第三章、并发技术上篇：多进程与多线程

第一节、前置知识概述：

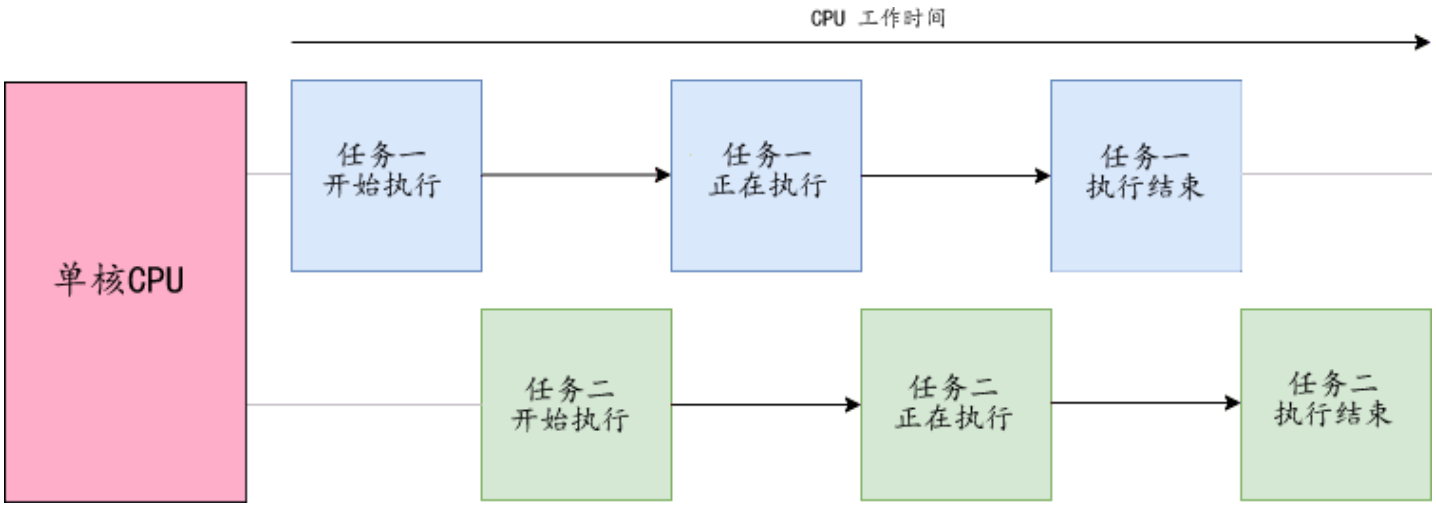
1.并发与并行

并发服务器和并行服务器是指可以处理多个请求的服务器，它们之间的区别在于请求的处理方式和服务器硬件资源的利用方式不同。

1.1并发

> 并发服务器可以同时处理多个请求，但它们是在同一个处理器或者计算机核心上交替执行的。在并发服务器中，服务器可以同时响应多个请求，但是每个请求必须等待前一个请求完成之后才能继续执行，因为它们共享同一个处理器资源。这种方式可以通过使用多线程、多进程、事件驱动等技术来实现，可以充分利用计算机的多核心资源。

