【网络编程】网络编程知识点总结 | socket通信，多进程，多线程，IO复用 ， 线程池等 （秋招篇）

今天一定要洛必达 已于 2023-08-10 09:40:44 修改

文章目录

秋招笔记汇总篇之**Linux网络编程**
笔者是拿chatgpt写的，所以可能部分答案存在一定出路（3.5版本 GPT 有些缺陷），有出入的地方还望各位同学指出。
参考教材：TCP/IP网络编程
2023.8.9 首次更新,有些内容未完成会持续更新

UDP也需要端口号

UDP（用户数据报协议）也需要端口号来实现通信。端口号是一个16位的数字，用于标识应用程序或服务在计算机上的特定通信端口。在UDP通信中，数据通过UDP数据包发送和接收。

基于TCP的socket通信中，简易服务端的六步依次为?

1）创建套接字（Socket）：
使用 socket() 函数创建一个套接字，并指定地址族（例如，IPv4或IPv6）和套接字类型（例如，TCP套接字）。

2）绑定套接字到本地地址和端口：
使用 bind() 函数将套接字绑定到一个本地的IP地址和端口号，用于监听客户端的连接请求。

3）监听连接请求：
使用 listen() 函数将套接字设置为监听状态，指定服务器同时处理的最大连接请求排队数量。

4）接受客户端连接请求：
使用 accept() 函数接受客户端的连接请求，该函数会阻塞程序执行，直到有客户端连接进来。一旦有连接请求，accept() 函数会返回一个新的套接字，通过这个套接字可以与客户端进行通信。

5）与客户端进行通信：
通过新的套接字，服务端可以与客户端进行数据交换，使用 recv() 函数接收客户端发送的数据，使用 send() 函数向客户端发送数据。

6）关闭套接字：
在通信完成后，使用 close() 函数关闭服务端与客户端的套接字连接。如果需要继续监听其他客户端的连接，可以回到第 4 步。

基于TCP的socket通信中，简易客户端的四步依次为?

基于TCP的Socket通信中，客户端的四个步骤依次为：

1）创建Socket：客户端创建一个套接字（Socket），用于与服务端建立连接。

2）建立连接：客户端通过套接字连接到指定的服务器IP地址和端口号。

3）发送数据：连接建立后，客户端可以使用套接字发送数据给服务端。

4）接收数据：客户端可以通过套接字接收服务端发送的数据，以进行双向通信。

介绍一下在linux环境下，服务器这六步的使用到的一些函数（参数，返回值类型等）

在传统的C语言中，基于TCP的Socket通信的函数原型可以分为以下几个主要函数：

1）socket 函数：

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

这个函数用于创建一个新的套接字，并返回一个文件描述符（socket descriptor）。参数 domain 指定套接字使用的地址族，常见的有 AF_INET 表示IPv4地址族，AF_INET6 表示IPv6地址族。参数 type 指定套接字的类型，常见的有 SOCK_STREAM 表示TCP套接字，SOCK_DGRAM 表示UDP套接字。**参数 protocol 通常设置为0，表示使用默认的协议。**（当protocol参数设置为0时，socket函数会根据第二个参数（socket类型）自动选择合适的协议。对于SOCK_STREAM类型的套接字，会选择TCP协议；对于SOCK_DGRAM类型的套接字，会选择UDP协议。除此之外还有**IPPROTO_TCP，IPPROTO_UDP，IPPROTO_SCTP等参数选项，分别代表TCP协议，UDP协议，SCTP协议**）
socket 函数的返回值：
成功创建套接字时，返回一个非负整数的套接字描述符（socket descriptor）。
失败时，返回-1，并设置全局变量 errno 表示具体的错误原因。

2）bind 函数：

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

这个函数用于将套接字绑定到一个特定的IP地址和端口号。sockfd 是由 socket() 返回的套接字描述符，addr 是一个指向 struct sockaddr 结构的指针，用于指定要绑定的地址。addrlen 是 addr 结构的大小。
bind 函数的返回值：

成功绑定套接字到指定地址和端口时，返回0。
失败时，返回-1，并设置全局变量 errno 表示具体的错误原因。

3）listen 函数：

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

这个函数用于将套接字设置为监听状态，等待客户端的连接请求。sockfd 是由 socket() 返回的套接字描述符，backlog 表示服务端允许的连接请求的最大排队数量。
listen 函数的返回值：
成功设置套接字为监听状态时，返回0。
失败时，返回-1，并设置全局变量 errno 表示具体的错误原因。

4）accept 函数：

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

这个函数用于接受客户端的连接请求，并返回一个新的套接字描述符，通过这个新的套接字可以与客户端进行通信。sockfd 是监听套接字的描述符，addr 是一个指向 struct sockaddr 结构的指针，用于接受客户端的地址信息。addrlen 是 addr 结构的大小。
accept 函数返回值：
成功接受客户端连接时，返回一个新的非负整数的套接字描述符，该套接字用于与客户端进行通信。
失败时，返回-1，并设置全局变量 errno 表示具体的错误原因。

5）connect 函数：

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

这个函数用于客户端向服务端发起连接请求。sockfd 是套接字描述符，addr 是指向服务端地址的指针，addrlen 是 addr 结构的大小。
connect 函数的返回值：
成功建立与服务端的连接时，返回0。
失败时，返回-1，并设置全局变量 errno 表示具体的错误原因。

6）recv 和 send 函数：

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

这两个函数分别用于接收和发送数据。sockfd 是套接字描述符，buf 是指向缓冲区的指针，用于存储接收或发送的数据。len 表示缓冲区的大小，flags 通常设置为0。
recv 和 send 函数的返回值：

成功接收或发送数据时，返回实际接收或发送的字节数。
失败时，返回-1，并设置全局变量 errno 表示具体的错误原因。

以上是基于C语言的套接字函数原型，其中函数的返回值类型是 int，表示函数调用成功返回非负整数作为操作结果，出现错误时返回-1。同时，这些函数在错误发生时会设置全局变量 errno，用于指示具体的错误原因。在实际使用中，可以通过 perror() 函数或其他手段查看和处理错误。

