Linux中的多路转接

1. Poll

`poll` 是 Linux I/O 多路复用的一种机制，用于同时监视多个文件描述符，检查它们是否可读、可写或是否出现异常。它是 `select` 的增强版，解决了 `select` 的一些根本性限制。

核心概念：I/O 多路复用

在网络编程或需要处理多个 I/O 源的场景中，为每个文件描述符创建一个线程使用阻塞 I/O 会消耗大量资源。I/O 多路复用允许单个线程/进程同时监视多个文件描述符的状态，当其中某个或多个文件描述符就绪（准备好进行 I/O 操作）时，再对其进行操作，从而极大地提高了程序的效率。

---

1. `poll` 系统调用

函数原型

代码块

```
#include <poll.h>

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

参数说明

1. **`fds`**： 指向一个 `struct pollfd` 数组的指针，每个数组元素代表一个被监视的文件描述符。
2. **`nfds`**： `fds` 数组中的元素个数，即要监视的文件描述符的数量。

3. **`timeout`**： 指定 `poll` 函数的超时时间（以毫秒为单位）。
    - **`-1`**： `poll` 将永远阻塞，直到某个被监视的文件描述符就绪。
    - **`0`**： `poll` 立即返回，无论文件描述符是否就绪（非阻塞轮询）。
    - **`> 0`**： `poll` 将阻塞最多 `timeout` 毫秒，如果在此期间没有文件描述符就绪，则超时返回。

返回值

- **`> 0`**： 成功，返回就绪的文件描述符的总数（即 `revents` 字段被设置为非零值的 `struct pollfd` 的数量）。
- **`0`**： 超时，在指定的 `timeout` 时间内没有任何文件描述符就绪。
- **`-1`**： 失败，并设置相应的 `errno`。

---

2. 关键数据结构 `struct pollfd`

`poll` 的核心是 `struct pollfd` 结构体，它告诉内核我们需要监视哪个文件描述符的什么事件。

代码块

```
struct pollfd {
    int   fd;         /* 要监视的文件描述符 */
    short events;     /* 我们关心的事件（输入） */
    short revents;    /* 实际发生的事件（输出） */
};
```

- **`fd`**： 要监视的文件描述符。如果不想监视某个元素，可以将其设置为负数（如 `-1`）。
- **`events`**： 一个位掩码，由应用程序在调用 `poll` **之前**设置，告诉内核“我们关心这个文件描述符的什么事件”。（例如：可读、可写）
- **`revents`**： 同样是一个位掩码，由内核在 `poll` **返回时**设置，告诉应用程序“这个文件描述符上实际发生了什么事件”。如果发生了某个事件，对应的位会被置 1。

常用的事件标志（`events` 和 `revents`）

事件标志	描述	是否可作为 `events` 输入	是否可作为 `revents` 输出
`POLLIN`	有数据可读（包括普通数据、带外数据，以及 TCP 连接的对端关闭）	是	是
`POLLPRI`	有紧急数据可读（例如 TCP 带外数据）	是	是
`POLLOUT`	文件描述符已准备好，可以写入数据而不被阻塞	是	是
`POLLRDHUP`	**（Linux 2.6.17+）** 流套接字的对端关闭了连接，或关闭了写入端	是	是
`POLLERR`	文件描述符上发生了错误	**否**	是
`POLLHUP`	文件描述符被挂起（例如，管道的写端被关闭后，读端将收到此事件）	**否**	是
`POLLNVAL`	文件描述符未打开（无效）	**否**	是

重要提示：

- `POLLERR`，`POLLHUP` 和 `POLLNVAL` 这三个事件**不能**在 `events` 字段中设置，它们只会在 `revents` 字段中由内核返回。
- 当 `POLLHUP` 或 `POLLERR` 被设置时，`poll` 会返回，即使你没有在 `events` 中请求这些事件。因此，在检查时，应该总是先检查错误事件。

---

3. 使用 `poll` 的基本流程

1. **初始化 `pollfd` 数组**：创建一个 `struct pollfd` 数组，并为每个需要监视的文件描述符设置 `fd` 和 `events` 字段。
2. **调用 `poll`**：在循环中调用 `poll` 函数。
3. **检查返回值**：
   - 如果返回 `-1`，检查 `errno` 处理错误（如果 `errno` 是 `EINTR`，表示被信号中断，通常可以继续）。
   - 如果返回 `0`，表示超时，可以进行一些其他操作或继续等待。
   - 如果返回大于 `0`，表示有就绪的文件描述符，进入下一步。
4. **遍历 `pollfd` 数组**：遍历数组，检查每个元素的 `revents` 字段。
   - 先检查 `POLLERR`，`POLLHUP`，`POLLNVAL` 等错误事件。
   - 然后检查你关心的事件，如 `POLLIN` 或 `POLLOUT`。
5. **处理就绪的文件描述符**：根据检测到的事件进行相应的 I/O 操作（`read`，`write`，`accept` 等）。
6. **重置或更新 `pollfd` 数组**：如果需要，可以修改 `pollfd` 数组（例如，移除已关闭的连接，添加新的连接），然后回到第 2 步。

---

4. 示例代码：使用 `poll` 处理标准输入和套接字

以下是一个简单的示例，监视标准输入（`stdin`）和一个 TCP 套接字。

代码块

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <poll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAX_BUFFER_SIZE 1024
```

```c
#define PORT 8080

int main() {
    int sockfd;
    struct sockaddr_in server_addr;
    struct pollfd fds[2];
    char buffer[MAX_BUFFER_SIZE];

    // 创建套接字
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr); // 连接到本地服务器

    // 连接服务器（这里仅为示例，假设服务器已运行）
    if (connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("connect failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("Connected to server...\n");

    // 初始化 pollfd 结构体数组
    fds[0].fd = STDIN_FILENO;      // 标准输入（文件描述符 0）
    fds[0].events = POLLIN;

    fds[1].fd = sockfd;            // 套接字
    fds[1].events = POLLIN;        // 监视套接字是否可读

    while (1) {
        // 调用 poll，无限期等待
        int ret = poll(fds, 2, -1);

        if (ret == -1) {
            perror("poll");
            break;
        } else if (ret == 0) {
            printf("Timeout occurred!\n");
            continue;
        }