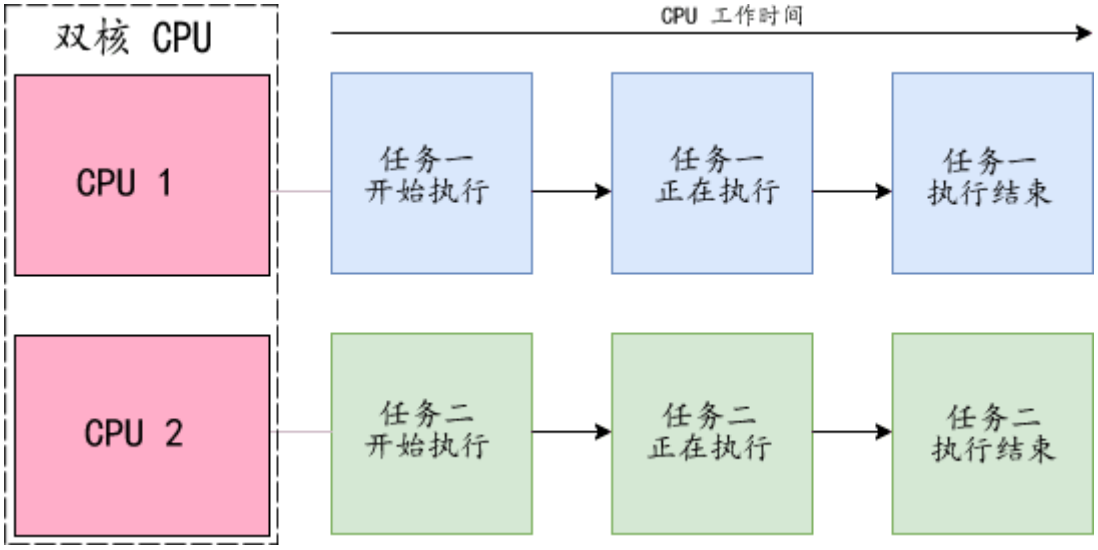
并发图解：



虽然 CPU 在同一时刻只能执行一个任务，但是通过将 CPU 的使用权在恰当的时机分配给不同的任务，使得多个任务在视觉上看起来是一起执行的。CPU 的执行速度极快，多任务切换的时间也极短，用户根本感受不到，所以并发执行看起来才跟真的一样。

1.2并行

> 而并行服务器则是可以同时处理多个请求，并且这些请求是在多个处理器或计算机核心上并行执行的。在并行服务器中，不同的请求可以被分配到不同的处理器或计算机核心上并行执行，因此可以更快地完成任务。这种方式可以通过使用分布式计算、集群等技术来实现，可以充分利用多台或多核计算机的计算资源。



双核 CPU 执行两个任务时，每个核心各自执行一个任务，和单核 CPU 在两个任务之间不断切换相比，它的执行效率更高。

1.3总结：

因此，**单核CPU上运行多进程或多线程，只能实现并发执行；如果在多核CPU上运行多进程或多线程，可以实现并行执行。**虽然并发服务器和并行服务器都可以处理多个请求，但它们的处理方式和资源利用方式是不同的。并发服务器主要是在单个处理器或计算机核心上交替处理多个请求，而并行服务器则是在多个处理器或计算机核心上同时处理多个请求。

> **注意：大家最近常听说的两个概念：**CUDA（Compute Unified Device Architecture）和OpenCL（Open Computing Language）都是针对GPU并行计算的编程模型和框架，它们的目标都是提高GPU的计算性能和吞吐量，这两个并行计算框架主要用于人工智能模型训练之中，图形图像处理，及大数据的并行处理之中。因为GPU所支持的运算数据类型相较于CPU有限，目前对浮点类型支持较好。CUDA的GPU并行框架只能用于英伟达的硬件之上。而OpenCL是一个通用的GPU并行计算框架，几乎对硬件没有限制，但在不同硬件上发挥的性能不一样，需要程序员们进行优化才可以。

3.IO操作：

IO操作是计算机系统中的一个重要组成部分，它包括输入和输出两个方面。在网络编程中，IO操作通常指数据的读取和发送等操作。常见的IO操作方式包括阻塞式IO、非阻塞式IO、异步IO和IO多路复用等方式。（可以类比取快递的方式）

