第5课 网络编程基础

IP 和端口

IP 地址

什么是 IP 地址

- **定义**：IP 地址（Internet Protocol Address）是指分配给网络设备的唯一地址，用于标识网络中的每一台计算机或设备。

- **作用**：在网络通信中，IP 地址用于定位和识别设备，以确保数据能够准确地从源地址发送到目的地址。

IP 地址的分类

- **IPv4 地址**：

  - **格式**：32 位二进制数，通常表示为点分十进制，如 `192.168.1.1`。

  - **范围**：从 `0.0.0.0` 到 `255.255.255.255`。

  - **不足之处**：地址数量有限，难以满足日益增长的网络设备需求。

- **IPv6 地址**：

  - **格式**：128 位二进制数，表示为八组四位十六进制数，如 `2001:0db8:85a3:0000:0000:8a2e:0370:7334`。

  - **优势**：地址空间巨大，能够满足未来互联网发展的需要。

公有 IP 和私有 IP

- **公有 IP 地址**：由互联网服务提供商（ISP）分配，全球唯一，可在互联网中访问。

- **私有 IP 地址**：用于局域网内部通信，不能直接通过互联网访问，需要通过网络地址转换（NAT）与外部通信。

IP 地址的获取方式

- **静态 IP**：手动配置，地址固定不变，适用于服务器等需要固定地址的设备。

- **动态 IP**：通过 DHCP（动态主机配置协议）自动获取，地址可能会变化，适用于普通用户设备。

端口号

什么是端口号

- **定义**：端口号是用于标识计算机上特定进程或网络服务的数字，范围从 0 到 65535。

- **作用**：通过端口号，操作系统能够将收到的数据包准确地交给对应的应用程序。

端口号的分类

- **知名端口（Well-known Ports）**：0 ~ 1023，通常分配给系统或知名网络服务，例如：

- HTTP：80
- HTTPS：443
- FTP：21
- SSH：22

- **注册端口（Registered Ports）**：1024 ~ 49151，可用于用户或特定服务。

- **动态端口（Dynamic Ports）**：49152 ~ 65535，通常用于客户端动态分配。

注意事项

- 同一台计算机上，同一时间，一个端口只能被一个应用程序使用。
- 尽量避免使用知名端口作为自定义服务端口，防止冲突。

网络模型

网络模型用于描述网络通信的分层结构，帮助我们理解复杂的网络协议和通信过程。

OSI 七层模型

OSI（开放式系统互联）模型将网络通信分为七个层次：

1. **物理层（Physical Layer）**
   - **功能**：传输比特流（0 和 1），定义物理设备标准，如电压、电缆类型、传输速率。
   - **示例**：网线、集线器、光纤。
2. **数据链路层（Data Link Layer）**
   - **功能**：将比特流组织成数据帧，进行物理地址（MAC 地址）寻址，提供错误检测。
   - **示例**：网卡驱动、交换机。
3. **网络层（Network Layer）**
   - **功能**：负责逻辑地址（IP 地址）寻址和路由选择，实现网络间的数据传输。
   - **示例**：IP 协议、路由器。
4. **传输层（Transport Layer）**

- **功能**：提供端到端的可靠或不可靠传输服务，数据传输的错误检测和恢复。
- **示例**：TCP、UDP 协议。

5. **会话层（Session Layer）**
   - **功能**：管理通信会话，建立、维护和终止会话。
   - **示例**：RPC、SQL 会话。
6. **表示层（Presentation Layer）**
   - **功能**：数据的格式化、加密、解密、压缩。
   - **示例**：JPEG、MPEG、SSL/TLS。
7. **应用层（Application Layer）**
   - **功能**：为应用程序提供网络服务。
   - **示例**：HTTP、FTP、SMTP。

