## 整体流程图

1. **初始化**
   └─> 建立互斥锁、条件变量
   └─> 启动若干工作线程
2. **主线程推送任务**
   └─> lock → 插入任务链表 → signal → unlock
3. **工作线程接收任务**
   └─> lock
   └─> if 无任务则 wait
   └─> 取出任务并从链表移除
   → unlock
   → 执行 `task_func`
4. **销毁线程池**
   └─> 标记所有线程 `terminate=1`
   └─> broadcast → 所有线程 wakeup → 退出循环 → free 自身

---

**启动**

- 主线程初始化锁（如需）、将 `count=0`。
- 创建 10 个子线程，它们立即进入各自的自增循环。

**竞争与同步**

- 线程并发修改 `count`，依据所选同步方式：
  - **无锁**：会出现丢计数的竞态，`count` 值远低于预期。
  - **互斥/自旋锁**：保证完全正确，但开销最大，增速最慢。
  - **汇编原子操作**：在多核上性能通常优于互斥锁，自增正确且开销更低。

**打印观察**

- 主线程每秒读一次 `count` 并打印，可实时看到增速曲线和策略差异。

**结束**

- 主循环结束后进程退出，所有线程随进程终止。

1. **初始化**
   - `count = 0`，无需显式初始化锁。
2. **并发自增**
   - 10 个线程并发执行 CAS 自旋循环，在高竞争时会有多次失败重试，但不进入内核态。
3. **实时打印**
   - 主线程每秒读取并打印一次 `count`，可直观观察增速。
4. **结束**
   - 打印完成后主线程退出，进程终止。

---

## 5.1 新连接处理流程

1. `accept`：从监听套接字 `fd` 上接收新客户端连接，返回 `clientfd`。
2. **非阻塞**：`fcntl(clientfd, F_SETFL, O_NONBLOCK)`，避免单个客户端读写阻塞整个循环。
3. **地址重用**：`setsockopt(..., SO_REUSEADDR)`，快速重启服务时避免“地址已占用”问题。
4. **加入 epoll**：将 `clientfd` 也注册到 epoll，监听 `EPOLLIN | EPOLLET`——
   - `EPOLLIN`：可读事件。
   - `EPOLLET`（边缘触发 Edge-Triggered）：仅在状态变化时通知一次，需循环读取直到返回 `EAGAIN`。

## 5.2 客户端数据处理流程

1. `recv`：从客户套接字 `fd` 上尝试读取数据。
2. `len <= 0`：
   - `len == 0`：对端正常关闭。
   - `len < 0`：出错（包括 `EAGAIN`、网络故障等）。
     在这两种情况下，调用 `close(fd)` 并用 `epoll_ctl(..., EPOLL_CTL_DEL)` 将该描述符从 epoll 中移除。
3. `len > 0`：打印收到的数据内容、长度及套接字编号。

---

# 6. 整体执行架构

1. **启动阶段**
   - 主线程在多个端口上启动监听，并在 epoll 中注册所有监听套接字。
2. **事件循环**
   - `epoll_wait` 驱动：高效地检测“新连接到达”与“客户端数据可读”两类事件。
3. **新连接接入**
   - 对新客户端设置非阻塞，加入 epoll，并继续循环等待更多事件。
4. **数据收发**
   - 对客户端读事件立即处理，出错或关闭时清理资源。
5. **优势**
   - **单线程/多路复用**：主线程可同时管理上百/上千个套接字而无需为每个连接创建线程。
   - **高性能**：边缘触发（`EPOLLET`）配合非阻塞，减少内核—用户态切换与不必要的系统调用。

---

# 1. 头文件与宏定义

```
c复制编辑#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

- **网络相关**：`<sys/socket.h>`、`<arpa/inet.h>`、`<netinet/in.h>` 提供套接字与地址转换函数；
  `<netdb.h>` 提供 DNS 解析接口；
  `<fcntl.h>` 用于文件描述符控制（如非阻塞模式）；
  `<unistd.h>` 包含 `close`、`read`/`write`、`select` 等系统调用。

```
#define HTTP_VERSION    "HTTP/1.1"
#define CONNECTION_TYPE "Connection: close\r\n"
#define BUFFER_SIZE     4096
```

- **HTTP 常量**：指定使用 HTTP/1.1 协议，并在请求头中声明 `Connection: close`，表示请求结束后服务器会关闭连接。
- **BUFFER_SIZE**：每次 `recv` 最多读取的字节数。

---

# 2. DNS 解析：`host_to_ip`