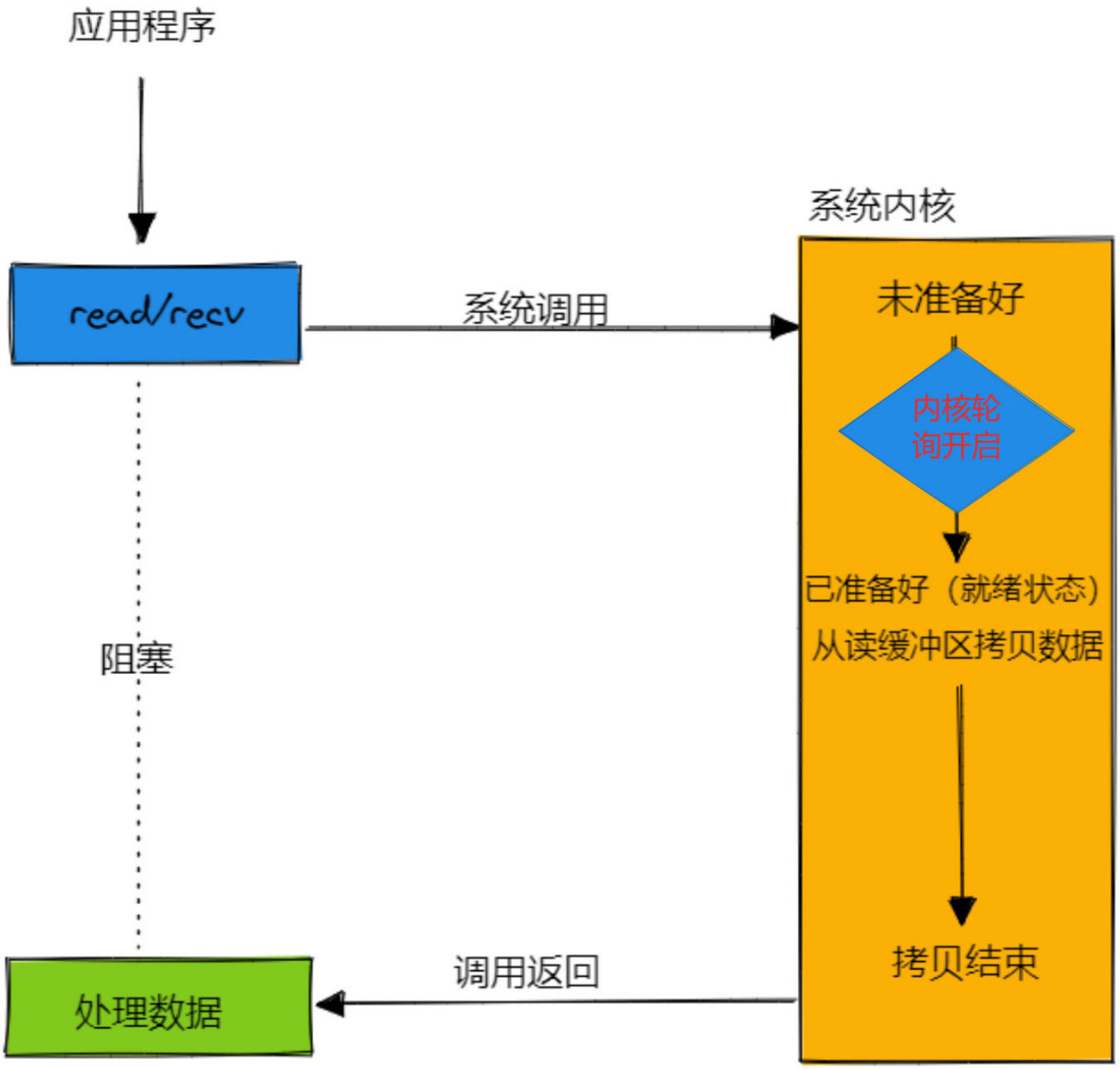
3.1、阻塞式IO

阻塞式IO是一种同步IO操作方式，即在进行IO操作时，程序会一直阻塞等待，直到IO操作完成后才会继续执行下一步操作。在网络编程中，通常会使用阻塞式IO进行数据的读取和发送等操作。阻塞式IO的优点是操作简单易用，缺点是只能处理一个IO操作，无法同时处理多个连接，可能会导致程序的效率低下。（我们之前的accept, connect,read,write都是阻塞IO）

认识内核事件轮询机制：

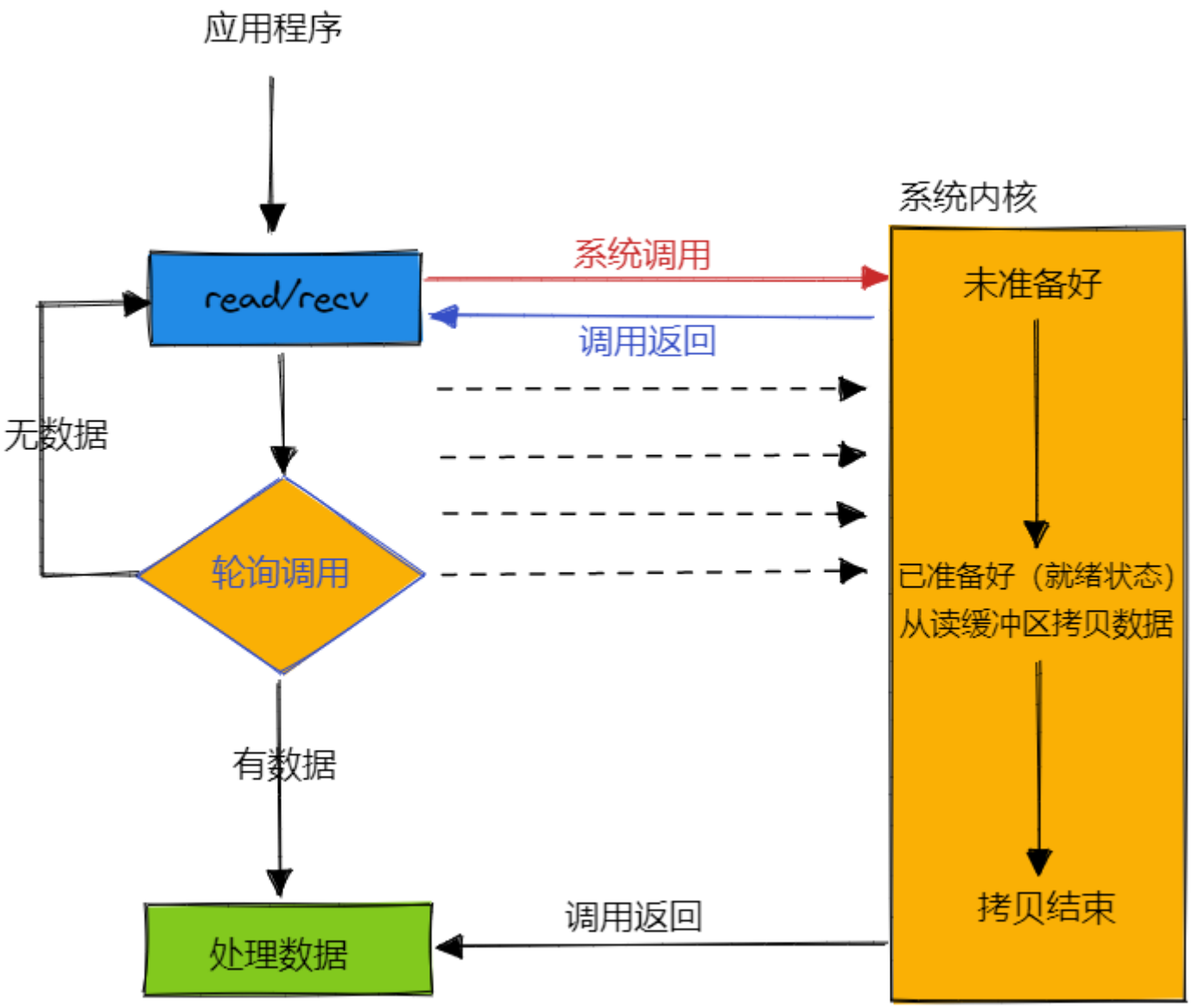
> 内核事件轮询检测（Kernel polling）是一种用于检测设备或文件描述符是否有数据可读或可写的技术。
> 比如，一般情况下，在计算中，当一个进程需要从非阻塞IO描述符中读取或写入数据时，通常会使用一个循环，不停的检测IO接口描述符中是否有数据产生，如果有就读取，如果没有就继续循环检测，但这样会大量的无效轮询，浪费大量的CPU资源。

