第5章 Socket编程

在现代软件开发中，网络通信是构建分布式应用、实现跨设备交互的核心技术。Socket编程作为网络通信的基础手段，为不同主机上的进程搭建了数据交互的桥梁，无论是即时通信工具、文件传输服务，还是分布式计算框架，都离不开它的支撑。本章将聚焦于Socket编程的核心知识，从最基础的概念讲起，帮助理解Socket作为网络通信端点的本质，弄清楚流式套接字与数据报套接字的区别，以及TCP和UDP在可靠性、传输效率上的差异。学习网络通信的基本流程，从套接字的创建、地址绑定，到连接建立、数据收发，每一步的实现细节都将被细致拆解。深入探讨不同的网络模型，看看循环服务器如何处理简单请求，并发服务器又如何通过多进程、多线程应对高并发场景，理解它们各自的优势与局限。阻塞I/O与非阻塞I/O的工作机制也会被详细解析，让人明白在不同场景下该如何选择合适的I/O模式。通过一系列编程实践，把理论知识转换为实际能力，掌握在网络环境下处理数据交互、解决并发冲突、保证通信效率的技巧。无论想开发网络应用，还是想理解分布式系统的底层通信机制，本章内容都将为之打下坚实的基础。

5.1 Socket编程基础

5.1.1 什么是Socket

Socket(套接字)是网络通信的基础组件，它本质上是一个“通信端点”，用于标识网络中不同主机上的进程。通过Socket，应用程序可以在网络中发送和接收数据，实现跨主机或跨进程的信息交互。形象地说，Socket就像“网络版的文件句柄”：如果把网络通信比作打电话，那么Socket就是“电话号码”，它包含了通信所需的地址(IP地址)和端口(进程标识)，确保数据能准确送达目标进程。在实际编程中，Socket通过一系列API函数实现功能，这些函数屏蔽了底层网络协议(如TCP/IP)的细节，让开发者无须深入理解协议栈即可实现通信。例如，发送数据时，开发者只需调用发送函数，无须关心数据如何被拆分成数据包、如何通过路由传输等底层操作。

5.1.2 Socket的类型

根据通信方式和协议的不同，Socket主要分为以下3类。

1. 流式套接字

流式套接字(SOCK_STREAM)基于TCP(Transmission Control Protocol，传输控制协

议),提供面向连接的可靠通信。它具有以下特点。

(1) 数据传输前需建立连接(类似打电话前拨号);

(2) 数据按顺序传输,无丢失、无重复(通过确认机制和重传机制保证);

(3) 适用于对可靠性要求高的场景,如文件传输、网页浏览、邮件发送等。

2. 数据报套接字

数据报套接字(SOCK_DGRAM)基于 UDP(User Datagram Protocol,用户数据报协议),提供无连接的不可靠通信。它具有以下特点。

(1) 无须建立连接,直接发送数据(类似发短信);

(2) 数据可能丢失、乱序,且大小有限制(通常不超过 64KB);

(3) 适用于对实时性要求高的场景,如视频通话、语音聊天、实时游戏等。

5.1.3 TCP 与 UDP 的对比

Socket 编程分为 TCP 和 UDP 两个模块,其中,TCP 是可靠的、安全的,常用于发送文件等,而 UDP 是不可靠的、不安全的,常用作视频通话等。TCP 和 UDP 是网络通信中最常用的两种协议,它们的核心区别如表 5-1 所示。

表 5-1 TCP 与 UDP 的比较

特　　性	TCP	UDP
是否连接	面向连接	无连接
是否可靠	可靠传输,使用流量控制和拥塞控制	不可靠传输,不使用流量控制和拥塞控制
连接对象个数	只能一对一通信	支持一对一、一对多、多对一和多对多交互通信
传输方式	面向字节流	面向报文
首部开销	首部最小 20B,最大 60B	首部开销小,仅 8B
使用场景	适用于要求可靠传输的应用,如文件传输	适用于实时应用(如 IP 电话、视频会议、直播等)