```
char *host_to_ip(const char *hostname)
{
    struct hostent *host_entry = gethostbyname(hostname);
    if (host_entry != NULL) {
        return inet_ntoa(*(struct in_addr *)*host_entry->h_addr_list);
    }
    return NULL;
}
```

1. `gethostbyname`：向系统配置的 DNS 服务器请求，将域名解析为一组 IP 地址（网络字节序）。
2. `h_addr_list[0]`：取返回列表中的第一个地址。
3. `inet_ntoa`：将网络字节序的 `struct in_addr` 转为点分十进制字符串（静态缓冲区），便于后续 `connect` 使用。

---

# 3. 创建并连接套接字：`http_create_socket`

```
int http_create_socket(char *ip)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
    struct sockaddr_in sin = {0};
    sin.sin_family      = AF_INET;
    sin.sin_port        = htons(80);
    sin.sin_addr.s_addr = inet_addr(ip);

    if (connect(sockfd, (struct sockaddr *)&sin, sizeof(sin)) != 0) {
        perror("connect");
        close(sockfd);
        return -1;
    }
    // 切换到非阻塞模式
    fcntl(sockfd, F_SETFL, O_NONBLOCK);
    return sockfd;
}
```

1. `socket`：创建一个 TCP（`SOCK_STREAM`）套接字。
2. **填充** `sockaddr_in`：设置 IPv4、端口 80（HTTP 默认端口）、目标 IP。
3. `connect`：发起三次握手。失败时打印错误并关闭套接字。
4. `fcntl(…, O_NONBLOCK)`：将套接字切换到非阻塞模式，以便后续使用 `select` 判断何时可读。

## 4. 发送请求并接收响应：`http_send_request`

```
char * http_send_request(const char *hostname, const char *resource)
{
    // 1. DNS → IP，创建并连接非阻塞 socket
    char *ip     = host_to_ip(hostname);
    int   sockfd = http_create_socket(ip);
    if (sockfd < 0) return NULL;

    // 2. 构造 HTTP GET 报文
    char buffer[BUFFER_SIZE] = {0};
    sprintf(buffer,
            "GET %s %s\r\n"
            "Host: %s\r\n"
            "%s\r\n",
            resource, HTTP_VERSION,
            hostname, CONNECTION_TYPE);

    // 3. 发送请求
    send(sockfd, buffer, strlen(buffer), 0);

    // 4. 准备用 select 等待可读
    fd_set    fdread;
    FD_ZERO(&fdread);
    FD_SET(sockfd, &fdread);
    struct timeval tv = { .tv_sec = 5, .tv_usec = 0 };  // 最多等 5 秒

    // 5. 动态构造响应缓冲串
    char *result = malloc(1);
    result[0]    = '\0';

```

```
    while (1) {
        int sel = select(sockfd + 1, &fdread, NULL, NULL, &tv);
        if (sel <= 0 || !FD_ISSET(sockfd, &fdread)) {
            // timeout 或 出错
            printf("timeout or error\n");
            break;
        }
        memset(buffer, 0, BUFFER_SIZE);
        int len = recv(sockfd, buffer, BUFFER_SIZE, 0);
        if (len <= 0) {
            // 对端关闭或出错
            break;
        }
        // 扩容并拼接新数据
        size_t old_len = strlen(result);
        result = realloc(result, old_len + len + 1);
        memcpy(result + old_len, buffer, len);
        result[old_len + len] = '\0';
    }

    close(sockfd);
    return result;
}
```

## 4.1 构造与发送

- `sprintf`：将请求行、`Host` 头、`Connection` 头拼成完整报文，以 `\r\n\r\n` 结尾标志头部结束。
- `send`：一次性将整个请求报文发给服务器。

## 4.2 非阻塞 + `select`

- **为什么用非阻塞**：避免 `recv` 阻塞；配合 `select` 灵活控制超时。
- `FD_SET` + `select`：监视套接字何时有数据可读，超时后退出循环避免无限等待。

## 4.3 动态缓冲区

- **起始分配**：`malloc(1)`，并置空字符串。
- **每次 `recv` 后**：通过 `realloc` 扩展已有字符串，以容纳新读取的 `len` 字节，并手动拼接。
- **优点**：适应任意大小的响应，无须事先知道内容长度；
- **注意**：大量 `realloc` 会有性能开销，真实应用可按块（如每次 +4KB）增长以减少次数。

---

## 5. 主函数：参数与调用