内核事件轮询检测技术可以解决这个问题：**它通过在内核中注册一个回调函数，当设备或文件描述符的状态发生变化时，内核会调用该回调函数，通知进程可以进行读取或写入操作。这种方式避免了进程在等待过程中的无效轮询，从而提高了CPU的利用率。**



3.2、非阻塞式IO

非阻塞式IO是一种异步IO操作方式，即在进行IO操作时，程序会立即返回，并继续执行其他操作，而不是一直等待IO操作完成。程序可以通过不断查询IO操作的状态，来检测IO操作是否完成。在网络编程中，通常会使用非阻塞式IO配合轮询或信号等方式，来实现多个连接的同时处理。非阻塞式IO的优点是可以处理多个连接，提高程序的并发性和响应速度，缺点是增加了程序的复杂度，需要进行状态查询，可能会浪费CPU资源。



在Linux中，可以使用fcntl系统调用来设置非阻塞IO。具体步骤如下：

fcntl函数介绍：

```
fcntl是Unix/Linux系统下的一个系统调用函数，全称：file control
用于对已经打开的文件描述符进行一些控制操作，如复制文件描述符，修改文件状态标志等。
fcntl函数的原型如下：
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
其中，fd参数是需要操作的文件描述符，cmd参数是控制命令，
arg是一个可选的参数，具体的含义和使用方式取决于cmd参数。如果不需要额外参数可写0或不写。
fcntl函数的常用控制命令如下：
F_DUPFD：复制文件描述符，可以用来获取一个新的文件描述符，
        该文件描述符与原来的描述符指向同一个文件。
F_GETFD：获取文件描述符的标记值（close-on-exec）。
F_SETFD：设置文件描述符的标记值。
F_GETFL：获取文件状态标志。
F_SETFL：设置文件状态标志。
F_GETLK：获取文件锁信息。
F_SETLK：设置文件锁。
```

```
F_SETLKW：设置文件锁，如果无法获取锁，则阻塞等待锁的释放。
F_GETOWN：获取文件所有者的进程。
F_SETOWN：设置文件所有者的进程。
其中，文件状态标志可以通过以下常量进行设置：
O_APPEND：写入时追加到文件末尾。
O_NONBLOCK：以非阻塞方式打开文件。
O_SYNC：强制写入到磁盘。
O_ASYNC：启用异步通知。
文件锁分为共享锁和独占锁两种类型，
可以通过fcntl函数设置和获取文件锁信息，
实现文件读写的同步控制。
```