TCP/IP 四层模型

TCP/IP 模型是互联网中实际使用的模型，更加简化实用，将通信过程分为四个层次：

1. **链路层（Link Layer）**
   - **对应 OSI 的物理层和数据链路层**。
   - **功能**：处理硬件设备和数据帧传输。
   - **示例**：以太网、Wi-Fi。
2. **网络层（Internet Layer）**
   - **对应 OSI 的网络层**。
   - **功能**：处理 IP 地址寻址和路由。
   - **示例**：IP 协议、ICMP。
3. **传输层（Transport Layer）**
   - **对应 OSI 的传输层**。
   - **功能**：提供端到端的数据传输服务。
   - **示例**：TCP、UDP 协议。
4. **应用层（Application Layer）**
   - **对应 OSI 的会话层、表示层、应用层**。
   - **功能**：为应用程序提供网络服务。
   - **示例**：HTTP、FTP、DNS。

两种模型的比较

- **OSI 模型**更注重理论，提供了一个全面的网络通信框架。

- **TCP/IP 模型**更加实际，直接对应互联网的实际协议和应用。

协议

TCP（Transmission Control Protocol）

- **特点**：

- 面向连接：通信前需要建立连接（三次握手）。
- 可靠传输：提供数据确认、重传机制，保证数据不丢失、不重复。
- 字节流传输：数据以字节流的形式传输，没有明确的消息边界。

- **应用场景**：文件传输（FTP）、邮件（SMTP）、网页浏览（HTTP/HTTPS）。

UDP（User Datagram Protocol）

- **特点**：

- 无连接：不需要建立连接，直接发送数据。
- 不可靠传输：不保证数据到达，不提供重传机制。
- 数据报传输：以独立的数据报形式传输，有明确的消息边界。

- **应用场景**：视频直播、在线游戏、语音通话、DNS 查询。

TCP 与 UDP 的对比

特性	TCP	UDP
是否面向连接	是	否
传输可靠性	可靠	不可靠
传输方式	面向字节流	面向数据报
速度	较慢（需连接和确认）	快速
资源消耗	较高（需要维护状态）	较低
应用场景	文件传输、网页浏览等	视频、音频、游戏等

字节序

大端字节序（Big Endian）

- **定义**：高位字节存储在内存的低地址处，低位字节存储在高地址处。

- **形象比喻**：数据的高位在左边，像阅读书本一样从左到右。

小端字节序（Little Endian）

- **定义**：低位字节存储在内存的低地址处，高位字节存储在高地址处。

- **形象比喻**：数据的低位在左边，与大端相反。

网络字节序

- **定义**：为了在不同字节序的主机之间正确传输数据，网络通信统一采用大端字节序。

- **转换函数**：

  - `htonl`：将 32 位主机字节序转换为网络字节序。

  - `htons`：将 16 位主机字节序转换为网络字节序。

  - `ntohl`：将 32 位网络字节序转换为主机字节序。

  - `ntohs`：将 16 位网络字节序转换为主机字节序。

为什么需要统一字节序

- **原因**：不同计算机架构可能采用不同的字节序，如果不统一，会导致数据解析错误。

- **解决方案**：在网络通信中，发送方将数据转换为网络字节序，接收方再转换回主机字节序。

IP 操作函数

在网络编程中，需要将 IP 地址在字符串和数值之间进行转换，常用的函数有 `inet_pton` 和 `inet_ntop`。

`inet_pton` 函数

- **作用**：将点分十进制的 IP 地址字符串转换为网络字节序的数值形式。

- **原型**：

```
int inet_pton(int af, const char *src, void *dst);
```

- **参数**：

- `af`：地址族，`AF_INET` 表示 IPv4，`AF_INET6` 表示 IPv6。
- `src`：点分十进制的 IP 地址字符串。
- `dst`：指向存储结果的内存地址。