```
int main(int argc, char *argv[])
{
    if (argc < 3) return -1;
    // argv[1] = hostname，argv[2] = resource（例如 "/index.html"）
    char *response = http_send_request(argv[1], argv[2]);
    if (response) {
        printf("response:\n%s\n", response);
        free(response);
    }
    return 0;
}
```

- **参数检查**：确保用户提供主机名和资源路径。
- **打印响应**：把完整 HTTP 响应（含状态行、头部、空行、body）打印到 stdout。

---

## 6. 整体调用流程

1. **DNS 解析**：`hostname` → IPv4 字符串。
2. **Socket 建立**：创建 TCP 连接并切换非阻塞。
3. **HTTP 请求**：构造并 `send`。
4. **响应接收**：
   - 用 `select` 等待可读（最多 5 秒）。
   - `recv` 循环读取，动态 `realloc` 保存。
   - 直至对端关闭或超时。
5. **结果返回**：返回分配好的完整响应字符串，由调用者释放。

---

通过上述分层解析，你可以清晰了解这段代码如何在 **非阻塞** 模式下，利用 **select** 与 **动态缓冲** 简单地完成一次 HTTP GET 请求并获取整个响应：从 DNS、连接，到请求、超时控制，再到增量读取与拼接，覆盖了客户端网络通信的大部分要点。

---

# 1. 主要数据结构

## 1.1 DNS 报文头 `dns_header_t`

```
c复制编辑typedef struct dns_header {
    unsigned short id;          // 事务 ID（随机）
    unsigned short flags;       // 标志字：查询/响应、递归支持等
    unsigned short questions;   // 问题条目数
    unsigned short answers;     // 回答条目数
    unsigned short authority;   // 授权记录数
    unsigned short additional; // 附加记录数
} dns_header_t;
```

- 按 RFC1035 定义：前 12 字节是固定头，后面依次是 Question、Answer、Authority、Additional 四个部分。

## 1.2 DNS 查询条目 `dns_question_t`

```
c复制编辑typedef struct dns_question {
    int length;            // 编码后的“QNAME”字段长度
    unsigned short qtype; // 问题类型（如 A 记录=1、CNAME=5 等）
    unsigned short qclass;// 问题类（IN=1）
    unsigned char *name;  // QNAME 字段：标签长度+标签内容...+0 结束
} dns_question_t;
```

- `name` 用 DNS 特有的“标签”格式（每段域名前加长度字节，最后以 `0x00` 结束）。

## 1.3 解析结果项 `dns_item_t`

```
c复制编辑typedef struct dns_item {
    char *domain; // 回答里解析出的域名（解码后的点分十进制或完整主机名）
    char *ip;     // 对应 IP 地址（字符串形式）
} dns_item_t;
```

- 用于存储在 Answer 段中以 A 记录形式返回的解析结果。

---

# 2. 构造请求报文

## 2.1 初始化报文头

```
c复制编辑int dns_create_header(dns_header_t *header) {
    memset(header, 0, sizeof(*header));
    header->id        = random() % 65536;   // 随机事务 ID
    header->flags     = htons(0x0100);      // 标准递归查询（RD=1）
    header->questions = htons(1);           // 仅一个 Question
    // answers/authority/additional 均为 0
}
```

- **ID**：客户端随机，一致性校验响应。

- **flags = 0x0100**：第 8—15 位中只设置 RD（Recursion Desired）位，告诉 DNS 服务器可以递归查询。
- **questions=1**：报文中只包含一个域名查询。

## 2.2 构造 Question 段

```
c复制编辑int dns_create_question(dns_question_t *q, const char *hostname) {
    q->length = strlen(hostname) + 2; // 最多每段标签长度+1，末尾再 +1
    q->name   = malloc(q->length);
    q->qtype  = htons(DNS_HOST);   // A 记录 = 1
    q->qclass = htons(1);          // IN 类