介绍一下在linux环境下，简易客户端这六步的使用到的一些函数（参数，返回值类型等）

以下是基于TCP的Socket通信中，客户端可能使用的一些重要函数的格式和说明：

1）socket 函数：

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

用于创建一个新的套接字。参数说明：

domain：地址族，例如 AF_INET 表示IPv4。

type：套接字类型，例如 SOCK_STREAM 表示TCP套接字。

protocol：通常为0，表示使用默认协议（自动选择适合的协议）。

返回值：成功创建套接字时，返回套接字描述符；失败时返回-1。

2）connect 函数：

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

用于连接到服务器的指定IP地址和端口号。参数说明：

sockfd：套接字描述符，由 socket 函数返回。

addr：指向目标服务器的地址结构。

addrlen：地址结构的大小。

返回值：成功连接时，返回0；失败时返回-1。

3）send 函数：

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

用于向服务端发送数据。参数说明：

sockfd：套接字描述符，连接到服务器。

buf：要发送的数据缓冲区。

len：要发送的数据长度。

flags：标志位，通常设置为0。

返回值：返回实际发送的字节数，出现错误时返回-1。

4）recv 函数：

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

用于从服务端接收数据。参数说明：

sockfd：套接字描述符，连接到服务器。

buf：接收数据的缓冲区。

len：缓冲区长度。

flags：标志位，通常设置为0。

返回值：返回实际接收的字节数，出现错误或连接关闭时返回0。

5）close 函数：

```
int close(int sockfd);
```

用于关闭套接字连接。参数说明：

sockfd：套接字描述符。

返回值：成功关闭返回0，失败返回-1。

一对简易的TCP服务器和客户端程序

pass

TCP数据传输不存在边界是什么意思，UDP为何存在数据边界

"TCP数据传输不存在边界"是指在TCP协议下，**应用程序发送的数据在传输过程中并没有固定的分割或边界，数据被视为一个连续的字节流。**这意味着，当应用程序发送数据时，TCP协议会将数据分割成一系列的数据段（segments）进行传输，而接收方则根据接收到的数据段重新组装成完整的数据。
与UDP协议不同，UDP是无连接的、面向数据报的协议，**每个UDP数据包都有固定的大小，数据包之间是独立的，不会被合并或分割。**
在TCP中，发送方将应用程序的数据分成适当大小的数据段，然后每个数据段都添加了TCP头部信息，包括序列号等。接收方根据序列号来确定数据段的顺序，并将它们组装成原始的数据流供应用程序使用。这种方式使得应用程序可以像读取连续的字节流一样读取数据，而不需要考虑数据段的具体边界。
因此，"TCP数据传输不存在边界"意味着TCP协议对数据的切分和组装是透明的，应用程序不需要关心数据段的大小或如何组装数据，它只需要关注发送和接收整个数据流。

简易UDP服务器和客户端用到了哪些和TCP不同的函数?

1）recvfrom 函数：

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

用于从客户端接收数据。在UDP服务器中，每个数据报文都是独立的，因此需要在每次接收时指定客户端的地址结构。

sockfd：套接字描述符，由 socket 函数返回。

buf：接收数据的缓冲区。

len：缓冲区的大小。

flags：标志位，通常设置为0。

src_addr：用于存储客户端地址的结构。

addrlen：src_addr 结构的大小，在调用后会被设置为实际客户端地址的大小。

返回值：返回实际接收的字节数，出现错误时返回-1。

2）sendto 函数：

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

用于向客户端发送数据。在UDP服务器中，需要指定目标客户端的地址结构。

sockfd：套接字描述符，由 socket 函数返回。

buf：要发送的数据缓冲区。

len：要发送的数据长度。

flags：标志位，通常设置为0。

dest_addr：目标客户端的地址结构。

addrlen：地址结构的大小。

返回值：返回实际发送的字节数，出现错误时返回-1。
这些函数是在UDP服务器中实现基本通信的关键。通过使用这些函数，你可以搭建一个简单的UDP服务器，用于接收和发送UDP数据报文。

一对简易的UDP服务器和客户端程序

UDP不同于TCP，不存在请求连接和受理过程（**在TCP通信中，通信双方需要经过三次握手的过程来建立连接，然后进行数据传输，最后通过四次挥手来终止连接。**）
UDP无listen 和accept函数
在UDP通信中，客户端和服务器通常只需要一个套接字（socket）来进行数据传输 相比之下，TCP通信通常需要一个套接字来监听连接请求（服务器端）以及一个套接字来进行实际的数据传输（客户端和服务器端）。
基于UDP的简易服务器和客户端实现各自有几个主要步骤。下面我将为你分别列出这些步骤，并提供主要的实现函数示例。
UDP服务器主要步骤：

1）创建UDP套接字（socket）

2）绑定套接字到服务器地址和端口（bind）

3）循环接收数据并处理客户端请求

接收数据（recvfrom）

处理数据

发送响应（可选，使用sendto）

4）关闭套接字（close）

UDP服务器实现示例（C++）：

```
#include <iostream>
#include <cstring>
#include <unistd.h>
#include <arpa/inet.h>

```

```cpp
int main() {
    int serverSocket = socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(12345);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr));

    char buffer[1024];
    struct sockaddr_in clientAddr;
    socklen_t clientAddrLen = sizeof(clientAddr);

    while (true) {
        memset(buffer, 0, sizeof(buffer));
        ssize_t bytesReceived = recvfrom(serverSocket, buffer, sizeof(buffer), 0,
                                         (struct sockaddr*)&clientAddr, &clientAddrLen);
        std::cout << "Received from " << inet_ntoa(clientAddr.sin_addr) << ": " << buffer << std::endl;

        // Process data and send response if needed

        if (bytesReceived < 0) {
            std::cerr << "Error receiving data." << std::endl;
        }
    }

    close(serverSocket);

    return 0;
}
```

UDP客户端主要步骤：

1）创建UDP套接字（socket）

2）填充服务器地址结构

3）发送数据到服务器（sendto）

4）等待接收服务器响应（可选，使用recvfrom）

5）关闭套接字（close）

UDP客户端实现示例（C++）：

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

int main() {
    int clientSocket = socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(12345);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    const char* dataToSend = "Hello, server! This is a message from the client.";
    sendto(clientSocket, dataToSend, strlen(dataToSend), 0, (struct sockaddr*)&serverAddr, sizeof(serverAddr));

    // Receive response from server if needed

    close(clientSocket);

