protobuf + 长度前缀
9  [varint length][protobuf message]
```

面试重点：你的项目如何解决粘包？

- 文件传输：使用长度前缀
- RPC 通信：protobuf 自带长度编码

6.2 惊群效应（Thundering Herd）

6.2.1 什么是惊群？

多个进程/线程监听同一个 socket，当有新连接时，所有进程/线程都被唤醒，但只有一个能 accept 成功，其他白白浪费 CPU。

```
1   // 多进程服务器（有惊群问题）
2   int listen_fd = create_listen_socket(8080);
3
4   for (int i = 0; i < 4; ++i) {
5       if (fork() == 0) {
6           // 子进程
7           while (true) {
8               // 所有子进程都在这里等待
```

```
            int client_fd = accept(listen_fd, nullptr, nullptr);
            // 只有一个子进程能accept成功
            handle_client(client_fd);
        }
    }
}
```

6.2.2 惊群的危害

1. **CPU 浪费**：大量进程/线程被唤醒后又睡眠
2. **性能下降**：上下文切换开销大
3. **负载不均**：可能某个进程连续 accept 多次

6.2.3 解决方案

6.2.3.1 方案 1：accept 惊群（Linux 2.6 已解决）

Linux 2.6+内核已经解决了 accept 的惊群问题：只唤醒一个进程。

6.2.3.2 方案 2：epoll 惊群

问题：多个进程/线程共享一个 epoll fd，仍有惊群。

```cpp
// 有惊群问题
int epfd = epoll_create1(0);
epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, ...);

for (int i = 0; i < 4; ++i) {
    if (fork() == 0) {
        while (true) {
            epoll_wait(epfd, ...);  // 所有进程都会被唤醒
            accept(listen_fd, ...);
        }
    }
}
```

解决：每个进程/线程有自己的 epoll fd。

```cpp
// 无惊群问题
for (int i = 0; i < 4; ++i) {
    if (fork() == 0) {
        int epfd = epoll_create1(0);  // 每个进程独立的epoll
        epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, ...);

        while (true) {
            epoll_wait(epfd, ...);
            accept(listen_fd, ...);
        }
    }
}
```

6.2.3.3 方案 3：SO_REUSEPORT（Linux 3.9+）

允许多个 socket 绑定到同一个端口，内核负责负载均衡。

```cpp
for (int i = 0; i < 4; ++i) {
    if (fork() == 0) {
        // 每个进程创建自己的listen socket
        int listen_fd = socket(AF_INET, SOCK_STREAM, 0);

```

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

        bind(listen_fd, ...);
        listen(listen_fd, ...);

        // 内核会将新连接分配给不同的进程
        while (true) {
            int client_fd = accept(listen_fd, ...);
            handle_client(client_fd);
        }
    }
}
```

优点：

- 内核负载均衡
- 无惊群问题
- **Nginx**、**Redis** 等都使用此方案

6.2.3.4 方案 4：加锁

使用锁保证只有一个进程/线程调用 accept。

```C++
std::mutex accept_mutex;

void worker_thread(int listen_fd) {
    while (true) {
        {
            std::lock_guard<std::mutex> lock(accept_mutex);
            int client_fd = accept(listen_fd, ...);
            if (client_fd < 0) continue;
            handle_client(client_fd);
        }
    }
}
```

缺点：加锁开销

6.3 零拷贝（Zero-Copy）

6.3.1 传统 I/O 的问题

```C++
// 传统文件传输
int fd = open("file.dat", O_RDONLY);
char buf[4096];
while (true) {
    ssize_t n = read(fd, buf, sizeof(buf));  // 1. 从磁盘->内核缓冲区->用户缓冲区
    if (n <= 0) break;
    send(sockfd, buf, n, 0);                 // 2. 从用户缓冲区->内核缓冲区->网卡
}
```

问题：数据经过 4 次复制、4 次上下文切换

1. 磁盘 → 内核 read 缓冲区（DMA）
2. 内核 read 缓冲区 → 用户缓冲区（CPU）
3. 用户缓冲区 → 内核 socket 缓冲区（CPU）
4. 内核 socket 缓冲区 → 网卡（DMA）