1. 首先，需要打开需要进行非阻塞IO操作的文件描述符。
2. 然后，使用fcntl系统调用来获取原来的文件状态标志（file status flags），通常是O_RDONLY、O_WRONLY或O_RDWR。这个文件状态标志包含了一些特定的标志位，例如O_NONBLOCK，表示是否设置了非阻塞IO模式。
3. 如果原来的文件状态标志不包含O_NONBLOCK标志位，就需要使用fcntl系统调用设置非阻塞IO模式。可以使用F_SETFL命令来设置文件状态标志，同时将O_NONBLOCK标志位与原来的标志位进行或运算。
4. 如果需要恢复阻塞IO模式，可以使用相同的方法，但是将O_NONBLOCK标志位设置为0，先取反，再&=按位与运算。

为什么使用|= 与&=来设置标识的底层位域运算原理：

> 举例：假设我们有一个8位的二进制数**x**，它的二进制表示为**xxxx xxxx**，现在想要将它的第3位设置为1，可以使用**|=**操作，即：
> x |= 0x04; // 将二进制数的第3位设置为1
> 这个操作会将0x04（二进制数为0000 0100）和**x**进行**按位或**操作，得到的结果就是将原来的第3位变为1，其它位不变。
>
> 而如果想要将**x**的第3位清除为0，可以使用**&=**操作，注意**先按位取反**，再**按位与**即：
> x &= ~0x04; // 将二进制数的第3位清除为0
>
> 这个操作会**先**将0x04进行**按位取反**操作，得到的结果是0xFB（二进制数为1111 1011），**然后将其**和**x**进行**按位与**操作，即可将**x**的第3位清除为0，其它位不变。

演示一个非阻塞IO通过fcntl设置为非阻塞的示例：

当程序执行到读操作的时候，如果缓冲区里面有内容，则读取内容继续向下执行。

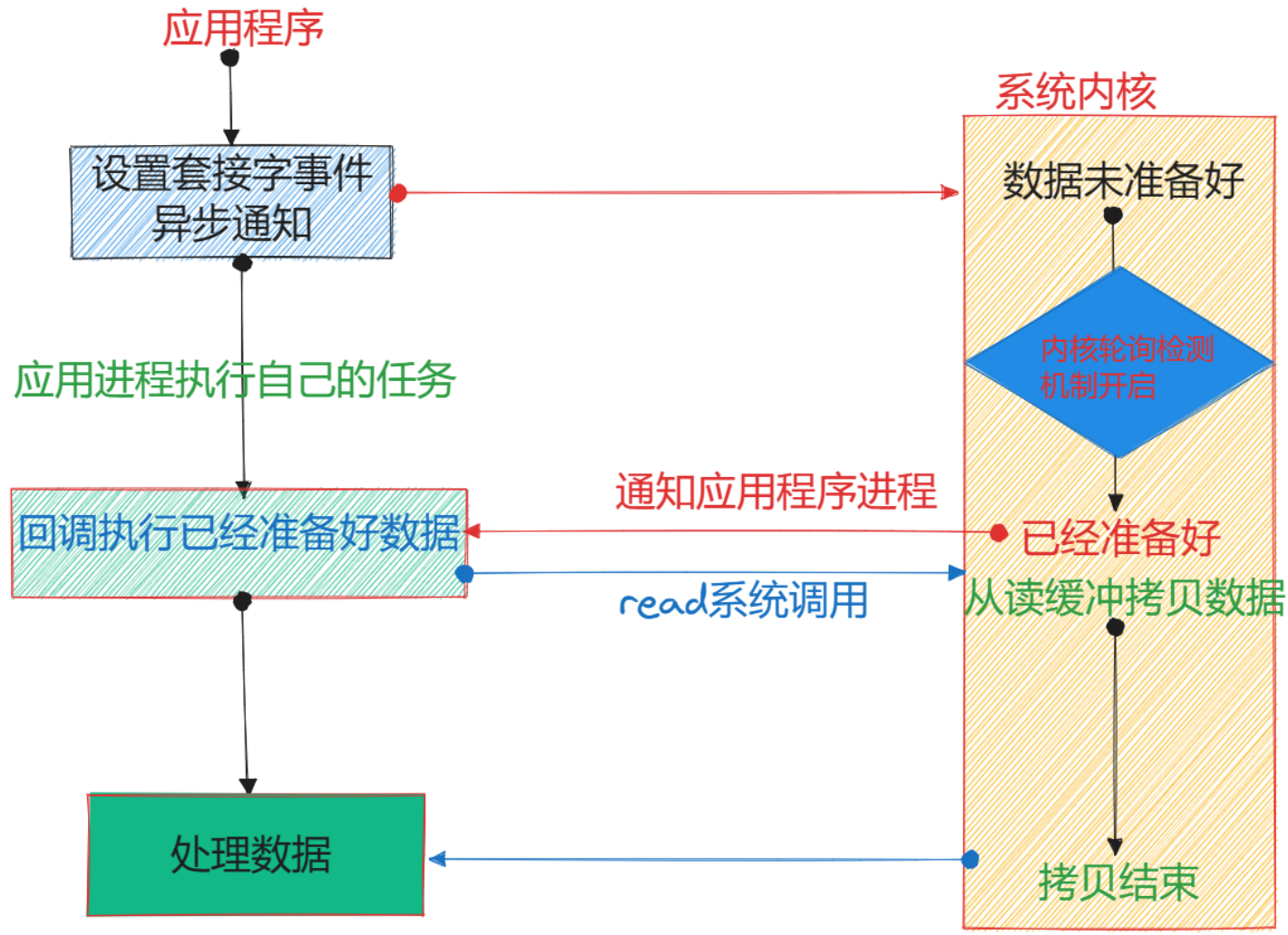
如果缓冲区中没有内容，则进程进入休眠状态，直到内核缓冲区中有内容了，内核唤醒此进程，读取内容继续向下执行。（使用udp服务器的recvform函数为例演示）