- **返回值**：

- 成功：返回 1。
- 失败：返回 0（无效地址）或 -1（系统错误）。

`inet_ntop` 函数

- **作用**：将网络字节序的 IP 地址数值转换为点分十进制的字符串形式。

- **原型**：

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

- **参数**：

- `af`：地址族，`AF_INET` 或 `AF_INET6`。
- `src`：指向网络字节序的 IP 地址数值。
- `dst`：用于存储结果字符串的缓冲区。
- `size`：缓冲区大小，IPv4 通常为 `INET_ADDRSTRLEN`（16）。

- **返回值**：

- 成功：返回指向结果字符串的指针。
- 失败：返回 `NULL`。

示例代码

下面是一个完整的示例，演示如何使用 `inet_pton` 和 `inet_ntop` 进行 IP 地址的转换。

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

int main() {
    // 点分十进制 IP 地址字符串
    const char* ip_str = "192.168.1.1";
    struct in_addr addr;

    // 将字符串转换为网络字节序的数值
    if (inet_pton(AF_INET, ip_str, &addr) == 1) {
        std::cout << "转换后的网络字节序数值: " << addr.s_addr << std::endl;
    } else {
        std::cerr << "无法将 IP 地址转换为数值形式" << std::endl;
        return 1;
    }

    // 用于存储转换后的字符串
    char ip_buffer[INET_ADDRSTRLEN];

    // 将网络字节序的数值转换回字符串
    if (inet_ntop(AF_INET, &addr, ip_buffer, INET_ADDRSTRLEN) != nullptr) {
        std::cout << "转换回的点分十进制 IP 地址: " << ip_buffer << std::endl;
    } else {
        std::cerr << "无法将数值形式转换回 IP 地址字符串" << std::endl;
        return 1;
    }

    return 0;
}
```


运行结果：

```
转换后的网络字节序数值: 16885952
转换回的点分十进制 IP 地址: 192.168.1.1
```

Socket 基础

Socket 定义

什么是 Socket

- **定义**：Socket（套接字）是网络通信的端点，用于描述 IP 地址和端口，是应用程序与网络之间的接口。

- **作用**：通过 Socket，应用程序可以向网络中发送和接收数据，实现进程间的通信。

Socket 的类型

- **流式套接字（SOCK_STREAM）**：用于面向连接的 TCP 通信，提供可靠的数据传输。

- **数据报套接字（SOCK_DGRAM）**：用于无连接的 UDP 通信，不保证数据可靠性。

Socket 通信的基本流程

1. **创建套接字**：调用 `socket()` 函数，指定地址族、套接字类型和协议。
2. **绑定地址**（服务器端）：调用 `bind()` 函数，将套接字绑定到特定的 IP 地址和端口号。
3. **监听连接**（服务器端）：调用 `listen()` 函数，等待客户端的连接请求。
4. **接受连接**（服务器端）：调用 `accept()` 函数，建立与客户端的连接。
5. **连接服务器**（客户端）：调用 `connect()` 函数，向服务器发起连接请求。
6. **数据传输**：使用 `send()`、`recv()` 或 `write()`、`read()` 进行数据发送和接收。
7. **关闭套接字**：调用 `close()` 函数，关闭连接。

`Sockaddr` 数据结构

在 Socket 编程中，需要使用特定的数据结构来表示地址信息。

通用地址结构 `sockaddr`

```
struct sockaddr {
    sa_family_t sa_family;   // 地址族，如 AF_INET、AF_INET6
    char        sa_data[14]; // 地址数据，具体内容与地址族相关
};
```

- **说明**：`sockaddr` 是一个通用的地址结构，适用于多种协议，但使用起来不方便。

IPv4 地址结构 `sockaddr_in`