6.3.2 sendfile（Linux 2.2+）

减少到 2 次复制、2 次上下文切换。

```cpp
#include <sys/sendfile.h>

// 零拷贝文件传输
int fd = open("file.dat", O_RDONLY);
off_t offset = 0;
struct stat st;
fstat(fd, &st);

// 数据直接从文件->socket，不经过用户空间
sendfile(sockfd, fd, &offset, st.st_size);
```

数据流：

1. 磁盘 → 内核 read 缓冲区（DMA）
2. 内核 read 缓冲区 → 内核 socket 缓冲区（CPU）
3. 内核 socket 缓冲区 → 网卡（DMA）

优点：

- 减少 2 次 CPU 复制
- 减少 2 次上下文切换

6.3.3 splice（Linux 2.6.17+）

在两个文件描述符之间移动数据，不经过用户空间。

```cpp
#include <fcntl.h>

int pipefd[2];
pipe(pipefd);

// 文件 -> pipe -> socket
splice(fd, nullptr, pipefd[1], nullptr, 4096, SPLICE_F_MOVE);
splice(pipefd[0], nullptr, sockfd, nullptr, 4096, SPLICE_F_MOVE);
```

6.3.4 mmap + write

将文件映射到内存，减少一次复制。

```cpp
// 映射文件到内存
int fd = open("file.dat", O_RDONLY);
struct stat st;
fstat(fd, &st);

void* addr = mmap(nullptr, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);

// 直接发送（实际上还是有复制）
send(sockfd, addr, st.st_size, 0);

munmap(addr, st.st_size);
```

缺点：

- 仍有 3 次复制
- mmap/munmap 开销
- 不如 sendfile 高效

6.3.5 零拷贝总结

| 方案 | 数据复制次数 | 上下文切换 | 适用场景 | |---|---|---|---| | read/write | 4 次 | 4 次 | 小文件、需要处理数据 | | sendfile | 2 次 | 2 次 | 大文件、不需要处理数据 | | mmap + write | 3 次 | 4 次 | 需要随机访问 | | splice | 0 次（理想） | 2 次 | 管道、socket 间转发 |

面试重点：你的项目用了零拷贝吗？

- 是的，文件传输使用 **sendfile**
- 减少 CPU 复制，提升吞吐量
- 适合大文件传输场景

6.4 SIGPIPE 信号处理

6.4.1 什么是 SIGPIPE？

向已关闭的 socket 写数据，会收到 SIGPIPE 信号，默认行为是终止进程。

```cpp
// 客户端关闭连接
close(client_fd);

// 服务器仍然发送数据
send(client_fd, buf, len, 0);  // 第一次send返回错误
send(client_fd, buf, len, 0);  // 第二次send触发SIGPIPE，进程退出！
```

6.4.2 解决方案

6.4.2.1 方案 1：忽略 SIGPIPE 信号

```cpp
// 进程启动时忽略SIGPIPE
signal(SIGPIPE, SIG_IGN);

// 之后send会返回错误（errno = EPIPE），而不是终止进程
ssize_t n = send(fd, buf, len, 0);
if (n < 0) {
    if (errno == EPIPE) {
        // 连接已关闭
        close(fd);
    }
}
```

6.4.2.2 方案 2：使用 MSG_NOSIGNAL 标志

```cpp
// 单次send不产生SIGPIPE
ssize_t n = send(fd, buf, len, MSG_NOSIGNAL);
if (n < 0) {
    if (errno == EPIPE) {
        // 连接已关闭
        close(fd);
    }
}
```