5.1.4 Socket 通信的基本流程

无论是 TCP,还是 UDP,Socket 通信都遵循一定的流程。下面分别介绍两种协议的典型通信步骤。

1. TCP 通信流程

TCP 是面向连接的协议,通信前必须建立连接。TCP 通信流程如图 5-1 所示,具体流程如下。

1) 服务器端

(1) 创建 Socket(socket()):生成一个套接字对象,指定协议类型(TCP)。

(2) 绑定地址和端口(bind()):将 Socket 与本机的 IP 地址和端口号关联,确保其他主机能找到该进程。

(3) 监听连接(listen()):进入等待状态,允许客户端发起连接请求(需指定最大等待队列长度)。

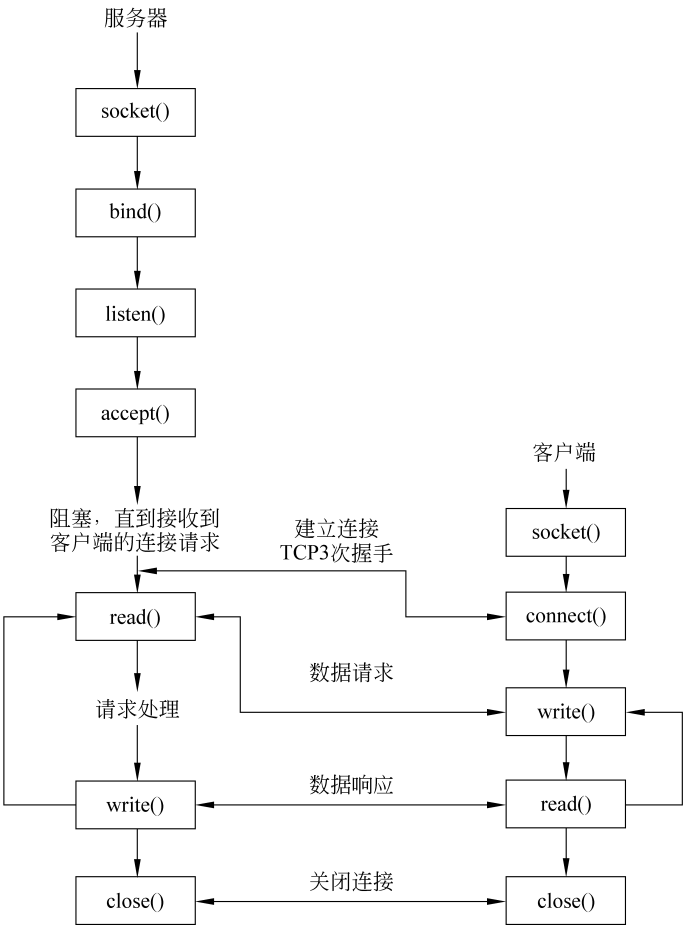

图 5-1　TCP 通信流程

(4) 接受连接(accept())：从等待队列中取出一个连接请求，创建新的 Socket 用于与该客户端通信(原 Socket 继续监听新请求)。

(5) 收发数据(recv()/send())：通过新创建的 Socket 与客户端交换数据。

(6) 关闭连接(closesocket())：通信结束后，释放资源。

2) 客户端

(1) 创建 Socket(socket())：与服务器端类似，生成套接字对象。

(2) 连接服务器(connect())：向服务器的 IP 地址和端口发起连接请求，完成 3 次握手。

(3) 收发数据(recv()/send())：与服务器端交换数据。

(4) 关闭连接(closesocket())：通信结束后，释放资源。

2. UDP 通信流程

UDP 是无连接的协议，无须建立连接。UDP 通信流程如图 5-2 所示，具体流程如下。