    return 0;
}
```

请注意，UDP是无连接的协议，因此在服务器端和客户端之间没有明确的连接状态。服务器在接收到数据后可以直接向客户端发送响应，而不需要提前建立连接。在UDP通信中，也可以通过sendto和recvfrom函数指定目标地址，避免建立连接。这使得UDP适用于一些轻量级、实时性要求较高的应用场景。

优雅地断开套接字连接 是什么意思 相比之下什么是不优雅的?

优雅地断开套接字连接意味着在网络通信中以一种谨慎、有序和可靠的方式关闭连接，以确保数据的完整性和通信的稳定性。这种方法可以最大程度地减少数据丢失和通信中断的风险，保证双方都有足够的时间来完成数据的传输和处理。

相比之下，不优雅的断开连接可能会导致数据的丢失、不完整或混乱，以及通信中的问题。一种不优雅的断开连接方式是突然关闭连接，而不考虑对方是否还有未处理的数据。这可能会导致接收方无法完整地接收到所有数据，或者发送方的数据被截断。不优雅的断开还可能引发网络中的问题，例如连接资源没有得到释放，造成资源泄漏。
不优雅地断开连接的例子包括：

1）强制关闭：某一方在没有通知对方的情况下，直接关闭了套接字连接。这可能导致未完成的数据传输丢失，另一方可能会因为无法正确处理而出现问题。

2）超时关闭：一方等待超时时间后强制关闭连接，而不管对方是否还有未处理的数据。这可能会导致数据不完整或丢失。

3）不处理错误：在发生错误时，不进行适当的错误处理，导致连接不正常关闭，而不是根据协议规范进行关闭。

优雅地断开连接是为了确保通信的可靠性和稳定性，尽量避免数据丢失和通信中断。在编写网络应用程序时，采用优雅的断开连接方式是至关重要的，以保障数据的完整性并提供良好的用户体验。
半关闭（Half-Close）是优雅地断开套接字连接的一种方式之一。

介绍一下半关闭，他和普通关闭有何区别？

半关闭（Half-Close）是优雅地断开套接字连接的一种方式之一。
在半关闭中，一个套接字连接中的一方先关闭其发送数据的能力，但仍然可以继续接收数据。这允许另一方继续向关闭连接的一方发送数据，直到完成所有数据传输。一旦数据传输完成，另一方也可以关闭连接，从而完成整个断开连接的过程。
半关闭的步骤如下：

1）发送方发出一个关闭请求，指示它不会再发送更多数据。

2）接收方继续接收来自发送方的数据，直到所有数据都接收完毕。

3）接收方发送一个确认，表示它已经接收了所有数据。

4）一旦确认收到，发送方关闭连接，完成断开连接的过程。

半关闭允许在断开连接之前完成所有数据的传输，从而减少数据丢失的风险。这种方法特别适用于需要双方进行双向通信并在完成数据传输后安全地关闭连接的情况，如文件传输、聊天应用等。

半关闭（Half-Close）与普通关闭（Close）之间的主要区别在于如何处理数据传输和连接断开。以下是使用C++的代码示例，展示了半关闭和普通关闭的不同之处：
半关闭的服务器端示例（C++）：

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

int main() {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
```

```
    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(12345);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
    listen(serverSocket, 1);

    std::cout << "Waiting for a connection..." << std::endl;
    int clientSocket = accept(serverSocket, nullptr, nullptr);
    std::cout << "Connected to client." << std::endl;

    const char* dataToSend = "Hello, client! This is a message from the server.";
    send(clientSocket, dataToSend, strlen(dataToSend), 0);

    // Perform half-close on the server side
    shutdown(clientSocket, SHUT_WR);

    // Continue reading data from client if needed

    close(clientSocket);
    close(serverSocket);

    return 0;
}
```

半关闭的客户端示例（C++）：

```
#include <iostream>
#include <cstring>
#include <unistd.h>
#include <arpa/inet.h>

int main() {
    int clientSocket = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddr;
```

```
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(12345);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr));

    char buffer[1024];
    memset(buffer, 0, sizeof(buffer));

    recv(clientSocket, buffer, sizeof(buffer), 0);
    std::cout << "Received: " << buffer << std::endl;

    // Perform half-close on the client side
    shutdown(clientSocket, SHUT_RD);

    close(clientSocket);

    return 0;
}
```

在半关闭的示例中，服务器端和客户端都使用shutdown函数进行半关闭操作。服务器在发送完数据后，通过shutdown(clientSocket, SHUT_WR)来关闭发送通道；客户端在接收完数据后，通过shutdown(clientSocket, SHUT_RD)来关闭接收通道。
相比之下，普通关闭只需要在需要关闭连接时调用close函数即可，不区分发送通道和接收通道。通常，普通关闭是在双方数据交换完成后断开连接，而半关闭允许在断开连接前继续进行单向数据传输。

window和linux上网络通信主要函数的一些区别（了解即可）

在大多数情况下，套接字函数的返回值在Linux系统和Windows系统上是类似的，遵循了一定的标准。但是，由于两个操作系统的实现和网络协议栈有所不同，存在一些细微的差异。
下面是一些可能存在差异的情况：

socket 函数：创建套接字时，返回的套接字描述符在Windows系统上是一个类型为 SOCKET 的数据类型，而在Linux系统上是一个整数类型（int）。

bind 函数：在绑定套接字到本地地址和端口时，Linux和Windows系统都可能返回-1，并设置相应的错误码。但是，具体的错误码可能不同，因为不同系统可能使用不同的错误代码来表示相同的错误情况。

listen 函数：将套接字设置为监听状态时，Linux和Windows系统的返回值都是0，表示成功。失败时，也会返回-1，并设置相应的错误码。

accept 函数：接受客户端连接时，Linux和Windows系统的返回值都是一个新的套接字描述符，用于与客户端进行通信。

connect 函数：在连接到服务端时，Linux和Windows系统上的返回值都是0表示成功，-1表示失败，并设置相应的错误码。

recv 和 send 函数：这两个函数在Linux和Windows系统上的返回值都表示实际接收或发送的字节数，-1表示失败，并设置相应的错误码。

虽然返回值可能存在一些差异，但是对于跨平台开发，通常可以使用宏或条件编译来处理这些差异，以便使代码在不同操作系统上都能正常工作。在实际开发中，建议在调用套接字函数后检查返回值并进行适当的错误处理，以保证程序的可靠性。

TCP不支持多播和广播

TCP协议确实不支持多播（Multicast）和广播（Broadcast）。
TCP是一种面向连接的可靠传输协议，它是点对点通信的，每个TCP连接都是一对一的。TCP连接的建立和维护需要双方互相交换数据包，而多播和广播都是一对多的通信方式，不符合TCP的连接模型。
多播和广播通常使用UDP协议来实现，因为UDP是无连接的，可以实现一对多的数据传输。在UDP中，可以使用组播（Multicast）地址实现多播通信，也可以使用广播地址实现广播通信。

然而，尽管TCP本身不支持多播，但可以通过一些技术手段实现TCP多播的功能。例如，可以使用IP多播（IPv4或IPv6）和组播（Multicast）地址来实现TCP多播。这种方法需要在应用层上进行额外的处理和协议设计，以实现TCP数据的多播传输。
另外，还有一些其他的协议和技术可以实现类似TCP多播的功能，例如使用UDP进行数据传输，并在应用层上设计自己的可靠性机制。这些方法可能会牺牲一些可靠性和顺序性，但可以实现一对多的数据传输。
总之，TCP协议本身不直接支持多播，但可以通过一些额外的技术手段实现类似TCP多播的功能。具体的实现方式取决于应用需求和网络环境。