推荐：方案 1（全局忽略 SIGPIPE）

6.5 连接管理技巧

6.5.1 半关闭（shutdown）

close 关闭读和写，shutdown 可以只关闭一个方向。

```cpp
// shutdown 函数
```

```
int shutdown(int sockfd, int how);
// how: SHUT_RD（关闭读）、SHUT_WR（关闭写）、SHUT_RDWR（都关闭）

// 应用场景：客户端发送完数据，但还要接收响应
send(sockfd, data, len, 0);
shutdown(sockfd, SHUT_WR);  // 关闭写，发送FIN

// 仍然可以接收数据
recv(sockfd, buf, sizeof(buf), 0);

close(sockfd);
```

shutdown vs close：

- shutdown：立即关闭，即使还有引用（多个进程共享 socket）
- close：引用计数-1，计数为 0 才真正关闭

6.5.2 SO_LINGER

控制 close 的行为。

```
struct linger {
    int l_onoff;   // 0: 默认行为，1: 启用linger
    int l_linger;  // 延迟时间（秒）
};

// 场景1：立即关闭，丢弃未发送数据
struct linger opt = {1, 0};
setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &opt, sizeof(opt));
close(sockfd);  // 立即返回，发送RST而非FIN

// 场景2：阻塞等待数据发送完成
struct linger opt = {1, 10};  // 最多等待10秒
setsockopt(sockfd, SOL_SOCKET, SO_LINGER, &opt, sizeof(opt));
close(sockfd);  // 阻塞最多10秒，直到数据发送完或超时
```

默认行为（l_onoff=0）：

- close 立即返回
- 内核负责发送剩余数据和 FIN
- 最安全的方式

6.6 性能优化技巧

6.6.1 1. 合理设置 TCP 缓冲区

```
// 文件传输服务器：增大缓冲区
int rcvbuf = 2 * 1024 * 1024;  // 2MB
int sndbuf = 2 * 1024 * 1024;
setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &rcvbuf, sizeof(rcvbuf));
setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &sndbuf, sizeof(sndbuf));
```

6.6.2 2. 禁用 Nagle 算法（低延迟场景）

```
int opt = 1;
setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt));
```

6.6.3 3. 启用 TCP Fast Open（Linux 3.7+）

```
// 服务器端
```

```
int qlen = 5;
setsockopt(listen_fd, IPPROTO_TCP, TCP_FASTOPEN, &qlen, sizeof(qlen));

// 客户端：第一次连接时发送数据
sendto(sockfd, data, len, MSG_FASTOPEN, (struct sockaddr*)&addr, sizeof(addr));
```

优点：减少一次 RTT（在 SYN 包中携带数据）

6.6.4 4. CPU 亲和性（affinity）

```cpp
#include <sched.h>

// 将线程绑定到特定CPU核心
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(core_id, &cpuset);
pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset);
```

优点：

- 提高 CPU 缓存命中率
- 减少上下文切换

6.6.5 5. 对象池复用

```cpp
// Buffer对象池（你的项目中使用了）
class BufferPool {
    std::vector<char*> pool_;
    std::mutex mutex_;

public:
    BufferPool(size_t capacity, size_t buf_size) {
        for (size_t i = 0; i < capacity; ++i) {
            pool_.push_back(new char[buf_size]);
        }
    }

    char* acquire() {
        std::lock_guard<std::mutex> lock(mutex_);
        if (pool_.empty()) return new char[4096];
        char* buf = pool_.back();
        pool_.pop_back();
        return buf;
    }

    void release(char* buf) {
        std::lock_guard<std::mutex> lock(mutex_);
        pool_.push_back(buf);
    }
};
```

优点：

- 减少内存分配/释放开销
- 提高性能

7 实战案例

7.1 案例 1：高并发文件传输服务器（对应你的简历项目）

需求：

- 支持 500+并发连接
- QPS 6000+
- P95 延迟 < 15ms

架构设计：

```
            Client Connections
                    |
                    v
          +-------------------+
          |  Main Thread      |
          |  (epoll + accept) |  <- Single Reactor
          +-------------------+
                    |
        +-----------+-----------+
        |           |           |
        v           v           v
  +---------+  +---------+  +---------+
  | Worker |  | Worker |  | Worker |  <- Thread Pool
  | Thread |  | Thread |  | Thread |
  +---------+  +---------+  +---------+
        |           |           |
        +-----------+-----------+
                    |
                    v
          File I/O (sendfile)
```