```
struct sockaddr_in {
    sa_family_t    sin_family; // 地址族，必须设置为 AF_INET
    in_port_t      sin_port;   // 16 位端口号，需要使用 `htons()` 转换为网络字节序
    struct in_addr sin_addr;   // 32 位 IP 地址，使用 `inet_pton()` 设置
    char           sin_zero[8];// 填充字段，必须设置为 0
};
```

- **`sin_family`**：地址族，IPv4 使用 `AF_INET`。
- **`sin_port`**：端口号，注意转换字节序。
- **`sin_addr`**：IP 地址，可以使用 `INADDR_ANY` 绑定所有本地地址。

IPv6 地址结构 `sockaddr_in6`

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;   // 地址族，AF_INET6
    in_port_t       sin6_port;     // 端口号，网络字节序
    uint32_t        sin6_flowinfo; // 流信息，通常为 0
    struct in6_addr sin6_addr;     // IPv6 地址
    uint32_t        sin6_scope_id; // 作用域 ID，通常为 0
};
```


地址结构的使用

- 在调用 Socket 函数时，需要将具体的地址结构转换为通用的 `sockaddr` 结构。

```
struct sockaddr_in addr;
// ... 初始化 addr ...
bind(sock_fd, (struct sockaddr*)&addr, sizeof(addr));
```

TCP 和 UDP 的区别

TCP

- **面向连接**：通信前需要建立连接。

- **可靠性**：提供确认机制，确保数据正确传输。

- **数据传输方式**：面向字节流，无明确的消息边界。

- **资源消耗**：需要维护连接状态，占用系统资源。

UDP

- **无连接**：无需建立连接，直接发送数据。

- **不可靠性**：不保证数据到达，不提供重传。

- **数据传输方式**：面向数据报，有明确的消息边界。

- **资源消耗**：无需维护连接，资源占用少。

选择使用 TCP 或 UDP

- **TCP 适用场景**：对数据可靠性要求高的应用，如文件传输、邮件、远程登录。

- **UDP 适用场景**：对实时性要求高但可容忍少量丢包的应用，如视频直播、在线游戏、语音通话。

TCP 通信

TCP 通信流程

服务器端

1. **创建套接字**：`socket()`
   - 指定地址族 `AF_INET`、套接字类型 `SOCK_STREAM`、协议 `0`。
2. **绑定地址**：`bind()`
   - 将套接字绑定到指定的 IP 地址和端口号。
3. **监听连接**：`listen()`
   - 开始监听客户端的连接请求，指定最大连接数。
4. **接受连接**：`accept()`
   - 阻塞等待客户端的连接请求，返回一个新的套接字用于通信。

客户端

1. **创建套接字**：`socket()`
   - 与服务器端相同。
2. **连接服务器**：`connect()`
   - 指定服务器的 IP 地址和端口号，发起连接请求。