1) 服务器端

(1) 创建 Socket(socket())：指定协议类型(UDP)。

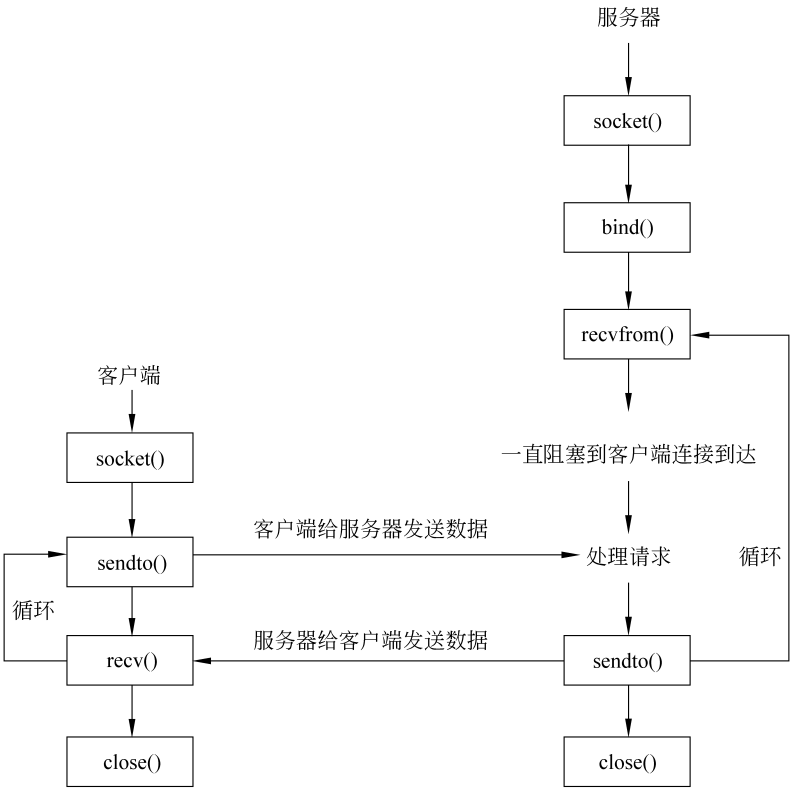

图 5-2　UDP 通信流程

(2) 绑定地址和端口(bind()):与 TCP 服务器相同,关联本机地址和端口。

(3) 收发数据(recvfrom()/sendto()):直接接收客户端发送的数据,或向指定客户端发送数据(需指定目标 IP 和端口)。

(4) 关闭连接(closesocket()):通信结束后释放资源。

2) 客户端

(1) 创建 Socket(socket()):生成 UDP 套接字。

(2) 收发数据(recvfrom()/sendto()):无须连接,直接向服务器的 IP 和端口发送数据,或接收服务器的回复。

(3) 关闭连接(closesocket()):通信结束后释放资源。

5.2 网络模型

网络模型是 Socket 编程的核心框架,它定义了服务器与客户端之间的通信规则和协作方式。在实际应用中,网络模型的选择直接影响程序的性能、稳定性和可扩展性。根据处理客户端请求方式的不同,常用的网络模型可分为循环服务器模型和并发服务器模型,其中并发服务器模型又可细分为多进程并发模型和多线程并发模型。本节将详细介绍这些模型的工作原理、适用场景及实现案例,帮助读者理解不同模型的特点和使用方法。

5.2.1　循环服务器模型

循环服务器模型是最基础的网络通信模型，其核心思想是服务器一次只能处理一个客户端请求，处理完成后再依次处理下一个请求。这种模型的设计简单、直观，适合客户端请求频率低、处理时间短的场景（如简单的查询服务），但无法应对多客户端同时请求的高并发场景。循环服务器模型的工作流程如图 5-3 所示。