核心代码实现：

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
#include <sys/epoll.h>
#include <sys/socket.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <netinet/in.h>
#include <cstring>
#include "ThreadPool.h"  // 前面实现的线程池
#include "AsyncLogger.h"  // 异步日志

class FileTransferServer {
    int epfd_;
    int listen_fd_;
    ThreadPool pool_;
    AsyncLogger logger_;

    // 文件元数据（哈希表 + 读写锁）
    std::unordered_map<std::string, std::string> file_map_;
    mutable std::shared_mutex map_mutex_;
```

```
24
25  public:
26      FileTransferServer(int num_threads)
27          : pool_(num_threads), logger_("server.log") {
28          // 初始化文件列表
29          file_map_["test.dat"] = "/data/test.dat";
30          file_map_["large.bin"] = "/data/large.bin";
31      }
32
33      void start(uint16_t port) {
34          // 创建epoll
35          epfd_ = epoll_create1(0);
36
37          // 创建监听socket
38          listen_fd_ = socket(AF_INET, SOCK_STREAM, 0);
39          int opt = 1;
40          setsockopt(listen_fd_, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
41          set_nonblocking(listen_fd_);
42
43          struct sockaddr_in addr;
44          memset(&addr, 0, sizeof(addr));
45          addr.sin_family = AF_INET;
46          addr.sin_addr.s_addr = INADDR_ANY;
47          addr.sin_port = htons(port);
48
49          bind(listen_fd_, (struct sockaddr*)&addr, sizeof(addr));
50          listen(listen_fd_, 128);
51
52          // 添加到epoll
53          struct epoll_event ev;
54          ev.events = EPOLLIN | EPOLLET;
55          ev.data.fd = listen_fd_;
56          epoll_ctl(epfd_, EPOLL_CTL_ADD, listen_fd_, &ev);
57
58          logger_.log("Server started on port " + std::to_string(port));
59
60          // 事件循环
61          run();
62      }
63
64  private:
65      void run() {
66          const int MAX_EVENTS = 64;
67          struct epoll_event events[MAX_EVENTS];
68
69          while (true) {
70              int nready = epoll_wait(epfd_, events, MAX_EVENTS, -1);
71
72              for (int i = 0; i < nready; ++i) {
73                  int fd = events[i].data.fd;
74
75                  if (fd == listen_fd_) {
76                      handle_accept();
77                  } else if (events[i].events & EPOLLIN) {
78                      handle_request(fd);
```

```
                }
            }
        }
    }

    void handle_accept() {
        while (true) {
            int client_fd = accept(listen_fd_, nullptr, nullptr);
            if (client_fd < 0) break;

            set_nonblocking(client_fd);

            struct epoll_event ev;
            ev.events = EPOLLIN | EPOLLET;
            ev.data.fd = client_fd;
            epoll_ctl(epfd_, EPOLL_CTL_ADD, client_fd, &ev);

            logger_.log("New connection: fd=" + std::to_string(client_fd));
        }
    }

    void handle_request(int fd) {
        // 读取请求（文件名）
        char buf[256];
        ssize_t n = recv(fd, buf, sizeof(buf), 0);
        if (n <= 0) {
            close_connection(fd);
            return;
        }

        std::string filename(buf, n);

        // 提交给线程池处理文件传输
        pool_.submit([this, fd, filename] {
            transfer_file(fd, filename);
        });
    }

    void transfer_file(int sockfd, const std::string& filename) {
        // 查找文件路径（读锁）
        std::string filepath;
        {
            std::shared_lock<std::shared_mutex> lock(map_mutex_);
            auto it = file_map_.find(filename);
            if (it == file_map_.end()) {
                send_error(sockfd, "File not found");
                return;
            }
            filepath = it->second;
        }

        // 打开文件
        int file_fd = open(filepath.c_str(), O_RDONLY);
        if (file_fd < 0) {
            send_error(sockfd, "Cannot open file");
```