    // 将 "www.example.com" 拆成 [3]www[7]example[3]com[0]
    char *dup = strdup(hostname), *tok = strtok(dup, ".");
    unsigned char *p = q->name;
    while (tok) {
        size_t len = strlen(tok);
        *p++ = len;
        memcpy(p, tok, len);
        p += len;
        tok = strtok(NULL, ".");
    }
    free(dup);
    *p = 0; // 结尾的零字节
}
```

- **标签格式**：每段前一个字节存长度，最后以 `0x00` 表示结束。
- `qtype`/`qclass`：网络字节序（`htons`）存储。

## 2.3 合并为完整请求

```
c复制编辑int dns_build_requestion(dns_header_t *h, dns_question_t *q, char
*req) {
    memcpy(req, h, sizeof(*h));
    int off = sizeof(*h);
    memcpy(req + off, q->name, q->length); off += q->length;
    memcpy(req + off, &q->qtype, sizeof(q->qtype)); off += 2;
    memcpy(req + off, &q->qclass, sizeof(q->qclass)); off += 2;
    return off; // 请求总长度
}
```

- 顺序：Header（12B）→ QNAME → QTYPE(2B) → QCLASS(2B)。

---

# 3. 网络交互：UDP 发送与接收

```
c复制编辑int dns_client_commit(const char *domain) {
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    connect(fd, DNS_SERVER_IP:53);    // 方便后续 sendto/recvfrom 不用指定地址
    // 构建 header/question/request
    int len = sendto(fd, request, req_len, 0, &server_addr, sizeof);
    int n   = recvfrom(fd, response, sizeof(response), 0, NULL, NULL);
    // 调用 dns_parse_response 解析 response[0..n-1]
}
```

- **UDP 套接字**：DNS 使用 UDP（除非报文过大会升级到 TCP）。
- `sendto` / `recvfrom`：分别向 DNS 服务器发送请求、接收响应。

---

# 4. 解析响应报文

## 4.1 报文头与 Question 部分跳过

```
c复制编辑unsigned char *ptr = buffer;
ptr += 4;                                  // 跳过 ID + flags
int qdcount = ntohs(*(unsigned short*)ptr);
ptr += 2;
int ancount = ntohs(*(unsigned short*)ptr);
ptr += 6;                                  // 跳过 questions, answers, authority, additional counts

// 跳过每个 Question: QNAME + QTYPE(2B) + QCLASS(2B)
for (i = 0; i < qdcount; i++) {
    while (*ptr) ptr += (*ptr + 1);
    ptr += 5;
}
```

- 先读出 question/answer 数量，再循环跳过所有 Question 段。

## 4.2 递归解析域名（处理指针压缩）

DNS 响应中的 NAME 字段可能使用“指针”形式压缩：

```
c复制编辑static void dns_parse_name(unsigned char *msg, unsigned char *p, char *out, int *olen) {
    while (1) {
        unsigned char len = *p;
        if (len == 0) break;               // 结束
        if ((len & 0xC0) == 0xC0) {        // 高两位 11 => 指针
            unsigned short off = ((len & 0x3F) << 8) | p[1];
            dns_parse_name(msg, msg + off, out, olen);
            p += 2;
            return;
        } else {
            p++;
            memcpy(out + *olen, p, len);
            *olen += len;
```

```
            p += len;
            if (*p) { out[(*olen)++] = '.'; }
        }
    }
}
```

- **指针格式**：两个字节，高两位 `11` 表示后 14 位为偏移量，指向报文开头的某处标签。
- **递归调用**：遇到指针时跳到偏移处继续解析，直到遇到 `len==0`。

## 4.3 解析 Answer 条目

```
c复制编辑for (i = 0; i < ancount; i++) {
    char name[128] = {0}, ipstr[20] = {0};
    int namelen = 0;
    dns_parse_name(buffer, ptr, name, &namelen);
    ptr += 2;

    int type    = ntohs(*(unsigned short*)ptr); ptr += 2;
    int class   = ntohs(*(unsigned short*)ptr); ptr += 2;
    int ttl     = ntohs(*(unsigned short*)ptr); ptr += 2; // 简化示例
    int dlen    = ntohs(*(unsigned short*)ptr); ptr += 2;