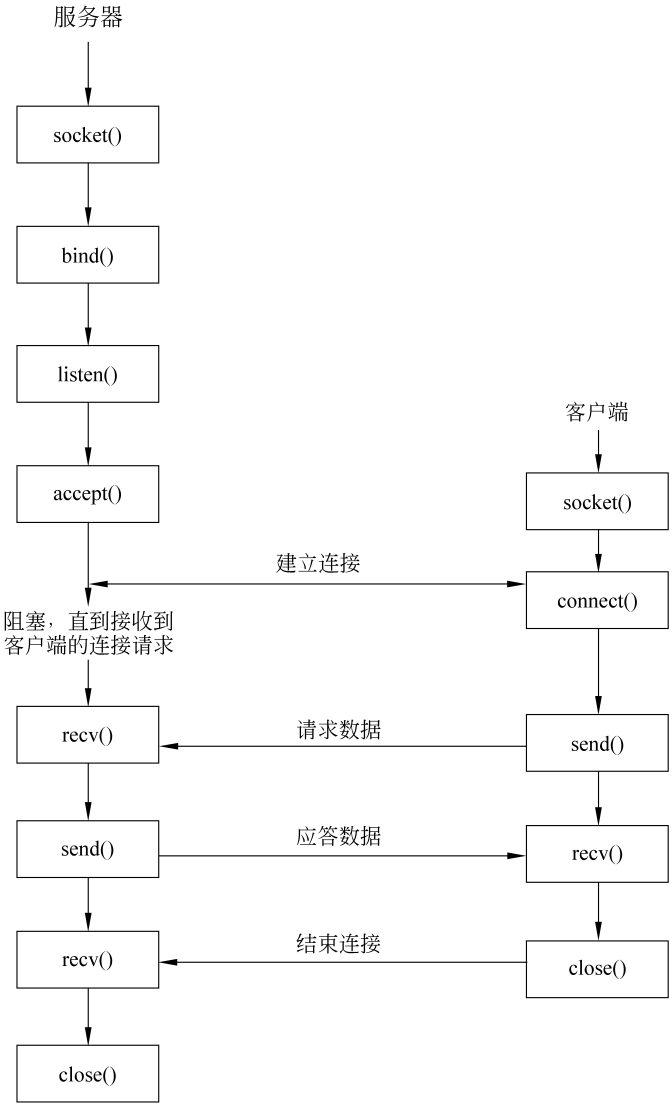

图 5-3　循环服务器模型的工作流程

1. 工作原理

循环服务器的工作流程可分为以下几个步骤。

（1）初始化阶段：服务器创建 Socket 对象，绑定到指定端口，并开始监听客户端的连接请求。

(2) 等待连接阶段：服务器通过 accept()函数阻塞等待客户端连接，此时服务器处于“停滞”状态，直到有客户端发起连接。

(3) 处理请求阶段：当客户端连接成功后，服务器与客户端进行数据交互(接收请求、处理数据、返回结果)。

(4) 循环等待阶段：处理完当前客户端的请求后，服务器关闭与该客户端的连接，再次回到 accept()函数处，等待下一个客户端连接。

2. 优缺点

(1) 优点：实现简单，无须考虑并发控制和资源共享问题，适合初学者理解网络通信的基本流程。

(2) 缺点：同一时间只能处理一个客户端，响应效率低，若前一个客户端的处理时间过长，则后续客户端会被长时间阻塞，不适合多客户端同时请求的场景。

3. 循环服务器与客户端的实现

以下是基于 TCP 的循环服务器和客户端简单实现，服务器接收客户端发送的字符串并返回其长度。

(1) 下面是服务器代码，使用 C 语言编写。

```
int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE] = {0};
    //创建 TCP Socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket 创建失败");
        exit(EXIT_FAILURE);
    }
    //配置服务器地址
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY; //监听所有网络接口
    address.sin_port = htons(PORT);
    //绑定 Socket 到端口
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("端口绑定失败");
        exit(EXIT_FAILURE);
    }
    //监听连接(最大等待队列长度为 5)
    if (listen(server_fd, 5) < 0) {
        perror("监听失败");
        exit(EXIT_FAILURE);
    }
    printf("循环服务器启动,监听端口 %d...\n", PORT);
    while (1) { //循环处理客户端
        //阻塞等待客户端连接
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
        (socklen_t*)&addrlen)) < 0) {
            perror("接受连接失败");
```

```
            continue;
        }
        printf("客户端连接成功\n");
        //读取客户端数据
        read(new_socket, buffer, BUFFER_SIZE);
        printf("收到客户端数据: %s\n", buffer);
        //处理数据: 计算字符串长度
        int len = strlen(buffer);
        char response[BUFFER_SIZE];
        sprintf(response, "字符串长度为: %d", len);
        //发送结果给客户端
        send(new_socket, response, strlen(response), 0);
        printf("发送响应: %s\n", response);
        //关闭连接
        close(new_socket);
        printf("客户端连接关闭\n");
    }
    return 0;
}
```