```
134             return;
135         }
136 
137         // 获取文件大小
138         struct stat st;
139         fstat(file_fd, &st);
140 
141         logger_.log("Transferring file: " + filename +
142                     ", size: " + std::to_string(st.st_size));
143 
144         // 使用sendfile零拷贝传输
145         off_t offset = 0;
146         while (offset < st.st_size) {
147             ssize_t sent = sendfile(sockfd, file_fd, &offset, st.st_size - offset);
148             if (sent <= 0) {
149                 if (errno == EAGAIN) {
150                     continue;  // 缓冲区满，重试
151                 }
152                 break;  // 错误或连接关闭
153             }
154         }
155 
156         close(file_fd);
157         close_connection(sockfd);
158 
159         logger_.log("Transfer completed: " + filename);
160     }
161 
162     void send_error(int sockfd, const std::string& msg) {
163         send(sockfd, msg.c_str(), msg.size(), 0);
164         close_connection(sockfd);
165     }
166 
167     void close_connection(int fd) {
168         epoll_ctl(epfd_, EPOLL_CTL_DEL, fd, nullptr);
169         close(fd);
170     }
171 
172     void set_nonblocking(int fd) {
173         int flags = fcntl(fd, F_GETFL, 0);
174         fcntl(fd, F_SETFL, flags | O_NONBLOCK);
175     }
176 };
177 
178 int main() {
179     // 忽略SIGPIPE
180     signal(SIGPIPE, SIG_IGN);
181 
182     // 4个工作线程
183     FileTransferServer server(4);
184     server.start(8080);
185 
186     return 0;
187 }
```

关键技术点：

1. **Reactor + 线程池**：主线程处理连接和读请求，工作线程处理文件传输
2. **epoll ET 模式**：边缘触发，高性能
3. **sendfile 零拷贝**：减少数据复制，提升吞吐
4. 读写锁：元数据并发读，细粒度控制
5. 异步日志：双缓冲，不阻塞主流程
6. 非阻塞 **I/O**：所有 socket 都设置为非阻塞

性能优化手段：

- Buffer 对象池（减少 new/delete）
- TCP 缓冲区调优（增大 SO_RCVBUF/SO_SNDBUF）
- CPU 亲和性（绑定线程到核心）

7.2 案例 2：简单的 RPC 通信框架（对应你的 Raft 项目）

需求：

- 支持 RequestVote、AppendEntries 等 RPC 调用
- 使用 protobuf 序列化
- 异步 RPC

消息格式：

```
[4字节消息长度][protobuf消息]
```

Protobuf 定义：

```protobuf
// raft.proto
syntax = "proto3";

message RequestVoteRequest {
    uint32 term = 1;
    uint32 candidate_id = 2;
    uint32 last_log_index = 3;
    uint32 last_log_term = 4;
}

message RequestVoteResponse {
    uint32 term = 1;
    bool vote_granted = 2;
}
```

RPC 客户端：

```C++
#include <google/protobuf/message.h>
#include "raft.pb.h"

class RpcClient {
    int sockfd_;
    std::string buffer_;

public:
    RpcClient(const std::string& host, uint16_t port) {
        sockfd_ = socket(AF_INET, SOCK_STREAM, 0);

        struct sockaddr_in addr;
        memset(&addr, 0, sizeof(addr));
        addr.sin_family = AF_INET;
```