```

```
57          // 检查哪个文件描述符就绪了
58          for (int i = 0; i < 2; i++) {
59              // 先检查错误
60              if (fds[i].revents & (POLLERR | POLLHUP | POLLNVAL)) {
61                  printf("File descriptor %d has an error or hung up.\n",
    fds[i].fd);
62                  if (fds[i].fd == sockfd) {
63                      // 服务器连接出问题，退出
64                      goto end;
65                  }
66              }
67
68              // 检查标准输入是否可读
69              if (fds[i].fd == STDIN_FILENO && (fds[i].revents & POLLIN)) {
70                  memset(buffer, 0, MAX_BUFFER_SIZE);
71                  if (fgets(buffer, MAX_BUFFER_SIZE, stdin) != NULL) {
72                      // 将输入发送给服务器
73                      write(sockfd, buffer, strlen(buffer));
74                  } else {
75                      // 读到 EOF (Ctrl+D)
76                      goto end;
77                  }
78              }
79
80              // 检查套接字是否可读（收到服务器消息）
81              if (fds[i].fd == sockfd && (fds[i].revents & POLLIN)) {
82                  memset(buffer, 0, MAX_BUFFER_SIZE);
83                  ssize_t n = read(sockfd, buffer, MAX_BUFFER_SIZE - 1);
84                  if (n > 0) {
85                      printf("Received from server: %s", buffer);
86                  } else if (n == 0) {
87                      // 服务器关闭了连接
88                      printf("Server closed the connection.\n");
89                      goto end;
90                  } else {
91                      perror("read");
92                      goto end;
93                  }
94              }
95          }
96      }
97
98  end:
99      close(sockfd);
100     return 0;
101 }
```

5. `poll` 与 `select` 的比较

特性	`select`	`poll`
文件描述符数量限制	有（`FD_SETSIZE`，通常 1024）	无（理论上只受系统资源限制）
效率	文件描述符数量大时，线性扫描效率低	文件描述符数量大时，线性扫描效率同样低
输入/输出参数	使用同一个 `fd_set` 参数，每次调用后会被内核修改，需要重置	使用独立的 `events` 和 `revents`，无需重置
可移植性	非常广泛	广泛，但不如 `select`
监视事件粒度	较粗（读、写、异常）	较细（更多事件类型，如 `POLLRDHUP`）