讲一下C/S架构 和B/S架构

C/S架构（Client/Server Architecture）和B/S架构（Browser/Server Architecture）是两种常见的计算机系统架构，用于描述客户端和服务器之间的关系以及数据传输方式。它们在不同的应用场景和需求下有不同的特点和优势。

1）C/S架构（Client/Server Architecture）：
C/S架构是一种传统的架构模式，它将应用程序分为两个主要组件：客户端（Client）和服务器（Server）。客户端通常是一个独立的软件应用程序，负责用户界面和用户交互，同时也处理一些本地的计算任务。服务器端负责处理数据存储、业务逻辑、计算和协调各个客户端之间的通信。

优势：
性能：C/S架构可以在客户端和服务器之间分担计算负载，有利于提高系统性能。
离线操作：客户端可以继续工作，即使与服务器失去连接，可以在连接恢复后将数据同步到服务器。
功能强大：客户端可以通过本地计算和存储提供更丰富的功能和用户体验。

缺点：
客户端安装：每个客户端都需要安装相应的软件，维护和更新相对繁琐。
平台依赖：不同平台（操作系统）可能需要不同的客户端版本。

2）B/S架构（Browser/Server Architecture）：
B/S架构是一种基于Web浏览器的架构模式，客户端通过浏览器访问远程的Web应用程序，而应用程序的逻辑和数据处理则完全在服务器端进行。

优势：
跨平台：由于应用程序逻辑在服务器端，用户只需要一个现代的Web浏览器就可以访问应用，无需安装额外的客户端软件。
简化维护：由于应用逻辑在服务器端集中，维护和更新变得更加简单，只需要在服务器上进行更新即可。
高度可扩展：可以通过增加服务器的性能和资源来扩展应用的容量和性能。

缺点：
依赖网络：B/S架构需要网络连接才能访问应用程序，无法在离线状态下使用。
功能限制：受限于浏览器的能力，一些高级的本地功能可能无法实现。

总之，C/S架构和B/S架构各有其优势和适用场景。选择哪种架构取决于具体的应用需求、用户体验以及系统维护等因素。

CS架构中 通信时必须由客户端去连接服务端才能通信 并且服务器端需要提前启动 等待客户端并，且服务器不能主动连接客户端

介绍一下进程的创建，用到哪个函数?

在UNIX和类UNIX系统中，进程的创建通常通过fork()函数实现。该函数会创建一个当前进程的副本，称为子进程，子进程将继承父进程的代码、数据、文件描述符等属性。fork()函数会返回两次，一次在父进程中返回子进程的ID，另一次在子进程中返回0。子进程可以使用这个ID来区分自己是父进程的哪个副本。
以下是 fork() 函数的原型、返回值以及用法：

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

原型： pid_t fork(void);

返回值：

在父进程中，fork() 返回子进程的进程ID（PID）。这个PID是一个正整数值。

在子进程中，fork() 返回0。

如果创建子进程失败，fork() 返回-1，并设置 errno 错误码来指示失败的原因。

多进程实现并发服务器

pass

讲一下进程间通信方式有哪些?

进程间通信（Inter-Process Communication，IPC）是操作系统中不同进程之间进行数据交换和通信的一种方式。以下是一些常见的进程间通信方式：

1）**管道（Pipe）**：

单向通信通道，分为命名管道和匿名管道。

命名管道可以用于无亲缘关系进程之间的通信，通过文件系统命名。

匿名管道通常用于有亲缘关系进程之间的通信，创建进程时自动创建，只能在父子进程之间使用。

2）消息队列（Message Queue）：

用于在不同进程之间传递消息的通信机制，消息有特定的类型。

具备灵活性，支持点对点通信和发布-订阅模式。

3）信号量（Semaphore）：

用于控制多个进程对共享资源的访问，可以用于同步和互斥。

信号量可以阻塞进程，直到资源可用。

4）共享内存（Shared Memory）：

允许多个进程访问同一块物理内存，可以高效地进行数据交换。

需要进行同步和互斥操作，以防止数据一致性问题。

5）套接字（Socket）：

在网络编程中使用的通信方式，也可以用于进程间通信。

支持在不同主机上的进程通信。

6）文件锁（File Locking）：

使用文件系统的文件锁机制，可以在进程间进行同步和互斥操作，以确保共享文件的一致性。

7）RPC（Remote Procedure Call）：

允许远程计算机上的进程调用本地进程的函数，使远程调用看起来像是本地函数调用。

通常用于分布式系统中的进程间通信。

8）SocketPair：

用于创建一对相互连接的套接字，通常用于父子进程之间通信。

9）信号（Signal）：

进程可以向另一个进程发送信号，用于通知某些事件的发生。

通常用于处理异步事件，如进程终止、中断等。

这些进程间通信方式在不同情况下有不同的适用性和性能特点。选择适当的通信方式取决于进程之间的关系、通信需求以及应用程序的设计。

介绍一下线程相关的主要函数，及其函数原型返回值等（创建，回收，退出，分离等）

在多线程编程中，操作系统提供了一些用于线程管理的主要函数，这些函数允许你创建、回收、退出和分离线程。下面是一些常见的线程管理函数及其函数原型、返回值等信息：

1）创建线程：

函数原型：

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
```

参数：

thread：指向 pthread_t 类型的变量，用于存储新创建线程的标识符。

attr：线程属性，通常可以传入 NULL 使用默认属性。

start_routine：线程要执行的函数，通常是线程的入口点。

arg：传递给 start_routine 的参数。

返回值：成功时返回 0，失败时返回一个非零错误码。

2）回收线程：

```
int pthread_join(pthread_t thread, void **retval);
```

参数：

thread：要回收的线程的标识符。

retval：用于存储线程返回值的指针。如果不关心线程的返回值，可以传入 NULL。

返回值：成功时返回 0，失败时返回一个非零错误码。

3）退出线程：

函数原型：