```
15          addr.sin_port = htons(port);
16          inet_pton(AF_INET, host.c_str(), &addr.sin_addr);
17
18          if (connect(sockfd_, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
19              throw std::runtime_error("Connect failed");
20          }
21      }
22
23      ~RpcClient() {
24          close(sockfd_);
25      }
26
27      // 发送RPC请求
28      void send_message(const google::protobuf::Message& msg) {
29          // 序列化
30          std::string data;
31          msg.SerializeToString(&data);
32
33          // 发送长度前缀
34          uint32_t len = htonl(data.size());
35          send(sockfd_, &len, sizeof(len), 0);
36
37          // 发送消息内容
38          send(sockfd_, data.c_str(), data.size(), 0);
39      }
40
41      // 接收RPC响应
42      bool recv_message(google::protobuf::Message& msg) {
43          // 读取长度前缀
44          while (buffer_.size() < 4) {
45              char buf[1024];
46              ssize_t n = recv(sockfd_, buf, sizeof(buf), 0);
47              if (n <= 0) return false;
48              buffer_.append(buf, n);
49          }
50
51          uint32_t len;
52          memcpy(&len, buffer_.data(), 4);
53          len = ntohl(len);
54
55          // 读取完整消息
56          while (buffer_.size() < 4 + len) {
57              char buf[1024];
58              ssize_t n = recv(sockfd_, buf, sizeof(buf), 0);
59              if (n <= 0) return false;
60              buffer_.append(buf, n);
61          }
62
63          // 反序列化
64          std::string data = buffer_.substr(4, len);
65          buffer_.erase(0, 4 + len);
66
67          return msg.ParseFromString(data);
68      }
```

```
    // 同步RPC调用
    RequestVoteResponse request_vote(const RequestVoteRequest& req) {
        send_message(req);

        RequestVoteResponse resp;
        recv_message(resp);

        return resp;
    }
};

// 使用示例
int main() {
    RpcClient client("192.168.1.100", 8080);

    // 构造请求
    RequestVoteRequest req;
    req.set_term(10);
    req.set_candidate_id(1);
    req.set_last_log_index(100);
    req.set_last_log_term(9);

    // 发送RPC
    RequestVoteResponse resp = client.request_vote(req);

    if (resp.vote_granted()) {
        std::cout << "Vote granted!" << std::endl;
    }

    return 0;
}
```

RPC 服务器：

```
class RpcServer {
    int epfd_;
    int listen_fd_;
    std::unordered_map<int, std::string> buffers_;  // 每个连接的接收缓冲区

public:
    void start(uint16_t port) {
        epfd_ = epoll_create1(0);

        listen_fd_ = socket(AF_INET, SOCK_STREAM, 0);
        // ... bind, listen, epoll_ctl ...

        run();
    }

private:
    void run() {
        struct epoll_event events[64];

        while (true) {
```

```
            int nready = epoll_wait(epfd_, events, 64, -1);

            for (int i = 0; i < nready; ++i) {
                int fd = events[i].data.fd;

                if (fd == listen_fd_) {
                    handle_accept();
                } else {
                    handle_message(fd);
                }
            }
        }
    }

    void handle_message(int fd) {
        char buf[4096];
        ssize_t n = recv(fd, buf, sizeof(buf), 0);
        if (n <= 0) {
            close_connection(fd);
            return;
        }

        auto& buffer = buffers_[fd];
        buffer.append(buf, n);

        // 尝试解析完整消息
        while (buffer.size() >= 4) {
            uint32_t len;
            memcpy(&len, buffer.data(), 4);
            len = ntohl(len);

            if (buffer.size() < 4 + len) break;  // 数据不完整

            // 提取并处理消息
            std::string data = buffer.substr(4, len);
            buffer.erase(0, 4 + len);

            process_rpc(fd, data);
        }
    }

    void process_rpc(int fd, const std::string& data) {
        // 这里简化处理，实际需要根据消息类型分发
        RequestVoteRequest req;
        if (req.ParseFromString(data)) {
            // 处理RequestVote
            RequestVoteResponse resp = handle_request_vote(req);