    if (type == DNS_CNAME) {
        // CNAME 指向另一个名称，跳过 dlen
        ptr += dlen;
    }
    else if (type == DNS_HOST && dlen == 4) {
        // A 记录：4 字节 IPv4
        inet_ntop(AF_INET, ptr, ipstr, sizeof(ipstr));
        printf("%s → %s, TTL %d\n", name, ipstr, ttl);
        list[cnt++] = { strdup(name), strdup(ipstr) };
        ptr += dlen;
    }
}
```

- **NAME**：可以是压缩或标签序列，用上面函数解析得到点分域名。
- **TYPE/CLASS/TTL/DATALEN**：解析后判断类型：
  - **CNAME (5)**：别名记录，`dlen` 字节内又是一个名字，通常递归解析或忽略。
  - **A (1)**：直接放 4 字节 IPv4 地址，用 `inet_ntop` 转为字符串。
- **收集结果**：存到动态分配的 `dns_item_t` 数组中。

---

# 5. 执行示例

```
bash复制编辑$ ./dns_client www.example.com
sendto() 32 bytes
recvfrom() 64 bytes
www.example.com → 93.184.216.34, TTL 300
cnt = 1
```

- 报文长度、接收长度根据真实网络和响应决定。

- 最终打印每条 A 记录的域名、IP 和 TTL。

---

通过以上分层解析，你可以清楚地了解到：

1. **DNS 报文的构造**：Header + Question 部分的编码细节；
2. **标签与指针压缩**：在域名编码与解析时如何高效处理；
3. **UDP 传输**：涉及 `sendto` / `recvfrom` 和面向连接的 `connect` 用法；
4. **Answer 部分解析**：区分 CNAME 和 A 记录，提取 IPv4 字符串；
5. **整体流程**：一个简易但完整的“手工” DNS 查询客户端。

---

下面按模块化结构来做深入解析，除了代码功能和流程，还会对各条 SQL 语句的含义及用法加以说明。

---

# 1. 配置与 SQL 宏定义

```
#define ZZX_DB_SERVER_IP       "192.168.5.128"
#define ZZX_DB_SERVER_PORT     3306
#define ZZX_DB_SERVER_NAME     "admin"
#define ZZX_DB_SERVER_PASSWORD "Zzx123456@"
#define ZZX_DB_DEFAULTDB       "ZZX_DB"
```

- **数据库连接参数**：IP、端口、用户名、密码、默认库名。

```
#define SQL_INSERT_TBL_USER   "INSERT TBL_USER(U_NAME, U_GENGDER) VALUE('wxm'
, 'woman');"
#define SQL_SELECT_TBL_USER   "SELECT * FROM TBL_USER;"
#define SQL_DELETE_TBL_USER   "CALL PROC_DELETE_USER('wxm');"
```

1. **插入语句**

   ```
   INSERT TBL_USER(U_NAME, U_GENGDER) VALUE('wxm', 'woman');
   ```

   向 `TBL_USER` 表插入一条新记录，列名 `U_NAME`、`U_GENGDER`，对应的值分别是 `'wxm'`、`'woman'`。

2. **查询语句**

```sql
SELECT * FROM TBL_USER;
```

读取 `TBL_USER` 表的所有行、所有列。

3. **删除（存储过程调用）**

```sql
CALL PROC_DELETE_USER('wxm');
```

调用存储过程 `PROC_DELETE_USER`，传入用户名 `'wxm'`，在数据库端定义的过程里负责删除该用户及相关联记录。

```
#define SQL_INSERT_IMG_USER   "INSERT TBL_USER(U_NAME, U_GENGDER, U_IMG) VALUE('wxm' , 'woman', ?);"
#define SQL_SELECT_IMG_USER   "SELECT U_IMG FROM TBL_USER WHERE U_NAME = 'wxm';"
```

1. **二进制数据插入（预处理语句）**
   - 使用参数占位符 `?`，通过 `mysql_stmt_prepare` + `mysql_stmt_send_long_data` 分片上传大 BLOB。
2. **二进制数据查询（预处理语句）**
   - 查询指定用户的 `U_IMG` 列，后续通过 `mysql_stmt_fetch_column` 循环读取完整 BLOB。

---

## 2. 结果集查询：`zzx_mysql_select`