6. 更现代的替代品：`epoll`

虽然 `poll` 解决了 `select` 的文件描述符数量限制问题，但当监视的文件描述符数量非常大时，`poll` 和 `select` 一样，都需要在**每次调用时**将整个文件描述符集合从用户空间拷贝到内核空间，并且在返回时需要**线性扫描**整个集合来查找就绪的文件描述符。这导致了性能瓶颈。

Linux 提供了 `epoll` 来解决这个问题：

- **`epoll_create`**： 创建一个 `epoll` 实例。
- **`epoll_ctl`**： 向实例中添加、修改或删除要监视的文件描述符。
- **`epoll_wait`**： 等待事件发生，类似于 `poll`。

`epoll` 的优势在于：

1. **无需每次拷贝**： 文件描述符集合在内核中管理，只需在变化时通过 `epoll_ctl` 更新。
2. **高效事件通知**： `epoll_wait` 只返回就绪的文件描述符，无需线性扫描。

因此，在需要处理大量并发连接的场景下（如高性能网络服务器），`epoll` 是比 `poll` 和 `select` 更优的选择。

总结

- **`poll`** 是 `select` 的一个有效增强，突破了文件描述符数量的限制，并且使用了更清晰的事件分离设计。
- 它通过 `struct pollfd` 结构体来管理监视列表，使用 `events` 和 `revents` 分别表示关心的事件和实际发生的事件。
- 对于小到中等规模的并发连接，`poll` 是一个简单可靠的选择。
- 对于大规模并发应用（如 C10K 问题），应优先考虑使用性能更高的 `epoll`。

2. Epoll

Epoll 是 Linux 内核为处理大量文件描述符而设计的高性能 I/O 多路复用机制。它被设计用来克服 `select` 和 `poll` 在处理大规模并发连接时的性能瓶颈。

---

1. 为什么需要 Epoll？ `select` / `poll` 的瓶颈

要理解 Epoll 的优势，首先要明白 `select` 和 `poll` 的工作方式及其缺陷：

1. **每次调用都需要传递完整的文件描述符集合**

- 每次调用 `select` / `poll` 时，都需要将整个需要监视的文件描述符集合从用户空间拷贝到内核空间。当集合很大时，这会带来显著的开销。

2. **线性扫描检查状态**

- 在内核中，`select` / `poll` 通过**线性遍历**整个文件描述符集合来检查每个描述符的状态是否就绪。时间复杂度为 O(n)。

3. **返回后需要线性扫描找出就绪的描述符**

- 当 `select` / `poll` 返回时，它们只告诉用户"有描述符就绪了"，但没明确指出是哪些。应用程序必须再次**线性遍历**整个集合，通过检查 `revents` 或 `FD_ISSET` 来找出就绪的描述符。

结论：随着监视的文件描述符数量 `n` 的增长，`select` / `poll` 的性能会**线性下降**。这在处理成千上万个并发连接（C10K 问题）时是无法接受的。

---

2. Epoll 的核心思想与优势

Epoll 的核心设计在于 **"事件驱动"** 和 **"内核级状态管理"**。

1. **内核状态托管**：

- 首先，你通过 `epoll_create` 在内核中创建一个 "epoll 实例"。
- 然后，通过 `epoll_ctl` 将需要监视的文件描述符**注册**到这个实例中。这个注册操作是**一次性的**。之后，内核会维护这个描述符集合，无需在每次调用时重复拷贝。

2. **高效事件就绪列表**：

- 当被监视的文件描述符状态发生变化（就绪）时，内核会将其放入一个"就绪列表"中。
- 应用程序调用 `epoll_wait` 时，内核只需将这个就绪列表中的内容返回给应用程序，而无需扫描整个集合。

优势：

- **时间复杂度**：`epoll_wait` 的时间复杂度是 O(就绪的文件描述符数量)，而不是 O(总监视数量)。这在活跃连接率较低时（这是网络服务器的典型场景）效率极高。
- **无需重复拷贝**：大大减少了用户空间和内核空间之间的数据拷贝。