3.3、异步IO，依赖于操作系统的内核中的事件检测机制

异步IO是一种另外的异步IO操作方式，**与非阻塞式IO相比，异步IO通过回调函数的方式来处理IO操作的完成事件，避免了阻塞和轮询等开销，程序的效率更高**。异步IO的优点是能够处理大量的并发IO请求，适合高并发场景，缺点

是增加了程序的复杂度，代码可读性差，难以调试和维护。

异步模式是一种编程模式，其特点是在调用一个耗时的操作时不会阻塞程序的执行，而是通过回调函数的方式在操作完成后再通知程序进行处理。在异步模式下，程序可以同时处理多个操作，提高了程序的并发性和响应速度。



异步模式通常使用事件循环来实现，程序通过注册事件和回调函数的方式告诉事件循环要监听哪些事件，当事件发生时，事件循环会自动调用相应的回调函数进行处理。事件循环可以使用操作系统提供的API（如**epoll**、kqueue等）或者第三方库（如libevent、libuv等）来实现。

异步模式在网络编程、GUI编程、多媒体处理等场景下非常常见，可以有效提高程序的并发性和响应速度。然而，异步模式也增加了程序的复杂性，需要处理回调函数的嵌套和错误处理等问题。

代码示例：

```c
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <signal.h>
// 修改文件描述符的行为或属性的方式：
// fcntl函数：
int set_nonblock(int fd)
{
```

```
    //获取文件描述符操作状态
    int flags = fcntl(fd, F_GETFL, 0);
    //增加文件描述符操作状态为非阻塞.
    flags |= O_NONBLOCK;
    //用新的状态设置文件描述符。
    if (fcntl(fd, F_SETFL, flags) == -1)
    {
        perror("fcntl err:");
        return -1;
    }
    return 0;
}

// 当异步信号发出时，自动执行的函数：槽函数：
static int mouse_fd = -1;
void async_slots_functions()
{
    char buf[128] = {0};
    int nbytes = read(mouse_fd, buf, sizeof(buf) - 1);
    if (nbytes == -1)
    {
        perror("read err:");
        usleep(10000);
    }
    static int i = 0;
    printf("%d读取了%d \n",i++, nbytes);
}

// 设置异步IO通知：
int set_async(int fd)
{
    //1.获取文件描述符的操作行为的状态。
    int flags = fcntl(fd, F_GETFL, 0);
    //2.添加异步通知操作状态。
    flags |= O_ASYNC;
    //3.设置状态到描述符之中
    if (fcntl(fd, F_SETFL, flags) == -1)
    {
        perror("fcntl err:");
        return -1;
    }
    //4.设置文件描述符异步通知的进程为：当前进程。
    if (fcntl(fd, F_SETOWN, getpid()) == -1)
    {
```

```
        perror("fcntl err:");
        return -1;
    }
    //5.设置异步通知回调的执行函数：异步信号处理槽函数：
    signal(SIGIO, async_slots_functions);
    return 0;
}

int main(int argc, char const *argv[])
{
    mouse_fd = open("/dev/input/mouse0", O_RDONLY);
    if (mouse_fd == -1)
    {
        perror("open err:");
        return -1;
    }
    // 用户数据缓冲区：
    char buf[128] = {0};
    // set_nonblock(mouse_fd);
    set_async(mouse_fd);
    while (true)
    {
        static int i = 0;
        printf("%d大家好,才是真的好！\n", i++);
        sleep(1);
    }
    return 0;
}