```c
int zzx_mysql_select(MYSQL *mysql)
{
    // 1. 发送查询
    mysql_real_query(mysql, SQL_SELECT_TBL_USER, strlen(SQL_SELECT_TBL_USER));
    // 2. 获取结果集
    MYSQL_RES *result = mysql_store_result(mysql);
    // 3. 行列统计
    int rows   = mysql_num_rows(result);
    int fields = mysql_num_fields(result);
    // 4. 遍历并打印
    while ((row = mysql_fetch_row(result)) != NULL) {
        for (int i = 0; i < fields; i++)
            printf("%s ", row[i] ? row[i] : "NULL");
        printf("\n");
    }
    // 5. 释放
    mysql_free_result(result);
}
```

- **mysql_real_query**：发送 SQL，返回非零则失败。
- **mysql_store_result**：将所有结果行全部读入客户端内存。
- **mysql_num_rows / mysql_num_fields**：获取行数与列数，用于打印格式。
- **mysql_fetch_row**：按行遍历，每个 `MYSQL_ROW` 实质上是 `char **`，按列索引访问。
- **场景**：适合中小规模查询；若结果集非常大，可改用 `mysql_use_result` 分块读取。

---

## 3. 本地文件 I/O：`read_image` / `write_image`

```
int read_image(char *filename, char *buffer) { … }
```

- **功能**：打开文件（`rb`），`fseek/ftell` 获得大小，`fread` 一次性读入 `buffer`，返回实际字节数。

```
int write_image(char *filename, char *buffer, int length) { … }
```

- **功能**：打开/创建文件（`wb+`），将 `buffer[0..length-1]` 写入磁盘。

---

## 4. 写入大 BLOB：`mysql_write_image`

```
int mysql_write_image(MYSQL *handle, char *buffer, int length) {
    MYSQL_STMT *stmt = mysql_stmt_init(handle);
    mysql_stmt_prepare(stmt, SQL_INSERT_IMG_USER, …);

    MYSQL_BIND param = {0};
    param.buffer_type = MYSQL_TYPE_LONG_BLOB;
    // 先不绑定具体缓冲区大小，使用 send_long_data 分片
    mysql_stmt_bind_param(stmt, &param);
    // 分片上传
    mysql_stmt_send_long_data(stmt, 0, buffer, length);
    mysql_stmt_execute(stmt);
    mysql_stmt_close(stmt);
}
```

1. **预处理**：`mysql_stmt_prepare` 将含 `?` 的 SQL 编译成二进制执行计划。
2. **参数绑定**：`MYSQL_BIND` 结构仅指定类型 `LONG_BLOB`。
3. **分片上传**：`mysql_stmt_send_long_data` 可多次调用，适合超大文件。
4. **执行与清理**：`mysql_stmt_execute` 将所有片段拼接入表，关闭 `stmt`。

## 5. 读取大 BLOB：`mysql_read_image`

```c
int mysql_read_image(MYSQL *handle, char *buffer, int length) {
    MYSQL_STMT *stmt = mysql_stmt_init(handle);
    mysql_stmt_prepare(stmt, SQL_SELECT_IMG_USER, …);

    MYSQL_BIND result = {0};
    result.buffer_type = MYSQL_TYPE_LONG_BLOB;
    unsigned long total_length = 0;
    result.length = &total_length;
    mysql_stmt_bind_result(stmt, &result);

    mysql_stmt_execute(stmt);
    mysql_stmt_store_result(stmt);

    // 边缘触发式分片取回
    while (mysql_stmt_fetch(stmt) == MYSQL_DATA_TRUNCATED) {
        // 每次只读1字节，然后循环移动 buffer
        mysql_stmt_fetch_column(stmt, &result, 0, start);
        start += result.buffer_length;
    }

    mysql_stmt_free_result(stmt);
    mysql_stmt_close(stmt);
    return total_length;
}
```

- **mysql_stmt_store_result**：将整个结果集缓冲到客户端，便于随机访问 BLOB。
- **mysql_stmt_fetch**：首次调用返回 `MYSQL_DATA_TRUNCATED`（BLOB 被截断），表示需要逐列读取。
- **mysql_stmt_fetch_column**：按偏移 `start` 分段把真正数据复制到 `buffer`。

## 6. 主流程：`main`

1. **初始化与连接**

```c
mysql_init(&mysql);
mysql_real_connect(&mysql, ZZX_DB_SERVER_IP, …);
```

2. **查询演示**

```c
zzx_mysql_select(&mysql);   // 打印当前所有用户
```

3. **存储过程删除**

```c
mysql_real_query(&mysql, SQL_DELETE_TBL_USER, …);
// 再次 zzx_mysql_select，观察“wxm”是否已被删除
```

4. **二进制图片写入/读取**