- **可扩展性**： 能够轻松处理数万甚至数十万的并发连接。

---

3. Epoll 的核心 API

Epoll 只有三个关键的系统调用。

3.1 `epoll_create` - 创建 Epoll 实例

代码块

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

int epoll_create(int size);
int epoll_create1(int flags); // 更现代的版本，推荐使用
```

- **`epoll_create1(int flags)`**：
  - `flags`： 通常设置为 `0`，或者 `EPOLL_CLOEXEC`（在 exec 时关闭文件描述符）。
  - **返回值**： 一个指向新创建的 epoll 实例的文件描述符。使用完毕后，应通过 `close()` 关闭。

> **注意**： 早期的 `epoll_create(size)` 中的 `size` 参数只是一个提示，告诉内核期望监视的文件描述符数量，内核会根据它进行初始分配。在现代内核中，这个参数已被忽略，但必须大于 0。推荐使用 `epoll_create1(0)`。

3.2 `epoll_ctl` - 管理监视列表

代码块

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

- **`epfd`**： 由 `epoll_create1` 返回的 epoll 实例描述符。
- **`op`**： 操作类型，指定要执行的动作：
  - `EPOLL_CTL_ADD`： 将新的文件描述符 `fd` 添加到 epoll 实例中。
  - `EPOLL_CTL_MOD`： 修改已注册的文件描述符 `fd` 相关联的事件。
  - `EPOLL_CTL_DEL`： 从 epoll 实例中删除文件描述符 `fd`。此时 `event` 参数可以为 `NULL`。
- **`fd`**： 要操作的目标文件描述符（例如，一个 socket）。
- **`event`**： 指向 `struct epoll_event` 的指针，描述了要监视的事件。

`struct epoll_event` 结构体

代码块

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

struct epoll_event
{
    uint32_t     events;    /* Epoll events (bit mask) */
    epoll_data_t data;      /* User data */
};
```

- **`events`**： 是一个位掩码，表示你关心的事件。常用事件有：
  - `EPOLLIN`： 对应的文件描述符可读。
  - `EPOLLOUT`： 对应的文件描述符可写。
  - `EPOLLET`： 设置边缘触发（Edge-Triggered）模式。默认为水平触发（Level-Triggered）。
  - `EPOLLONESHOT`： 一次性监视。该事件被处理一次后，描述符会被禁用，需要重新用 `EPOLL_CTL_MOD` 激活。

- `EPOLLRDHUP`： 流套接字的对端关闭了连接，或关闭了写入端。
- `EPOLLPRI`： 有紧急数据可读。
- `EPOLLERR` / `EPOLLHUP`： 错误和挂起事件。它们会被**自动监视**，即使你没有在 `events` 中指定。

- **`data`**： 一个联合体（union），用于在事件发生时携带用户数据。这是 Epoll 非常强大的一个特性。最常见的是：
  - `data.fd`： 存储文件描述符本身。
  - `data.ptr`： 存储一个自定义结构体的指针，通常包含文件描述符和其他上下文信息（如连接状态、缓冲区等），这在面向对象的网络编程中非常有用。

3.3 `epoll_wait` - 等待事件

代码块

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

- **`epfd`**： epoll 实例描述符。
- **`events`**： 一个由调用者分配的 `struct epoll_event` 数组。**内核会将就绪的事件填充到这个数组中**。
- **`maxevents`**： `events` 数组的大小，表示一次最多能获取多少个事件。
- **`timeout`**： 超时时间（毫秒）。 `-1` 表示阻塞等待， `0` 表示立即返回， `>0` 表示超时时间。
- **返回值**:
  - `>0`： 就绪的文件描述符数量，即 `events` 数组中有效元素的个数。
  - `0`： 超时。
  - `-1`： 出错。

关键点： `epoll_wait` 返回时， `events` 数组中已经全是就绪的描述符及其事件信息。应用程序可以直接遍历 `0` 到 `返回值-1` 这个范围，**无需扫描所有监视的描述符**。