```

3.4、IO多路复用

IO多路复用本身是使用单个线程来同时处理多个连接的IO请求。在网络编程中，常见的IO多路复用方式包括select、poll和epoll等。IO多路复用的优点是可以避免创建大量的线程或进程来处理IO请求，提高了程序的效率，同时可以处理大量的并发IO请求。缺点是增加了程序的复杂度，需要进行状态查询，同时IO多路复用的实现方式不同，可能会存在性能差异。

总体来说，不同的IO操作方式各有优缺点，应根据具体场景和需求选择合适的方式。例如，在高并发的网络编程中，可以使用IO多路复用技术来实现非阻塞式IO操作，同时避免了线程和进程的创建，提高了程序的效率。而在低并发的场景中，可以使用阻塞式IO操作，简单易用。在实现高性能和高并发的网络编程中，可以结合使用多种IO操作方式，如使用异步IO技术来处理大量的并发IO请求，使用IO多路复用技术来实现非阻塞式IO操作，提高程序的效率和响应速度。

在不同的IO操作方式中，非阻塞式IO和IO多路复用技术通常被使用在高并发的网络编程中。非阻塞式IO通过轮询或信号等方式来实现多个连接的同时处理，可以减少阻塞等待IO操作完成的时间，提高程序的效率。IO多路复用技术则是通过一个线程来同时处理多个连接的IO请求，减少了线程和进程的创建，降低了系统开销，提高了程序的效率和响应速度。

总的来说，不同的IO操作方式都有各自的优缺点，需要根据具体的场景和需求进行选择和使用。同时，也需要在实践中不断调整和优化，以达到更好的性能和可靠性。

第二节、多进程并发服务器：

2.1认识fork：

在Linux中，当一个进程调用**fork()**系统调用创建一个子进程时，**子进程将会复制父进程的内存映像，包括代码段、数据段、堆栈**等。**子进程与父进程会共享内核空间中的代码段、数据段以及其他内核数据结构，如进程表、文件表**等。

这个结论可以在Linux的手册页中找到，具体可以使用**man 2 fork**命令查看。其中关于进程创建的描述如下：

> On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.The child process and the parent process run in separate memory spaces. At the time of fork(2), the entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of **pthread_atfork(3)** may be helpful for dealing with problems that this can cause.

可以看到，文档中明确指出，父进程和子进程在不同的内存空间中运行，但在**fork()**时，父进程的整个虚拟地址空间都被复制到子进程中。

因此，子进程与父进程之间会共享内核空间，但是它们各自拥有自己的用户空间，因此它们可以独立地进行内存管理和其他操作。

```c
#include <unistd.h>
pid_t pid = fork(void);
if (pid == 0) {
    // 子进程代码
} else if (pid > 0) {
    // 父进程代码
} else {
    // 创建子进程失败的处理代码
}
```

> fork函数介绍：
> 该函数的作用是创建一个新的进程，该进程是调用进程（即父进程）的一个副本。新进程称为子进程，与父进程具有相同的代码段、数据段、堆和栈，但是拥有自己的地址空间和资源。
> fork()函数会返回两次，一次在父进程中，一次在子进程中。在父进程中，fork()函数返回子进程的进程ID（PID），在子进程中，fork()函数返回0。因此，父进程可以通过fork()的返回值判断是否是子进程，子进程则可以通过返回值判断自己是哪个子进程。
> 在多进程编程中，fork()函数非常重要，可以通过fork()函数创建新的进程来并发执行代码，实现多任务处理。
> 强调一下：fork()函数在子进程中返回的是0，而在父进程中返回的是新子进程的进程ID。因此，父进程可以通过检查 fork()函数的返回值是否为0来确定自己是否正在运行子进程的代码。如果返回值是0，则说明当前进程是子进程；如果返回值大于0，则说明当前进程是父进程。

分叉（fork）是UNIX术语，当分叉(fork)一个进程（一个运行的程序）时，基本上是复制了它，并且分叉后的两个进程都从当前的执行点继续运行，并且每个进程都有自己的内存副本（比如变量）。一个进程（原来的那个）成为父进程，另一个（复制的）成为子进程。**如果你是一个科幻小说迷，可以把它们想象成平行宇宙（parallel world）。**

分叉操作在时间线（timeline）上创建了一个分支，最后得到了两个独立存在的进程。幸好进程可以判断哪个是原进程哪个是子进程（通过查看fork函数的返回值）。因此它们所执行的操作不同（如果相同，那么还有什么意义？）。

在一个使用分叉（fork）的服务器中，每一个客户端机连接都利用分叉(fork)创造一个子进程。父进程继续监听新的连接，同时子进程处理客户端。当客户端的请求结束时，子进程就退出了。因此分叉的进程是并行运行的，客户端之间不必互相等待。