数据传输

- **发送数据**：`send()` 或 `write()`

- **接收数据**：`recv()` 或 `read()`

关闭连接

- **关闭套接字**：`close()`

TCP 三次握手

建立 TCP 连接的过程，确保双方都准备好进行通信。

第一次握手

- **客户端**：发送 SYN 包（同步序列号），请求建立连接。

- **包含信息**：客户端的初始序列号（ISN）。

第二次握手

- **服务器**：收到 SYN 包，发送 SYN-ACK 包，表示同意连接。

- **包含信息**：服务器的 ISN，确认序列号（客户端 ISN + 1）。

第三次握手

- **客户端**：收到 SYN-ACK 包，发送 ACK 包，确认连接建立。

- **包含信息**：确认序列号（服务器 ISN + 1）。

连接建立

- **状态**：双方进入 ESTABLISHED 状态，开始数据传输。

TCP 滑动窗口

流量控制机制，用于控制发送方的发送速率，确保接收方有足够的缓冲区处理数据。

流量控制

- **目的**：防止发送方发送过快，接收方处理不过来，导致数据丢失。

滑动窗口机制

- **发送窗口**：发送方维护，表示允许发送但未确认的数据量。

- **接收窗口**：接收方维护，表示可接收数据的缓冲区大小。

- **动态调整**：根据网络状况和接收方反馈，调整窗口大小。

工作原理

1. **发送数据**：发送方根据窗口大小发送数据，不必等待每个数据的确认。

2. **接收确认**：接收方接收数据后，发送 ACK 确认，并告知可用窗口大小。

3. **窗口滑动**：收到 ACK 后，发送方窗口向前滑动，继续发送新的数据。

优点

- 提高网络吞吐量，充分利用带宽。
- 避免网络拥塞，提高传输效率。

TCP 四次挥手

终止 TCP 连接的过程，确保双方都同意关闭连接。

第一次挥手

- **主动关闭方（通常是客户端）**：发送 FIN 包，表示不再发送数据。

- **状态变为**：FIN_WAIT_1。

第二次挥手

- **被动关闭方（通常是服务器）**：收到 FIN 包，发送 ACK 包，确认收到关闭请求。

- **状态变为**：CLOSE_WAIT。

第三次挥手

- **被动关闭方**：处理完剩余数据后，发送 FIN 包，表示同意关闭连接。

- **状态变为**：LAST_ACK。

第四次挥手

- **主动关闭方**：收到 FIN 包，发送 ACK 包，确认连接关闭。

- **状态变为**：TIME_WAIT，等待一段时间后进入 CLOSED。

为什么需要四次挥手

- **原因**：TCP 是全双工通信，发送和接收独立进行，双方需要分别关闭发送和接收通道。

- **确保双方都已完成数据传输，避免数据丢失。**

TCP 通信并发处理

实现高并发的网络服务器，需要有效地管理多个客户端连接。

多进程模型

- **原理**：为每个客户端连接创建一个新的进程。

- **优点**：进程隔离性强，安全性高。

- **缺点**：系统开销大，创建进程代价高。

多线程模型

- **原理**：为每个客户端连接创建一个新的线程。

- **优点**：比多进程开销小，共享内存空间。

- **缺点**：需要注意线程同步，可能出现竞争条件。

I/O 多路复用

- **原理**：使用 `select`、`poll`、`epoll` 等系统调用，一个进程同时管理多个网络连接。

- **优点**：高效处理大量并发连接，资源占用少。

- **缺点**：编程复杂度较高，需要管理事件和状态。

TCP 状态转换

描述 TCP 连接的不同阶段，包括以下状态：

常见状态解释

- **LISTEN**：服务器监听状态，等待连接请求。

- **SYN_SENT**：客户端发送 SYN 后的状态，等待服务器的 SYN-ACK。

- **SYN_RECEIVED**：服务器收到 SYN 后发送 SYN-ACK，等待客户端的 ACK。

- **ESTABLISHED**：连接已建立，双方可通信。

- **FIN_WAIT_1**：主动关闭方发送 FIN 后的状态，等待对方的 ACK。

- **FIN_WAIT_2**：收到对方的 ACK 后，等待对方的 FIN。

- **CLOSE_WAIT**：被动关闭方收到 FIN 后的状态，等待应用程序处理完数据并发送 FIN。

- **CLOSING**：双方同时发送 FIN 的状态。

- **LAST_ACK**：被动关闭方发送 FIN 后，等待对方的 ACK。

- **TIME_WAIT**：主动关闭方等待一段时间，确保对方收到 ACK。

- **CLOSED**：连接关闭，无任何连接信息。

半关闭

半关闭：TCP 连接的一方完成数据发送后，可以关闭发送方向，但仍然可以接收数据。

函数：`shutdown(socket, how)`

- `how` 参数：
  - `SHUT_RD`：关闭接收通道。
  - `SHUT_WR`：关闭发送通道。
  - `SHUT_RDWR`：同时关闭发送和接收通道。

应用场景

- **长连接中**：一方发送完数据后，通知对方自己不再发送数据，但仍需接收对方的数据。
- **节省资源**：及时关闭不必要的通道，减少资源占用。

端口复用的实现与应用

端口复用

定义：允许多个套接字绑定到同一 IP 地址和端口号的机制。

实现方式

- `SO_REUSEADDR`：
  - **作用**：允许在套接字关闭后立即重用地址。
  - **使用场景**：服务器重启时，端口尚未释放，设置该选项可以立即绑定。