---

4. Epoll 的工作模式：LT vs ET

这是 Epoll 的一个核心概念，理解它们至关重要。

4.1 水平触发（LT - Level-Triggered，默认模式）

- **行为：** 只要文件描述符处于就绪状态（例如，接收缓冲区不为空），`epoll_wait` 就会持续通知你。
- **优点：**
  - 编程更简单，不容易遗漏事件。
  - 可以任意决定何时、如何读取/写入数据。如果一次没处理完，下次调用 `epoll_wait` 会再次通知你。
- **缺点：** 可能会产生更多的系统调用。

示例： 假设 socket 接收缓冲区有 2KB 数据。

1. 调用 `epoll_wait`，它会返回，告知该 socket 可读。
2. 你只读取了 1KB 数据。
3. 再次调用 `epoll_wait`，它**会再次立即返回**，告诉你这个 socket 仍然可读，因为缓冲区里还有 1KB 数据。

4.2 边缘触发（ET - Edge-Triggered）

- **行为：** 只有当文件描述符状态**发生变化**时（例如，从不可读变为可读），`epoll_wait` 才会通知你一次。
- **优点：**
  - 减少了 `epoll_wait` 被重复触发的次数，性能更高。
- **缺点：**
  - 编程更复杂。当事件被通知时，**你必须一次性处理完所有可用数据**，因为除非状态再次发生变化，否则你不会再收到通知。如果处理不完全，剩余的数据将一直留在缓冲区中，导致程序"饿死"。

使用 ET 模式的准则：

1. 必须使用**非阻塞（non-blocking）** 文件描述符，因为我们在读取时不知道内部数据的多少，如果内部为空那么这个文件就阻塞了，只有设置非阻塞fd，在读完后再读会报错提醒。
2. 在 `read` 或 `write` 返回 `EAGAIN` 或 `EWOULDBLOCK` 之前，要持续进行读取或写入，直到数据耗尽。

示例（接上例）：

1. 调用 `epoll_wait`，它会返回，告知该 socket 可读。
2. 你必须循环调用 `read`，直到它返回 `-1` 并且 `errno` 被设置为 `EAGAIN`，这表明你已经读完了所有数据。
3. 如果你只读了 1KB 就停止了，那么剩下的 1KB 数据将留在缓冲区里，而 `epoll_wait` 不会再通知你，直到对端发送**新的**数据过来（导致状态再次变化）。

---

5. 完整示例：一个简单的 ET 模式 Echo 服务器

这个示例展示了如何使用 Epoll 的 ET 模式和非阻塞 socket 构建一个简单的 Echo 服务器。

代码块

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <errno.h>

#define MAX_EVENTS 10
#define PORT 8080
#define BUFFER_SIZE 1024

```

```c
// 设置文件描述符为非阻塞模式
int set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) return -1;
    return fcntl(fd, F_SETFL, flags | O_NONBLOCK);
}

int main() {
    int listen_sock, conn_sock, epoll_fd, nfds, i;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);
    struct epoll_event ev, events[MAX_EVENTS];
    char buffer[BUFFER_SIZE];

    // 创建监听 socket
    listen_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_sock == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // 设置 SO_REUSEADDR
    int optval = 1;
    setsockopt(listen_sock, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));

    // 绑定地址和端口
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);
    if (bind(listen_sock, (struct sockaddr*)&server_addr, sizeof(server_addr))
== -1) {
        perror("bind");
        close(listen_sock);
        exit(EXIT_FAILURE);
    }

    // 开始监听
    if (listen(listen_sock, SOMAXCONN) == -1) {
        perror("listen");
        close(listen_sock);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    // 创建 epoll 实例
```