```
read_image("test.jpg", buffer);
mysql_write_image(&mysql, buffer, length);
memset(buffer,0,…);
length = mysql_read_image(&mysql, buffer,…);
write_image("test1.jpg", buffer, length);
```

5. **清理断开**

```
mysql_close(&mysql);
```

---

## 小结

- **通用 CRUD**：`mysql_real_query` + `mysql_store_result` 适合普通文本数据。
- **存储过程**：`CALL PROC_DELETE_USER(arg)` 用于将复杂操作封装在 DB 端。
- **大对象（BLOB）**：须用预处理 + `mysql_stmt_send_long_data` / `fetch_column` 分片读写。
- **错误处理**：每步都判断返回值并打印 `mysql_error()`，方便定位问题。

通过以上说明，你应能全面理解如何在 C 程序中：

1. 建立并管理 MySQL 连接；
2. 执行普通 SQL、存储过程；
3. 读写二进制大数据（图片、文件）。

---

## 写入时的分片：`mysql_stmt_send_long_data`

```
// 1. 预处理 SQL，SQL_INSERT_IMG_USER 中只有一个占位符 “? ”
MYSQL_STMT *stmt = mysql_stmt_init(handle);
mysql_stmt_prepare(stmt, SQL_INSERT_IMG_USER, strlen(SQL_INSERT_IMG_USER));

// 2. 构造参数绑定结构体，仅指定类型为 LONG_BLOB
MYSQL_BIND param = {0};
param.buffer_type = MYSQL_TYPE_LONG_BLOB;
// 不在这里提供 buffer/length
mysql_stmt_bind_param(stmt, &param);

// 3. 分片上传数据
//    参数索引 0 对应 SQL 中第一个 “?” 占位符
//    将整个 buffer（图片数据）一次性当作一个“片”上传
mysql_stmt_send_long_data(stmt, 0, buffer, length);

// 4. 真正执行：服务器端会把所有 send_long_data 累积的数据拼接到一起
mysql_stmt_execute(stmt);
```

- **要点**

    1. **不在 `MYSQL_BIND` 里指定 `buffer` 和 `buffer_length`**，而是将这两项留空（`param.buffer = NULL; param.length = NULL;`）。

2. **调用** `mysql_stmt_send_long_data`：你可以多次调用它，为同一个参数传送多段数据。每次调用都会将那段数据附加到该参数的内部缓冲区里。
3. **最后调用** `mysql_stmt_execute`：MySQL 客户端库会将所有先前“分片”上传的内容组合成完整的 BLOB，然后一次性发送给服务器执行插入。

- **好处**
  - 如果你上传的 BLOB 很大（比如几 MB、几十 MB），可以分多次调用 `mysql_stmt_send_long_data(..., ptr + offset, chunk_size)`，避免一次性分配一个巨大的 `buffer`，也减少单次网络负载峰值。

---

# 读取时的分片：`mysql_stmt_fetch_column`

```
// 1. 预处理 SQL，绑定输出类型为 LONG_BLOB
MYSQL_STMT *stmt = mysql_stmt_init(handle);
mysql_stmt_prepare(stmt, SQL_SELECT_IMG_USER, strlen(SQL_SELECT_IMG_USER));
MYSQL_BIND result = {0};
result.buffer_type   = MYSQL_TYPE_LONG_BLOB;
unsigned long total_length = 0;
result.length        = &total_length;      // 用于接收整个 BLOB 的总长度
mysql_stmt_bind_result(stmt, &result);

// 2. 执行并缓存结果集
mysql_stmt_execute(stmt);
mysql_stmt_store_result(stmt);