```
`void pthread_exit(void *retval);`
```

参数：retval：线程的返回值。通常可以通过此值传递线程结束时的状态信息。

4）分离线程：

函数原型：int pthread_detach(pthread_t thread);

参数：thread：要分离的线程的标识符。

返回值：成功时返回 0，失败时返回一个非零错误码。

这些函数是 POSIX 线程库（pthread）中常用的线程管理函数。它们允许你创建和管理多个线程，等待线程结束并回收资源，或将线程设置为分离状态以自动释放资源。需要注意的是，线程函数返回值通常用于传递线程的结束状态或结果。在线程中调用 pthread_exit() 可以终止线程并传递返回值，而在主线程中调用 pthread_join() 可以等待其他线程结束并获取其返回值。分离线程可以使线程在结束后自动释放资源，而无需显式调用 pthread_join()。

线程同步是什么意思 为何要线程同步 竞态是啥

线程同步（Thread Synchronization）是指在多线程编程中，为了避免并发操作导致的数据不一致性、资源竞争等问题，采取的一种手段，确保多个线程按照预期的顺序协调执行，从而保证程序的正确性和稳定性。
在多线程环境中，多个线程可能同时访问共享的数据、资源或代码段，如果没有适当的同步措施，就可能出现以下问题：

1）数据不一致性： 多个线程同时对共享数据进行读写，可能导致数据不一致，即一个线程的操作影响了另一个线程期望的结果。

2）资源竞争： 多个线程同时竞争获取同一资源，如文件、内存区域等，可能导致资源被多个线程交叉使用，导致错误。
竞态（Race Condition）是指多个线程同时访问共享资源，而最终的执行结果取决于线程执行的顺序，从而导致程序的输出结果不确定。竞态可能导致数据不一致、崩溃等问题。线程同步的目标之一就是消除竞态，使得多线程程序能够在不同的执行顺序下仍能得到一致的结果。

常见的线程同步机制有哪些

互斥锁（Mutex）： 用于确保在任何时刻只有一个线程可以访问临界区（关键代码段），从而避免数据不一致和资源竞争。

条件变量（Condition Variable）： 用于线程之间的等待和通知，允许线程在某些条件满足时继续执行，以避免忙等待。

信号量（Semaphore）： 用于控制对资源的访问权限，允许一定数量的线程同时访问共享资源。

读写锁（Read-Write Lock）： 允许多个线程同时读取共享数据，但只允许一个线程写入数据。

介绍一下互斥锁实现线程同步的相关函数

在Linux中，你可以使用POSIX线程库（pthread）来操作互斥锁。
以下是一些与互斥锁相关的pthread函数：

初始化和销毁：

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)：初始化互斥锁对象。
int pthread_mutex_destroy(pthread_mutex_t *mutex)：销毁互斥锁对象。
```

加锁和解锁：

```
int pthread_mutex_lock(pthread_mutex_t *mutex)：尝试获得互斥锁，如果锁已被其他线程占用，则当前线程被阻塞，直到获得锁为止。

int pthread_mutex_trylock(pthread_mutex_t *mutex)：尝试获得互斥锁，如果锁已被占用则立即返回失败。

int pthread_mutex_unlock(pthread_mutex_t *mutex)：释放互斥锁，允许其他线程获得锁。
```

超时操作：

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *abs_timeout)：尝试在一定时间内获得互斥锁，超过指定时间后返回失败。
```

下面是一个使用pthread库的简单示例，演示了如何在两个线程中保护一个共享资源：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex;  // 创建一个互斥锁对象
int sharedValue = 0;

void* incrementSharedValue(void* arg) {
    for (int i = 0; i < 5; ++i) {
        pthread_mutex_lock(&mutex);  // 加锁，保护共享变量
        ++sharedValue;
        printf("Thread %ld: Incremented sharedValue to %d\n", (long)arg, sharedValue);
```

```
        pthread_mutex_unlock(&mutex);  // 解锁
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_mutex_init(&mutex, NULL);  // 初始化互斥锁

    pthread_create(&thread1, NULL, incrementSharedValue, (void*)1);
    pthread_create(&thread2, NULL, incrementSharedValue, (void*)2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    pthread_mutex_destroy(&mutex);  // 销毁互斥锁

    return 0;
}
```

在这个例子中，我们使用了pthread库创建了互斥锁，并在两个线程中使用pthread_mutex_lock和pthread_mutex_unlock来保护共享变量sharedValue的访问。这样可以确保在任何时刻只有一个线程可以修改sharedValue，从而避免了竞态条件。
需要注意的是，Linux下的互斥锁函数与之前提到的C++标准库中的std::mutex函数类似，但调用方式稍有不同。在使用互斥锁时，需要确保正确地加锁和解锁，以避免死锁和其他多线程问题。

介绍一下条件变量实现线程同步的相关函数

在Linux中，条件变量通常与互斥锁一起使用，用于线程间的同步和通信。
以下是一些与条件变量相关的pthread函数：

初始化和销毁：

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)：初始化条件变量对象。

int pthread_cond_destroy(pthread_cond_t *cond)：销毁条件变量对象。
```

等待和唤醒：

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)：在条件变量上等待，会阻塞当前线程并释放互斥锁，直到另一个线程调用pthread_cond_signal

int pthread_cond_signal(pthread_cond_t *cond)：唤醒一个在条件变量上等待的线程。

int pthread_cond_broadcast(pthread_cond_t *cond)：唤醒所有在条件变量上等待的线程。
```

超时操作：

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abs_timeout)：在指定时间内等待条件变量，超过指定时间后
```

下面是一个基于Linux库的简单示例，演示了如何使用条件变量实现线程间的通信，这里以生产者-消费者问题为例：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex;
pthread_cond_t cond;
int buffer = 0;
int buffer_full = 0;

void* producer(void* arg) {
    for (int i = 0; i < 5; ++i) {
        pthread_mutex_lock(&mutex);
        while (buffer_full) {
            pthread_cond_wait(&cond, &mutex);  // 等待条件变量并释放互斥锁
        }
        buffer = i;
        buffer_full = 1;
        printf("Produced: %d\n", buffer);
        pthread_cond_signal(&cond);  // 唤醒等待的消费者线程
        pthread_mutex_unlock(&mutex);
        usleep(500000);  // 模拟生产耗时
    }
    return NULL;
}

void* consumer(void* arg) {
```

```
    for (int i = 0; i < 5; ++i) {
        pthread_mutex_lock(&mutex);
        while (!buffer_full) {
            pthread_cond_wait(&cond, &mutex);  // 等待条件变量并释放互斥锁
        }
        printf("Consumed: %d\n", buffer);
        buffer_full = 0;
        pthread_cond_signal(&cond);  // 唤醒等待的生产者线程
        pthread_mutex_unlock(&mutex);
        usleep(300000);  // 模拟消费耗时
    }
    return NULL;
}

int main() {
    pthread_t producer_thread, consumer_thread;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);

    return 0;
}
```

在这个示例中，我们使用了条件变量和互斥锁来解决生产者-消费者问题。生产者线程负责生产物品并通知消费者，而消费者线程负责消费物品并通知生产者。通过使用条件变量，我们确保了生产者和消费者之间的同步和通信，避免了竞态条件。

介绍一下信号量实现线程同步的相关函数

当涉及使用信号量来实现线程同步时，Linux提供了一组函数用于创建、操作和销毁信号量。信号量是在多线程编程中用于控制对共享资源的访问的重要工具。下面是一些在Linux中用于信号量操作的函数：

初始化和销毁信号量：

```
int sem_init(sem_t *sem, int pshared, unsigned int value): 初始化信号量。pshared 参数为 0 表示信号量在当前进程的多个线程之间共享，非 0 表示信号量在多个进程
int sem_destroy(sem_t *sem): 销毁信号量。注意，只有在信号量不再使用时才应调用此函数。
```