            // 发送响应
            send_message(fd, resp);
        }
    }

    RequestVoteResponse handle_request_vote(const RequestVoteRequest& req) {
```

```
75          // 实际的Raft逻辑
76          RequestVoteResponse resp;
77          resp.set_term(req.term());
78          resp.set_vote_granted(true);
79          return resp;
80      }
81
82      void send_message(int fd, const google::protobuf::Message& msg) {
83          std::string data;
84          msg.SerializeToString(&data);
85
86          uint32_t len = htonl(data.size());
87          send(fd, &len, sizeof(len), 0);
88          send(fd, data.c_str(), data.size(), 0);
89      }
90  };
```

关键技术点：

1. 长度前缀：解决 TCP 粘包问题
2. **protobuf** 序列化：跨语言、高效
3. 缓冲区管理：每个连接维护独立缓冲区
4. **epoll** 事件驱动：高并发 RPC

8 常见面试问题总结

8.1 TCP 相关

1. TCP 三次握手，为什么不是两次或四次？

- 为什么不是两次：无法确认客户端的接收能力，无法防止旧连接请求到达
- 为什么不是四次：第二次握手可以同时发送 SYN 和 ACK，没必要分开

2. TIME_WAIT 状态的作用？过多怎么办？

- 作用：确保最后一个 ACK 送达；让旧连接的报文消失（2MSL）
- 解决：调整内核参数（tw_reuse）、SO_REUSEADDR、让客户端主动关闭

3. CLOSE_WAIT 过多的原因？

- 收到 FIN 后没有调用 close()关闭 fd
- 检查代码逻辑，确保异常路径也关闭连接

4. TCP 如何保证可靠性？

- 序列号和确认号
- 超时重传
- 滑动窗口（流量控制）
- 拥塞控制（慢启动、拥塞避免、快速重传、快速恢复）

5. TCP 粘包问题如何解决？

- 固定长度
- 分隔符
- 长度前缀（推荐）
- 应用层协议（HTTP、protobuf）

8.2 epoll 相关

1. epoll 为什么比 select/poll 高效？

- 内核维护红黑树，不需要每次复制 fd 集合
- 就绪队列，只返回就绪的 fd，不需要遍历
- 时间复杂度 O(1) vs O(n)

2. epoll 的 LT 和 ET 模式有什么区别？

- **LT**（水平触发）：只要有事件就通知，可以分次读取
- **ET**（边缘触发）：状态变化时通知一次，必须一次读完（循环读到 EAGAIN）

3. ET 模式为什么必须配合非阻塞 I/O？

- ET 需要循环读取直到 EAGAIN
- 如果是阻塞 I/O，最后一次 read 会阻塞（没有数据了）
- 非阻塞 I/O 会返回 EAGAIN，告诉你读完了

4. epoll_wait 返回后，一定能读到数据吗？

- 不一定！ET 模式下，其他线程可能已经读走数据
- 需要配合 EPOLLONESHOT 或加锁

8.3 Reactor 相关

1. 什么是 Reactor 模式？

- 事件驱动的并发 I/O 模式
- 将 I/O 事件等待和处理分离
- 使用 I/O 多路复用（epoll）监听事件，回调处理

2. Single Reactor vs Multi Reactor？

- **Single Reactor**：一个线程处理所有 I/O，可能成为瓶颈
- **Multi Reactor**：主 Reactor 负责 accept，多个子 Reactor 负责 I/O，更高性能

3. Reactor vs Proactor？

- **Reactor**：同步非阻塞，应用程序负责读写数据
- **Proactor**：异步 I/O，内核负责读写数据（Windows IOCP、Linux io_uring）

8.4 高性能技术

1. 什么是零拷贝？你的项目用了吗？

- 减少数据在内核空间和用户空间之间的复制
- **sendfile**：文件->socket，减少到 2 次复制
- 项目中使用 **sendfile** 进行大文件传输

2. 什么是惊群效应？如何解决？