(2) 下面是客户端代码,使用 C 语言编写。

```
int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
    //创建 TCP Socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket 创建失败");
        exit(EXIT_FAILURE);
    }
    //配置服务器地址
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    serv_addr.sin_addr.s_addr = INADDR_ANY;     //连接本地服务器
    //连接服务器
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("连接服务器失败");
        exit(EXIT_FAILURE);
    }
    //发送数据到服务器
    printf("请输入字符串: ");
    fgets(buffer, BUFFER_SIZE, stdin);
    buffer[strcspn(buffer, "\n")] = '\0';        //去除换行符
    send(sock, buffer, strlen(buffer), 0);
    printf("数据已发送: %s\n", buffer);
    //接收服务器响应
    read(sock, buffer, BUFFER_SIZE);
    printf("服务器返回: %s\n", buffer);
    //关闭连接
    close(sock);
    return 0;
}
```

4. 代码说明

服务器通过 socket()创建 TCP 套接字，通过 bind()绑定到 8080 端口，通过 listen()开始监听连接。在无限循环中，accept()阻塞等待客户端连接，处理完一个客户端后关闭连接并继续等待下一个客户端。

客户端通过 socket()创建套接字，通过 connect()连接服务器，发送字符串后，接收服务器返回的长度信息。

运行时，服务器会依次处理客户端的请求，若有多个客户端同时连接，则只有前一个客户端处理完成后，后一个客户端才能被响应。

5.2.2 并发服务器模型

当需要同时处理多个客户端请求时，循环服务器模型无法满足需求，此时需使用并发服务器模型。服务器接收客户端连接后，通过多进程或多线程为每个客户端分配独立的处理单元，主服务器则继续等待新的连接请求。这种模型能显著提高服务器的并发处理能力，适合客户端数量多、请求频繁的场景。

1. 多进程并发模型

多进程并发模型通过 fork()系统调用创建子进程，由子进程专门处理客户端请求，主进程则继续监听新的连接。每个子进程拥有独立的内存空间和资源，因此无须担心数据竞争问题，但进程的创建和切换开销较大。多进程并发服务器模型如图 5-4 所示。