等待和发布操作：

```
int sem_wait(sem_t *sem): 对信号量执行 P（wait）操作，如果信号量的值大于 0，则将值减一；否则线程会阻塞，直到信号量值大于 0 为止。
int sem_post(sem_t *sem): 对信号量执行 V（signal）操作，将信号量的值加一。如果有线程在等待该信号量，一个等待的线程会被唤醒。
```

获取信号量的值：

```
int sem_getvalue(sem_t *sem, int *sval): 获取信号量的当前值，并将其存储在 sval 中。
```

示例代码：
下面是一个简单的示例代码，展示了如何在Linux中使用信号量实现线程同步，以解决生产者-消费者问题：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t empty, full;
int buffer = 0;

void* producer(void* arg) {
    for (int i = 0; i < 5; ++i) {
        sem_wait(&empty);
        buffer = i;
        printf("Produced: %d\n", buffer);
        sem_post(&full);
        usleep(500000);  // 模拟生产耗时
    }
    return NULL;
```

```
}
void* consumer(void* arg) {
    for (int i = 0; i < 5; ++i) {
        sem_wait(&full);
        printf("Consumed: %d\n", buffer);
        sem_post(&empty);
        usleep(300000);  // 模拟消费耗时
    }
    return NULL;
}

int main() {
    pthread_t producer_thread, consumer_thread;

    sem_init(&empty, 0, 1);  // 初始化信号量
    sem_init(&full, 0, 0);

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    sem_destroy(&empty);  // 销毁信号量
    sem_destroy(&full);

    return 0;
}
```

这个示例代码中，sem_init 初始化了两个信号量 empty 和 full，分别表示缓冲区为空和满的状态。生产者在生产时调用 sem_wait(&empty) 来等待空缓冲区，而消费者在消费时调用 sem_wait(&full) 来等待满缓冲区。生产者和消费者完成操作后，分别调用 sem_post 发布信号量，通知另一方可以进行操作。

介绍一下读写锁线程同步的相关函数

读写锁（Read-Write Lock）是一种用于线程同步的机制，它允许多个线程在不互斥地访问共享数据的情况下进行读取操作，但在进行写操作时必须互斥。在Linux中，读写锁通过读写锁函数进行操作。下面是一些在Linux中用于读写锁操作的函数：

初始化和销毁读写锁：

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr): 初始化读写锁。可以通过 attr 参数传递属性，通常可以设置为 NULL。

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock): 销毁读写锁。注意，只有在读写锁不再使用时才应调用此函数。
```

读操作：

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock): 对读写锁执行读取锁定操作。允许多个线程同时获得读取锁，只要没有线程拥有写锁。

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock): 尝试以非阻塞方式获取读取锁。
```

写操作：

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock): 对读写锁执行写入锁定操作。写锁是互斥的，当有线程拥有写锁或读锁时，其他线程将被阻塞。

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock): 尝试以非阻塞方式获取写入锁。
```

解锁操作：

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock): 释放读写锁。根据锁的类型（读锁或写锁），其他线程可以获得读锁或写锁。
```

示例代码：
下面是一个简单的示例代码，展示了如何在Linux中使用读写锁来实现线程同步，以解决读者-写者问题：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_rwlock_t rwlock;
int shared_data = 0;

void* reader(void* arg) {
    pthread_rwlock_rdlock(&rwlock);
    printf("Reader: %d\n", shared_data);
    pthread_rwlock_unlock(&rwlock);
    return NULL;
}

void* writer(void* arg) {
```

```
    pthread_rwlock_wrlock(&rwlock);
    shared_data++;
    printf("Writer: wrote data\n");
    pthread_rwlock_unlock(&rwlock);
    return NULL;
}

int main() {
    pthread_t reader_thread, writer_thread;

    pthread_rwlock_init(&rwlock, NULL);

    pthread_create(&reader_thread, NULL, reader, NULL);
    pthread_create(&writer_thread, NULL, writer, NULL);

    pthread_join(reader_thread, NULL);
    pthread_join(writer_thread, NULL);

    pthread_rwlock_destroy(&rwlock);

    return 0;
}
```

在这个示例代码中，pthread_rwlock_init 初始化读写锁，读者通过 pthread_rwlock_rdlock 获取读锁，而写者通过 pthread_rwlock_wrlock 获取写锁。读者可以同时获取读锁，而写者在获取写锁时会互斥。通过 pthread_rwlock_unlock 解锁读写锁，使其他线程可以继续获取锁。

互斥锁和读写锁的区别

互斥锁（Mutex Lock）和读写锁（Read-Write Lock）是两种不同类型的线程同步机制，用于管理多线程对共享资源的访问。它们在应用场景和行为上有一些区别，下面是它们之间的主要区别：

1)共享访问控制：

互斥锁： 互斥锁用于实现互斥访问，即同一时刻只允许一个线程访问临界区（共享资源）。当一个线程获得互斥锁后，其他线程需要等待锁的释放才能进入临界区。

读写锁： 读写锁允许多个线程同时进行读取操作，但在写入操作时需要互斥。这样可以提高并发性能，因为多个线程可以同时读取共享资源，但只有一个线程能够进行写入。

2)锁类型：

互斥锁： 互斥锁只有两种状态：锁定或解锁。只能被一个线程持有，其他线程必须等待锁的释放。

读写锁： 读写锁有两种状态：读取锁定和写入锁定。多个线程可以同时获得读取锁定，但写入锁定是互斥的，即只能有一个线程持有写入锁定。

3)性能方面：

互斥锁： 互斥锁的开销较大，因为每次只能有一个线程访问临界区，其他线程需要等待，可能导致性能下降。

读写锁： 读写锁在读取操作频繁、写入操作较少的情况下可以提供更好的性能，因为多个线程可以同时读取，只有写入时需要互斥。

4)适用场景：

互斥锁： 适用于在临界区内进行写操作的场景，例如更新共享资源。

读写锁： 适用于多线程环境下，读取操作远远多于写入操作的场景，例如数据缓存、数据库读取等。

5)死锁风险：

互斥锁： 存在死锁风险，如果多个线程循环等待锁定资源，可能导致程序无法继续运行。

读写锁： 由于读取锁定可以同时被多个线程持有，死锁的风险较小。但写入锁定需要互斥，因此在写操作时仍需要注意死锁。

综合考虑，互斥锁适用于强烈需要互斥访问的情况，而读写锁适用于读取操作频繁、写入操作较少的情况，可以在合适的场景中提高并发性能。选择合适的线程同步机制取决于应用的需求和性能考虑。

多线程实现并发服务器

pass

IO复用有哪些方式?