- 多个进程/线程监听同一个 socket，新连接到达时所有都被唤醒，但只有一个成功
- 解决：SO_REUSEPORT（内核负载均衡）、独立 epoll fd、加锁

3. 如何处理 **SIGPIPE** 信号？

- 向已关闭的 socket 写数据会收到 SIGPIPE，默认终止进程
- 解决：全局忽略 signal(SIGPIPE, SIG_IGN)，或使用 MSG_NOSIGNAL

8.5 项目相关（针对你的简历）

1. 你的高并发文件传输系统架构是怎样的？

- **Single Reactor +** 线程池
- 主线程：epoll 监听连接和请求
- 工作线程池：处理文件传输
- sendfile 零拷贝传输
- 异步日志（双缓冲）

2. 为什么选择 **Reactor +** 线程池而不是 **Multi Reactor**？

- 业务场景是文件传输，计算密集度不高
- Reactor + 线程池足够支撑 6000+ QPS
- 实现相对简单，便于调试和维护

3. 如何保证 **P95** 延迟小于 **15ms**？

- epoll ET 模式，减少系统调用
- 非阻塞 I/O，避免阻塞
- Buffer 对象池，减少内存分配
- TCP 缓冲区调优
- CPU 亲和性绑定

4. 异步日志的双缓冲如何实现？

- 前端缓冲区：业务线程无锁写入
- 后端缓冲区：日志线程批量刷盘
- 条件变量：前端满时通知后端
- swap 交换缓冲区，实现零拷贝切换

5. Raft 项目中如何解决 **TCP** 粘包问题？

- 使用长度前缀协议：[4 字节长度][protobuf 消息]
- 维护接收缓冲区，解析完整消息

- protobuf 自带序列化和边界处理

6. 文件传输如何优化性能？

- **sendfile** 零拷贝：避免用户空间复制
- 增大 **TCP** 缓冲区：提高吞吐量
- 非阻塞 **I/O**：避免阻塞等待
- 对象池：复用 Buffer，减少内存分配

7. 如何处理大量并发连接？

- epoll 高效监听（O(1)复杂度）
- 非阻塞 I/O + ET 模式
- 线程池复用，避免频繁创建销毁
- 连接超时管理（心跳、keepalive）

8. 如何定位性能瓶颈？

- **perf**：CPU profiling，找到热点函数
- **valgrind**：内存泄漏和缓存性能
- **GDB**：多线程竞态调试
- **wrk**：压力测试，验证 QPS 和延迟

8.6 深度问题

1. listen 的 **backlog** 参数是什么？

- 全连接队列（已完成三次握手）的最大长度
- 超过 backlog 后，新连接会被拒绝或延迟
- SYN 队列（半连接队列）是另一个队列

2. send/recv 的返回值有哪些情况？

- **> 0**：成功发送/接收的字节数（可能小于请求的大小）
- **== 0**：对端关闭连接（仅 recv）
- **< 0**：错误
  - EAGAIN/EWOULDBLOCK：非阻塞 I/O，暂时无数据
  - EINTR：被信号中断
  - EPIPE：对端关闭（send）

3. SO_REUSEADDR vs SO_REUSEPORT？

- **SO_REUSEADDR**：允许绑定处于 TIME_WAIT 的端口
- **SO_REUSEPORT**：允许多个 socket 绑定同一端口，内核负载均衡

4. TCP_NODELAY 的作用？

- 禁用 Nagle 算法
- Nagle：合并小包，减少网络包数量，但增加延迟
- 低延迟场景（游戏、实时通信）应该禁用

5. shutdown vs close？

- **shutdown**：关闭读/写方向，立即生效，即使有其他引用
- **close**：引用计数-1，为 0 时关闭

6. 为什么需要网络字节序转换？

- 不同 CPU 架构字节序不同（小端/大端）
- 网络协议统一使用大端序（网络字节序）
- htonl/htons：主机->网络，ntohl/ntohs：网络->主机

8.7 最终建议

面试准备：

1. 熟练掌握 epoll 的使用和原理
2. 理解 Reactor 模式，能手写简单版本
3. 深入了解你简历中项目的每一个技术点
4. 准备好性能指标的来源（如何压测？如何优化？）
5. 能画出系统架构图，讲清楚数据流

回答技巧：

1. 先说结论，再说原因
2. 结合项目实际经验
3. 对比不同方案的优缺点
4. 提到性能数据（QPS、延迟、吞吐）
5. 展示问题排查能力（GDB、perf、valgrind）

加分项：

1. 提到零拷贝、对象池等优化
2. 了解 Linux 内核相关知识（epoll 实现原理）
3. 阅读过优秀网络库源码（Muduo、libevent）
4. 能讲出遇到的 Bug 和解决过程
5. 了解现代技术（io_uring、DPDK）