TCP Socket 编程基础

Goals:

用 C++ 实现一个最基础的 TCP 服务端，它能

- 创建 1 个 socket
- 绑定到本地 IP 和端口
- 监听客户端连接请求（不接受连接，只监听）
- 保持运行，准备进入 epoll 的下一步

① 创建套接字（create a socket）

```
int socket(int domain, int type,
           int protocol);
```

- domain：地址族，常用 AF_INET（IPV4）
- type：套接字类型，TCP 使用 SOCK_STREAM
- protocol：常设为 0（系统自动选择）

解释：使用 socket() 创建 1 个 TCP socket.
像插了根网线准备通信

② 设置本地地址

(prepare local address structure)

需要用1个 sockaddr_in 结构体来监听的地址和端口：

```
struct sockaddr_in
{
    short sin_family; //地址族 AF_INET
    unsigned short sin_port; //端口(使用htons())
    struct in_addr sin_addr; // IP
    char sin_zero[8]; //填充
};
sin_family = AF_INET;
sin_port = htons(12345); //将端口号转为网络字节序
sin_addr.s_addr = INADDR_ANY;
                  //任何本地地址都接受
                  默认
```

sockaddr_in 是对 IPV4 地址的结构表示，系统通过它知道你要在哪个IP和端口监听。

③ 绑定 socket (Bind the socket to address)

```
int bind(int socketfd, Const struct sockaddr *addr,
         Socklen_t addrlen);
```

- sockfd：第一步 socket() 返回的描述符
- addr：强制转换为 (struct sockaddr*)
- addrlen：使用 sizeof(struct sockaddr_in)

bind() 相当于告诉系统："我的 socket 负责监听哪个门牌号(端口)"

④ 设置监听状态 (start listening)

```
int listen(int sockfd, int backlog);
```

- sockfd：socket 描述符
- backlog：同一时间最多允许排队等待连接的数量 (常设为 5, 10)

解释：

listen() 把 socket 设置为"被动模式"

最终结构：基本的监听流程

//伪代码

```
int server_fd = socket(...); //创建socket
sockaddr_in addr = {...}; //设置地址结构体
bind(server_fd, ...); //绑定IP和端口
listen(server_fd, 10); //开始监听
```

调用 telnet localhost 12345 测试连接是否成功

非阻塞I/O 基础（Non-blocking I/O with fcntl()）

为什么要用非阻塞？（Why Non-blocking）

在默认情况下，Socket 是阻塞的，意味着：

如果你调用 accept() / recv()，而没有客户端连接或数据，它会卡住，程序不返回

但对于服务器：

- 主线程应不断地响应新的客户端
- 不应被某个客户端连接或读取卡死
- 所以我们需要设置 socket 为非阻塞模式

How？

使用 fcntl() 系统调用修改文件描述符的行为

```
int fcntl(int fd, int cmd, ...);
```

step1：获取原来的 flags

```
int flags = fcntl(fd, F_GETFL, 0);
```

fd：创建的 socket

F_GETFL：获取当前 flag

Step2：添加非阻塞标志位（O_NONBLOCK）

fcntl(fd, F_SETFL, flags | O_NONBLOCK);

F_SETFL：设置新 flag

O_NONBLOCK：是非阻塞标志位

用 flags | O_NONBLOCK 表示“保留原设置 + 新增非阻塞”

现在 socket 是非阻塞的了

epoll 机制基础

📌 什么是 epoll？（What is epoll?）

`epoll` 是 Linux 提供的高性能 I/O 事件通知机制。

它是 `select()` 和 `poll()` 的现代替代方案，专为**高并发网络服务**设计。

🚀 优点：

- 不再每次轮询所有文件描述符，只处理"活跃事件"
- 支持上万连接（select 上限是 1024）
- 内核和用户态共享事件集合（更高效）

🧠 epoll 基本使用三步法（Three Core Steps）

> 总结为：**创建、注册、等待**

🔹 1. 创建 epoll 实例（Create an epoll instance）

```c
int epfd = epoll_create1(0); // 推荐用 epoll_create1
```

- 返回值是 epoll 实例的文件描述符
- 参数可以设为 0，若失败返回 -1

🔹 2. 注册事件（Register socket events）

```c
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

参数含义：

参数	含义
`epfd`	第一步创建的 epoll 实例
`op`	操作类型，如 `EPOLL_CTL_ADD` 添加事件
`fd`	要监听的 socket（如监听 socket）
`event`	你关心的事件（如读、写）

🔹 事件结构体定义：

```c
struct epoll_event {
    uint32_t events; // EPOLLIN, EPOLLOUT 等
    int data;        // 你可以存 socket fd，或指针（union）
};
```

示例（监听读事件）：

```c
event.events = EPOLLIN;
event.data.fd = listen_fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, &event);
```

🔹 3. 等待事件触发（Wait for ready events）

```c
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

- `events` 是用来接收就绪事件的数组
- `maxevents` 是能处理的最多事件个数
- `timeout`：等待时间（ms），-1 表示无限等待

返回值：

- 返回就绪事件数量
- 小于 0 表示错误（`errno`）