```
62      epoll_fd = epoll_create1(0);
63      if (epoll_fd == -1) {
64          perror("epoll_create1");
65          close(listen_sock);
66          exit(EXIT_FAILURE);
67      }
68
69      // 将监听 socket 添加到 epoll，使用 ET 模式
70      ev.events = EPOLLIN | EPOLLET;
71      ev.data.fd = listen_sock;
72      if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_sock, &ev) == -1) {
73          perror("epoll_ctl: listen_sock");
74          close(listen_sock);
75          close(epoll_fd);
76          exit(EXIT_FAILURE);
77      }
78
79      // 主循环
80      while (1) {
81          nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
82          if (nfds == -1) {
83              perror("epoll_wait");
84              break;
85          }
86
87          for (i = 0; i < nfds; i++) {
88              // 1. 处理新的客户端连接
89              if (events[i].data.fd == listen_sock) {
90                  // 注意：因为监听 socket 是 ET 模式，必须循环 accept 直到 EAGAIN
91                  while (1) {
92                      conn_sock = accept(listen_sock, (struct
    sockaddr*)&client_addr, &addr_len);
93                      if (conn_sock == -1) {
94                          if (errno == EAGAIN || errno == EWOULDBLOCK) {
95                              // 已经 accept 完所有新连接
96                              break;
97                          } else {
98                              perror("accept");
99                              break;
100                         }
101                     }
102
103                     printf("New connection from %s:%d\n",
104                            inet_ntoa(client_addr.sin_addr),
    ntohs(client_addr.sin_port));
105
106                     // 设置新连接 socket 为非阻塞
```

```
107                     set_nonblocking(conn_sock);
108 
109                     // 将新连接添加到 epoll，监视读事件，并使用 ET 模式
110                     ev.events = EPOLLIN | EPOLLET | EPOLLRDHUP; // 也监视对端关闭
111                     ev.data.fd = conn_sock;
112                     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn_sock, &ev) ==
    -1) {
113                         perror("epoll_ctl: conn_sock");
114                         close(conn_sock);
115                     }
116                 }
117             }
118             // 2. 处理客户端数据
119             else {
120                 int client_fd = events[i].data.fd;
121 
122                 // 检查对端是否关闭连接
123                 if (events[i].events & (EPOLLRDHUP | EPOLLHUP)) {
124                     printf("Client %d disconnected.\n", client_fd);
125                     close(client_fd);
126                     continue;
127                 }
128 
129                 // 处理可读事件
130                 if (events[i].events & EPOLLIN) {
131                     // ET 模式，必须循环读取直到 EAGAIN
132                     ssize_t total_read = 0;
133                     while (1) {
134                         ssize_t n = read(client_fd, buffer + total_read,
    BUFFER_SIZE - total_read - 1);
135                         if (n > 0) {
136                             total_read += n;
137                             // 检查缓冲区是否已满
138                             if (total_read >= BUFFER_SIZE - 1) {
139                                 break;
140                             }
141                         } else if (n == 0) {
142                             // 对端关闭连接
143                             printf("Client %d closed the connection.\n",
    client_fd);
144                             close(client_fd);
145                             goto next_event;
146                         } else {
147                             if (errno == EAGAIN || errno == EWOULDBLOCK) {
148                                 // 数据已读完
149                                 break;
150                             } else {
```

```
                                perror("read");
                                close(client_fd);
                                goto next_event;
                            }
                        }
                    }

                    if (total_read > 0) {
                        buffer[total_read] = '\0';
                        printf("Received from client %d: %s", client_fd,
    buffer);

                        // 简单 Echo：将数据写回（这里为了简单，假设写操作一次完成）
                        // 在实际生产中，写操作也应使用 ET 模式循环写入。
                        write(client_fd, buffer, total_read);
                    }
                }
    next_event:
                ; // 空语句，用于标签后接语句
            }
        }
    }

    close(listen_sock);
    close(epoll_fd);
    return 0;
}
```

---

6. 总结

特性	`select` / `poll`	`epoll`
效率	连接数增加时，性能线性下降	高效，只关心就绪的描述符
描述符管理	每次调用传递整个集合	内核托管，一次性注册
触发模式	仅支持水平触发（LT）	支持 LT 和 ET
可扩展性		非常高，轻松应对 C10K+

	受 `FD_SETSIZE` 或系统资源限制	
编程复杂度	简单	稍复杂，尤其 ET 模式

选择建议：

- 对于需要监视的描述符数量少、连接不频繁的场景，`select` / `poll` 足够简单有效。
- 对于需要处理**成千上万并发连接**的高性能网络服务器，**Epoll** 是 Linux 平台上的不二之选。其 ET 模式结合非阻塞 I/O 能提供最高的性能。