- `SO_REUSEPORT`：
  - **作用**：允许多个进程或线程绑定到同一 IP 和端口，实现负载均衡。
  - **使用条件**：需要内核和库的支持（Linux 内核 3.9 及以上）。

使用方法

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

- **参数说明**：

- `sock_fd`：套接字描述符。
- `SOL_SOCKET`：套接字层级。
- `SO_REUSEADDR`：选项名称。
- `&opt`：选项值，`1` 表示启用。

I/O 多路复用

定义：一种允许一个进程同时等待多个文件描述符的机制，当其中一个或多个描述符就绪时，操作系统通知进程进行相应的 I/O 操作。

常用系统调用

- `select`：
  - **特点**：跨平台，但对文件描述符数量有限制（通常为 1024）。

- `poll`：
  - **特点**：没有描述符数量限制，使用链表存储，但在大量描述符时效率下降。

- `epoll`：
  - **特点**：Linux 特有，使用事件驱动，效率高，适用于大并发场景。

本地套接字（UNIX 套接字）

定义：用于同一主机上进程间通信的套接字，不经过网络层。

特点

- **高效性**：数据不经过网络协议栈，传输速度快。
- **安全性**：只能在本地使用，不会被网络中其他主机访问。

使用场景

- **本地进程通信**：如系统日志服务、数据库连接。

编程示例

```
struct sockaddr_un {
    sa_family_t sun_family;   // 地址族，AF_UNIX 或 AF_LOCAL
    char        sun_path[108];// 文件系统路径
};
```

- **注意**：使用时需要指定套接字类型为 `AF_UNIX`。

网络编程面试常问问题

1. **TCP 和 UDP 的区别是什么？**
   - **TCP**：面向连接，提供可靠的数据传输，有流量控制和拥塞控制。
   - **UDP**：无连接，不保证数据可靠性，传输速度快。

2. **描述 TCP 三次握手和四次挥手的过程。**
   - **三次握手**：建立连接，双方同步序列号。
   - **四次挥手**：关闭连接，双方确认数据发送完毕。

3. **什么是 TIME_WAIT 状态，为什么需要等待 2MSL？**
   - **TIME_WAIT**：主动关闭方等待时间，确保对方收到 ACK，防止旧连接的数据混淆新的连接。

4. **如何实现高并发的网络服务器？**
   - **方法**：使用 I/O 多路复用、线程池、事件驱动模型（如 Reactor、Proactor）。

5. **解释什么是端口复用，如何使用？**
   - **定义**：允许多个套接字绑定同一端口。
   - **使用**：设置套接字选项 `SO_REUSEADDR` 或 `SO_REUSEPORT`。

6. **什么是半关闭，在哪些情况下使用？**

- **定义**：关闭套接字的发送或接收方向。
- **使用场景**：一方不再发送数据，但需要继续接收。

7. **描述 epoll 的工作模式。**

- **模式**：水平触发（LT）和边缘触发（ET）。
- **区别**：
  - **LT**：只要文件描述符有事件未处理，会持续通知。
  - **ET**：只在状态变化时通知，提高效率。

8. **字节序的重要性，如何处理不同字节序的主机通信？**

- **重要性**：确保数据在不同架构的主机间正确解析。
- **处理方法**：使用网络字节序，大端模式。

9. **socket 的常用系统调用有哪些？**

- **创建套接字**：`socket()`
- **绑定地址**：`bind()`
- **监听连接**：`listen()`
- **接受连接**：`accept()`
- **连接服务器**：`connect()`
- **发送数据**：`send()`
- **接收数据**：`recv()`
- **关闭套接字**：`close()`

10. **什么是粘包和拆包，如何解决？**

- **粘包**：多个小数据包粘在一起发送或接收。
- **拆包**：一个大数据包被拆分为多个小包。
- **解决方法**：添加消息边界，使用定长消息，或在消息前添加长度信息。