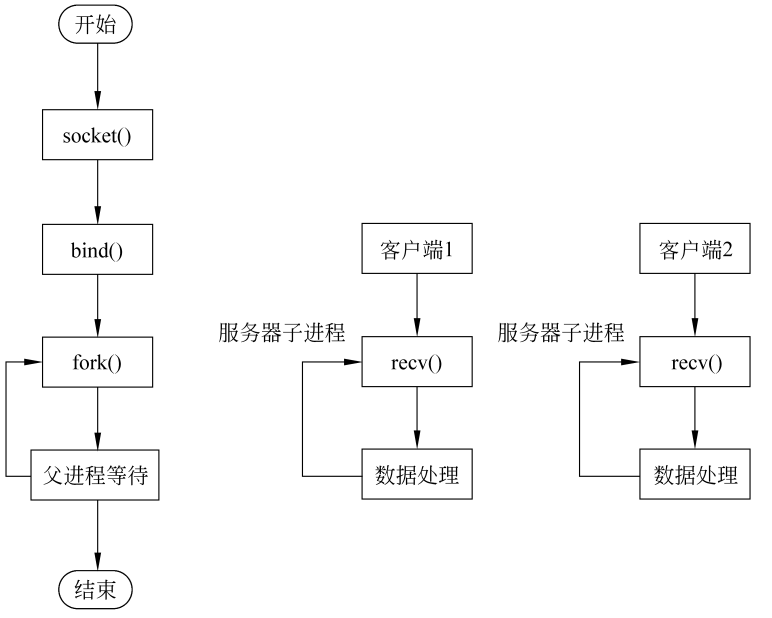

图 5-4 多进程并发服务器模型

1) 工作原理

多进程并发服务器的工作流程如下。

(1) 初始化阶段：主服务器创建 Socket、绑定端口并开始监听连接，与循环服务器

相同。

(2) 接受连接阶段：主服务器通过 accept()接受客户端连接，得到与该客户端对应的 Socket 描述符。

(3) 创建子进程阶段：主服务器调用 fork()创建子进程，子进程继承主进程的资源(包括与客户端的 Socket 连接)。

(4) 分工处理阶段。

子进程，负责与客户端进行数据交互(处理请求、返回结果)，完成后关闭连接并退出。

主进程，关闭与当前客户端的 Socket 描述符(避免资源泄露)，继续通过 accept()等待新的客户端连接。

2) 优缺点

(1) 优点：进程间相互独立，一个子进程崩溃不会影响其他进程或主进程，安全性高。

(2) 缺点：进程创建和销毁的开销大，内存占用高，不适合超高并发场景(如同时处理数千个客户端请求)。

3) 多进程并发服务器实现

以下是多进程并发服务器的简单实现，功能与循环服务器相同(计算字符串长度)，但通过子进程并发处理多个客户端请求。

下面是服务器代码，用 C 语言编写。

```
//回收僵尸进程(子进程退出后释放资源)void handle_zombie(int sig) {
    waitpid(-1, NULL, WNOHANG);}
//子进程处理客户端请求 void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE] = {0};
    read(client_socket, buffer, BUFFER_SIZE);
    printf("子进程(PID: %d)收到数据: %s\n", getpid(), buffer);
    //计算字符串长度并返回
    int len = strlen(buffer);
    char response[BUFFER_SIZE];
    sprintf(response, "字符串长度为: %d", len);
    send(client_socket, response, strlen(response), 0);
    close(client_socket);
    printf("子进程(PID: %d)处理完毕,退出\n", getpid());
    exit(0);}
int main() {
    int server_fd, client_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    pid_t pid;
    //注册信号处理函数,回收僵尸进程
    signal(SIGCHLD, handle_zombie);
    //创建、绑定、监听 Socket(与循环服务器相同)
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket 创建失败");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
```

```
    address.sin_port = htons(PORT);
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 5);
    printf("多进程服务器启动,监听端口 %d...\n", PORT);
    while (1) {
        //接受客户端连接
        client_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_
t*)&addrlen);
        if (client_socket < 0) {
            perror("接受连接失败");
            continue;
        }
        printf("收到客户端连接\n");
        //创建子进程处理客户端
        pid = fork();
        if (pid < 0) {
            perror("创建子进程失败");
            close(client_socket);
            continue;
        } else if (pid == 0) {
            //子进程: 处理客户端请求
            close(server_fd);     //子进程无须监听端口,关闭主服务器 Socket
            handle_client(client_socket);
        } else {
            //主进程: 关闭客户端 Socket,继续等待新连接
            close(client_socket);
            printf("主进程: 子进程(PID: %d)已创建,继续等待新连接\n", pid);
        }
    }
    return 0;
}
```

4) 代码说明

(1) 僵尸进程处理: 通过 signal(SIGCHLD,handle_zombie)注册信号处理函数,在子进程退出后自动回收其资源,避免内存泄漏。

(2) 进程分工: 子进程专注于处理客户端请求,主进程则继续等待新连接,实现并发处理。

2. 多线程并发模型

多线程并发模型通过创建线程处理客户端请求,线程是进程内的执行单元,共享进程的内存空间和资源,因此创建和切换的开销远小于进程。这种模型适合高并发场景,但需注意线程间的资源同步(如使用互斥锁避免数据竞争)。多线程并发服务器模型如图 5-5 所示。

1) 工作原理

多线程并发服务器的工作流程如下。

(1) 初始化阶段: 主服务器创建 Socket、绑定端口并监听连接,与前面的模型相同。

(2) 接收连接阶段: 主服务器通过 accept()接受客户端连接,得到 Socket 描述符。

(3) 创建线程阶段: 主服务器调用线程创建函数(如 pthread_create()),创建新线程处