I/O 复用是一种技术，用于同时监视多个文件描述符的状态，以便在有事件发生时进行处理。以下是常见的几种 I/O 复用方式：

select：select 是一种较早的 I/O 复用方式，可以同时监视多个文件描述符的可读、可写和异常状态。它通过文件描述符集合来指定需要监听的文件描述符，然后在调用时会阻塞，直到指定的文件描述符中有一个或多个就绪，或者超过了设置的超时时间。

poll：poll 是对 select 的改进，也可以同时监视多个文件描述符的状态。与 select 不同的是，poll 使用一个结构体数组来描述需要监听的文件描述符，更加直观。类似于 select，poll 在调用时也会阻塞，直到有文件描述符就绪或超过超时时间。

epoll：epoll 是 Linux 特有的一种 I/O 复用方式，通过使用事件驱动的方式来监视文件描述符的状态。它使用红黑树和链表等数据结构来高效管理文件描述符，能够在大规模连接下有更好的性能表现。epoll 支持水平触发和边缘触发两种模式，边缘触发模式只在文件描述符状态变化时通知应用程序。

kqueue：kqueue 是类似于 epoll 的 I/O 复用机制，可在 BSD 系统（如 FreeBSD、macOS）上使用。它使用了类似于事件队列的数据结构，能够高效地处理大量并发连接。

IOCP：I/O 完成端口（IOCP，I/O Completion Ports）是 Windows 系统下的一种异步 I/O 复用机制。它使用了事件回调的方式，在有 I/O 完成时异步通知应用程序。IOCP 在 Windows 平台上用于高性能的异步 I/O 操作。

这些 I/O 复用方式在不同的操作系统上有不同的名称和实现，但基本原理相似：都是为了能够同时有效地处理多个文件描述符的状态变化，从而提高程序的性能和并发能力。选择合适的 I/O 复用方式取决于目标平台和应用程序的需求。（**看前三个即可**）

介绍一下IO复用的select

IO复用是一种提高网络编程效率的技术，它允许单个进程或线程同时监听多个输入流（例如套接字），并在有数据可读或可写时立即通知应用程序进行相应处理，而不需要阻塞等待每个输入流。
在传统的网络编程中，如果要同时处理多个套接字连接，常见的做法是使用多线程或多进程来处理每个连接，**但这样会导致资源消耗和管理复杂性的增加。IO复用技术通过在一个线程或进程中监听多个套接字，从而避免了频繁创建和销毁线程或进程的开销。**
其中，select函数是一个用于IO复用的系统调用（或库函数，取决于编程语言和操作系统），它的主要作用是检测一组套接字是否处于可读、可写或异常等状态，从而实现异步IO操作。
select函数的原型为：

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

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

参数说明：

nfds: 最大的文件描述符（套接字描述符）**加1**。

readfds: 用于检测可读事件的文件描述符集合。

writefds: 用于检测可写事件的文件描述符集合。

exceptfds: 用于检测异常事件的文件描述符集合。

timeout: 超时时间，用于设置select函数的阻塞时间，当为NULL时表示永久阻塞，直到有事件发生或调用被信号中断。

函数返回值：

当有事件发生时，select函数返回就绪文件描述符的数量。

如果超时时间到达而没有任何文件描述符就绪，返回0。

如果出现错误，返回-1，并设置errno变量以表示错误原因。

使用select函数时，需要注意以下几点：

在调用select函数前，需要将要监听的套接字加入到相应的文件描述符集合中。

在select函数返回后，可以通过遍历文件描述符集合来确定哪些套接字处于就绪状态，然后进行相应的读写操作。

select函数的效率在一定程度上取决于所监听的套接字数量，当套接字数量较多时，效率可能会降低。

介绍一下IO复用的poll

poll 是另一种用于实现 I/O 复用的系统调用，类似于 select，它也能够同时监听多个文件描述符的状态，并在有事件发生时通知应用程序进行相应处理。poll 在某些方面相对于 select 更为灵活，但也有一些限制。下面是关于 poll 函数的介绍：
poll 函数的原型为：

```
#include <poll.h>

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

参数说明：

fds：一个指向 pollfd 结构体数组的指针，每个结构体描述了一个文件描述符和所关注的事件。

nfds：fds 数组中元素的数量。

timeout：超时时间，单位为毫秒。当设置为负数时，poll 将一直阻塞，直到有事件发生；当设置为0时，poll 将立即返回，不论是否有事件发生；当设置为正数时，poll 将在超过指定时间后返回，不论是否有事件发生。

struct pollfd 结构体定义如下：

```
struct pollfd {
    int fd;       // 文件描述符
    short events; // 需要关注的事件
    short revents; // 实际发生的事件
};
```

fd：需要检测的文件描述符。

events：关注的事件，可以是 POLLIN（可读事件）、POLLOUT（可写事件）等组合。

revents：实际发生的事件，poll 在返回时会将发生的事件填充到该字段中。

函数返回值：

当有事件发生时，poll 返回就绪文件描述符的数量。

如果超时时间到达而没有任何文件描述符就绪，返回0。

如果出现错误，返回-1，并设置 errno 变量以表示错误原因。

使用 poll 函数的流程类似于 select 函数：

1.初始化 struct pollfd 数组，填充需要关注的文件描述符和事件。

2.调用 poll 函数，等待事件发生。

3.遍历 struct pollfd 数组，检查每个文件描述符的 revents 字段，确定发生的事件类型。

4.根据发生的事件类型，执行相应的读写操作。

相比于 select，poll 的一些优点包括：

没有文件描述符数量的限制，能够适用于较大数量的文件描述符。

不需要使用位图，使用结构体数组更加直观。

然而，poll 也有一些限制，如效率问题，特别是在监听大量文件描述符时。在实际使用时，可以根据具体的需求选择使用 select 还是 poll。

介绍一下IO复用的epoll

epoll 是 Linux 下的一种高效的 I/O 复用机制，与传统的 select 和 poll 相比，它在大规模连接的情况下具有更好的性能表现。epoll 使用了事件驱动的方式，能够有效地管理和监控大量的文件描述符，同时支持水平触发（LT）和边缘触发（ET）两种模式。
epoll 提供了三个相关的系统调用：epoll_create、epoll_ctl 和 epoll_wait。

1)epoll_create：创建一个 epoll 实例，返回一个文件描述符用于引用该实例。

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

int epoll_create(int size);
```