// 3. 首次调用 fetch：如果 BLOB 长度超过 result.buffer_length（这里未设置，默认为 0），
//    mysql_stmt_fetch 返回 MYSQL_DATA_TRUNCATED，表示需要分片读取
int ret = mysql_stmt_fetch(stmt);
if (ret == MYSQL_DATA_TRUNCATED) {
    // total_length 里已经有了完整的 BLOB 长度
    int start = 0;
    // 4. 循环调用 fetch_column，每次一个小片（示例中 buffer_length=1），
    //    并更新 result.buffer 指向 buffer + start
    while (start < (int)total_length) {
        result.buffer        = buffer + start;
        result.buffer_length = 1;  // 每次只取 1 字节
        mysql_stmt_fetch_column(stmt, &result, 0, start);
        start += result.buffer_length;
    }
}
```

- **要点**
  1. `mysql_stmt_store_result`：把整个行集缓冲到客户端内存，准备随机访问 BLOB。
  2. `mysql_stmt_fetch`：首次调用时若 `result.buffer_length` 不足以一次性容纳整个 BLOB，返回 `MYSQL_DATA_TRUNCATED`，并在 `total_length` 中告知完整长度。
  3. `mysql_stmt_fetch_column`：在同一个 `stmt` 上针对列索引（这里是第 0 列），指定 **偏移量** `start`，并用 `buffer_length` 指定每次想要读取的字节数，迭代直到读完。
- **灵活性**

- 你可以根据内存或网络情况，选择一次读 1KB、4KB、64KB……每次调整 `result.buffer_length`，以获得最佳性能。
- 这样就能避免一次性为超大 BLOB 分配非常大的临时缓冲，也可在分块读取时做流式处理。

---

## 总结

- **写入**：利用 `mysql_stmt_send_long_data` 按需将大 BLOB 拆成若干片段逐次发送；在最终 `mysql_stmt_execute` 时，库会自动合并。
- **读取**：在 `mysql_stmt_fetch` 后，用 `mysql_stmt_fetch_column` 按需分片拉取 BLOB 内容，直到累计读取的字节数达到 `total_length`。

---

# 为什么要用预处理（Prepared Statement）

1. **防止 SQL 注入**
   将 SQL 语句结构（模板）与参数分离，用户输入永远不会被直接拼到 SQL 字符串里。即使参数中包含 `'; DROP TABLE …` 等，也只是当作普通数据插入，不会被解析为 SQL。
2. **提升性能**
   - **编译复用**：同一个语句模板只需在服务器端编译一次，后续只需传输不同参数即可。适合在短时间内多次执行同一结构的 SQL（如批量插入、批量查询）。
   - **网络开销**：SQL 结构只发送一次，减少了网络传输大小。
3. **处理大数据**
   对于大 BLOB、CLOB 等数据，预处理接口允许分片发送（`mysql_stmt_send_long_data`），避免一次性分配巨大的内存缓冲区。

---

# 预处理的完整生命周期

下面以“插入图片”流程为例，说明典型的五部曲：

| 步骤 | 函数 | 说明 |
| --- | --- | --- |
| **1. 初始化 Statement** | `MYSQL_STMT *stmt = mysql_stmt_init(MYSQL *mysql)` | 创建一个空的语句句柄，与 `MYSQL` 连接对象关联；若返回 NULL，说明内存或连接有问题。 |
| **2. 预编译 SQL** | `mysql_stmt_prepare(stmt, query, query_len)` | 将带 `?` 占位符的 SQL 发送到服务器端编译；失败可通过 `mysql_error(mysql)` 获取错误信息。 |

| 步骤 | 函数 | 说明 |
| --- | --- | --- |
| **3. 绑定参数（可多次）** | `mysql_stmt_bind_param(stmt, MYSQL_BIND *params)` | 用 `MYSQL_BIND` 数组填充所有 `?` 对应的类型、长度、指针等。若是大 BLOB，又或不提前知道长度，可留空，后续用 `mysql_stmt_send_long_data` 分片上传。 |
| **4. 传输长数据（可选）** | `mysql_stmt_send_long_data(stmt, param_index, buffer, len)` | 如果某个参数非常大（如图片、音频），可多次调用此函数分块上传，每次提供一段数据；在 `mysql_stmt_execute` 时服务器才真正拼装。 |
| **5. 执行 & 关闭** | `mysql_stmt_execute(stmt)` `mysql_stmt_close(stmt)` | 向服务器发出执行命令，等待结果（对于 INSERT/UPDATE/DELETE 无返回行集）。用完后释放 `stmt` 资源。 |

> **Tip**：如果你要拿回查询结果，还需在执行前后加上：
>
> - `mysql_stmt_bind_result(stmt, MYSQL_BIND *results);` 绑定输出列缓冲区
> - `mysql_stmt_execute(stmt); mysql_stmt_store_result(stmt);` 获得行集
> - `mysql_stmt_fetch(stmt);` 或 `mysql_stmt_fetch_column()` 逐行/逐列读取