参数 size 用于指定在内核中管理的文件描述符的数量上限。这个参数在 epoll 中并不是一个固定的限制，只是一个给内核的一个提示。

2)epoll_ctl：用于向 epoll 实例中添加、修改或删除需要监听的文件描述符。

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

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

epfd：epoll 实例的文件描述符。

op：操作类型，可以是 EPOLL_CTL_ADD（添加文件描述符）、EPOLL_CTL_MOD（修改文件描述符）或 EPOLL_CTL_DEL（删除文件描述符）。

fd：需要添加、修改或删除的文件描述符。

event：一个 epoll_event 结构体，用于描述需要监听的事件类型。

3)epoll_wait：等待事件发生，类似于 select 和 poll。

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

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

epfd：epoll 实例的文件描述符。

events：一个数组，用于存放发生事件的信息。

maxevents：数组的大小，表示最多可以接收多少个事件。

timeout：超时时间，单位为毫秒。当设置为负数时，epoll_wait 将一直阻塞，直到有事件发生；当设置为0时，epoll_wait 将立即返回，不论是否有事件发生；当设置为正数时，epoll_wait 将在超过指定时间后返回，不论是否有事件发生。

struct epoll_event 结构体定义如下：

```c
struct epoll_event {
    uint32_t events; // 事件类型
    epoll_data_t data; // 用户数据，可以是文件描述符或指针等
};
```

events：需要监听的事件类型，例如 EPOLLIN（可读事件）和 EPOLLOUT（可写事件）等。

data：用户数据，可以是文件描述符或者指针，用于标识相关事件的源。

epoll 具有一些优点，如：

在大规模连接下性能表现优异，能够高效处理大量并发连接。

支持水平触发和边缘触发两种模式，边缘触发模式能够更加精确地监测事件。

然而，epoll 仅在 Linux 系统上可用，不能在其他操作系统中使用。在高并发场景下，使用 epoll 能够显著减少系统开销，提高应用程序的性能。

分析一下select poll epoll三者优缺点

这三种 I/O 复用机制，即 select、poll 和 epoll，都是用于实现在一个进程或线程中同时监听多个文件描述符，以便在有事件发生时通知应用程序进行处理。它们的实现和特性有所不同，下面是它们的主要区别：

1）性能和扩展性：

select 和 poll 都有一个较为明显的性能问题，当要监听的文件描述符数量增加时，会导致效率下降，因为每次调用时都需要遍历整个文件描述符集合。

epoll 在大规模连接下有更好的性能表现，因为它使用了红黑树和链表等数据结构来管理文件描述符，能够高效处理大量并发连接。而且 epoll 使用事件驱动的方式，只有就绪的文件描述符会被返回，减少了不必要的遍历。

2）支持的平台：

select 可以在多个操作系统上使用，但有一些平台可能对文件描述符数量有限制。

poll 也可以在多个操作系统上使用，相对于 select，它没有文件描述符数量的限制，但效率仍然会受到影响。

epoll 是 Linux 特有的，因此只能在 Linux 系统上使用。

3）事件通知方式：

select 和 poll 使用轮询方式，即程序需要不断地查询文件描述符是否就绪。

epoll 使用事件驱动方式，当文件描述符就绪时，内核会主动通知应用程序。

4）事件触发方式：

select 和 poll 都是水平触发模式，即当文件描述符就绪时，只要有数据可读或可写，它们会一直通知应用程序，直到应用程序处理完所有数据。

epoll 可以选择水平触发模式（LT）或边缘触发模式（ET）。在边缘触发模式下，只有在文件描述符状态变化时才会通知应用程序，需要应用程序自己保证读写操作的完整性。

5）代码维护性和易用性：

select 的接口较为古老，使用时需要维护文件描述符集合和设置状态的位图。

poll 对 select 进行了一定的改进，使用结构体数组更加直观。

epoll 提供了更简洁的接口，对大规模连接下的性能需求提供了更好的支持。

总的来说，虽然 select 和 poll 可以在多个平台上使用，但在性能和扩展性方面存在限制，适用于较小规模的并发连接。**而 epoll 在 Linux 系统下具有显著的性能优势，适用于需要处理大量并发连接的高性能应用。选择合适的 I/O 复用机制取决于应用程序的需求和目标平台。**

介绍一下Reactor和proactor

当谈到并发编程中的 I/O 复用时，Reactor 和 Proactor 是两种常见的模式，它们都可以用于处理多个 I/O 操作，但它们在处理方式上有所不同。

10）Reactor 模式：

Reactor 模式是一种事件驱动的并发模式，主要用于处理多个事件和 I/O 操作。

在 Reactor 模式中，有一个 Reactor 负责监听多个事件源（例如文件描述符），一旦有事件就绪，Reactor 将通知相应的处理器来处理这些事件。

事件处理是由应用程序自己的代码完成的，Reactor 只负责分发事件，处理事件的细节由应用程序决定。

2）Proactor 模式：

Proactor 模式也是一种事件驱动的并发模式，同样用于处理多个事件和 I/O 操作，但在处理方式上有所不同。

在 Proactor 模式中，有一个 Proactor 负责监听多个事件源，但与 Reactor 不同，Proactor 在事件就绪后会主动发起事件的处理，而不是将事件分发给应用程序。

事件的实际处理是由框架或系统完成的，应用程序只需提供相应的回调函数或处理函数，Proactor 负责调用这些函数来完成事件处理。

关于 I/O 复用，两种模式都可以用于实现 I/O 复用，即同时处理多个 I/O 操作。Reactor 模式和 Proactor 模式的区别在于谁来完成事件的实际处理，以及处理方式的不同。Reactor 模式中，应用程序自身负责事件的处理；而 Proactor 模式中，底层系统或框架负责实际的事件处理，应用程序只需提供处理函数。
总结起来，Reactor 和 Proactor 都是并发编程中用于处理多个 I/O 操作的模式，它们在事件处理方式上有所不同，适用于不同的应用场景

在编写基于 Reactor 或 Proactor 模式的并发程序时，你会使用不同的函数和工具来实现事件处理和 I/O 操作。以下是一些常见的实现函数和工具，这些函数可以在不同编程语言和框架中找到：
Reactor 模式实现函数和工具：

select/poll/epoll/kqueue：这些是用于多路复用的系统调用，用于监听多个文件描述符上的事件，并在事件就绪时通知应用程序。

Socket 接口函数：用于创建、连接、发送和接收数据的函数，如 socket()、bind()、connect()、send()、recv() 等。

事件处理函数：由应用程序编写的处理函数，用于处理特定类型的事件。这些函数会在事件发生时被 Reactor 调用。

事件循环：用于循环监听事件并调用相应处理函数的主循环，通常是一个无限循环。

Proactor 模式实现函数和工具：

异步 I/O 函数：框架或系统提供的异步 I/O 函数，用于发起异步的 I/O 操作，如异步读写数据。

事件处理函数/回调函数：由应用程序提供的处理函数或回调函数，用于在事件就绪时执行特定的操作。

事件循环：类似于 Reactor 模式，也需要一个循环来监听事件就绪并调用相应的处理函数。

线程池/任务调度器：用于处理 Proactor 模式中的异步操作，可以使用线程池或任务调度器来管理和执行异步任务。