因为分叉有点耗费资源（每个分叉(fork)出来的进程都需要自己的内存），这就存在了另一个选择：线程。线程是轻量级的进程或子进程，所有的线程都存在于相同的（真正的）进程中，共享内存。资源消耗的下降伴随着一个缺陷：因为线程共享内存，所以必须确保它们的变量不会冲突，如果在同一时间修改同一内容，这就会造成混乱。这些问题都可以归结为同步问题。在现代操作系统中（Windows除外，它不支持分叉），分叉实际是很快的，现代的硬件能比以往更好地处理资源消耗。如果不想被同步问题所困扰，分叉是一个很好的选择。

> **fork()** 函数在操作系统中的重要性不仅体现在它的功能上，还因为它对 Unix 操作系统的设计哲学产生了深远的影响。在 Unix 系统中，所有进程都是通过 **fork()** 函数创建的。这种基于进程的设计哲学使得 Unix 操作系统具有高度的灵活性和可扩展性，可以实现非常复杂的系统和应用程序。
> 虽然 **fork()** 是一个强大而灵活的函数，但也存在一些缺点。比如，复制虚拟地址空间和进程资源需要耗费大量的时间和内存，特别是在父进程和子进程之间共享的资源较多时，这种复制操作的开销更大。此外， **fork()** 函数也容易导致一些资源竞争问题，比如共享的文件描述符和网络连接等，需要开发人员注意避免。

2.2子进程结束后注意回收其资源，避免僵尸进程占用资源。

> **Linux中使用ps -aux命令查看进程状态：**
> ps -aux命令显示的状态列中的
> D 不可中断 Uninterruptible sleep (usually IO)
> R 正在运行，或在队列中的进程
> S 处于休眠状态
> < 高优先级
> N 低优先级
> L 有些页被锁进内存
> s 包含子进程
> - 位于后台的进程组；
>
> l 多线程，克隆线程 multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
> T 停止或被追踪
> **Z 僵尸进程**
> W 进入内存交换（从内核2.6开始无效）
> X 死掉的进程
> 在服务器程序中，很会让父进程去阻塞等待并回收子进程的资源，而更多使用信号的方式，进行回调，然后在回调函数中回收子进程的资源。
> 认识回收子进程资源的函数 waitpid函数

```c
#include <unistd.h>
#include <stdio.h>
#include <wait.h>
#include <stdlib.h>
int main(int argc, char const *argv[])
{
    int pid = fork();
    if(pid > 0)
    {
```

```
        //父进程:
        printf("阻塞等待子进程的退回");
        int pid = waitpid(-1,NULL,0);
        printf("回收了子进程%d的资源",pid);
    }
    else if(pid == 0)
    {
        //子进程:
        for(int i = 0; i < 10; i++)
        {
            printf("子进程输出:%d \n",i);
            sleep(1);
        }
        exit(0);
    }
    else{
        //子进程创建失败:
        perror("fork err:");
        exit(-1);
    }
    return 0;
}

```

2.21 waitpid函数:

waitpid() 函数用于(默认为阻塞)等待一个指定的子进程结束或者改变其状态。该函数的原型如下:

```
pid_t waitpid(pid_t pid, int *status, int options);

```

- pid：要等待的子进程的进程 ID。如果 pid 的值为 -1，则表示等待任何子进程结束。如果 pid 的值大于 0，则表示等待进程 ID 为 pid 的子进程结束。如果 pid 的值为 0，则表示等待与调用进程处于同一个进程组的任何子进程结束。
- status：一个指向 int 类型的指针，用于存储子进程的退出状态或终止信号。如果不关心子进程的退出状态或终止信号，可以将该参数设为 NULL。
- options：指定额外的选项。通常将其设置为 0即为阻塞模式。**如果将此函数设定为非阻塞，则options = WNOHANG，（wait no hang不挂起等待）此函数会立即返回。如果使用非阻塞WNOHANG，就需要一种轮询检测返回值，以判断是否回收了子进程资源。如果返回0则表示没有可回收的子线程，如果返回子线程线程号，则表示回收了此子线程的资源。如果是一个父进程，有多个子进程时，可以使用 waitpid(-1,NULL,WNOHANG) 来结束所有子线程。**

该函数的返回值为等待的子进程的进程 ID。如果出现错误，返回值为 **-1**。

> 调用 **waitpid()** 函数会使父进程暂停执行，直到子进程结束或者改变其状态。如果子进程已经结束，那么 **waitpid()** 函数会立即返回，否则会一直等待子进程结束。当子进程结束时，父进程可以通过 **status** 指针获取子进程的退出状态或终止信号。如果父进程不关心子进程的退出状态或终止信号，可以将 **status** 参数设置为 **NULL**。

2.22 注册信号函数的使用：

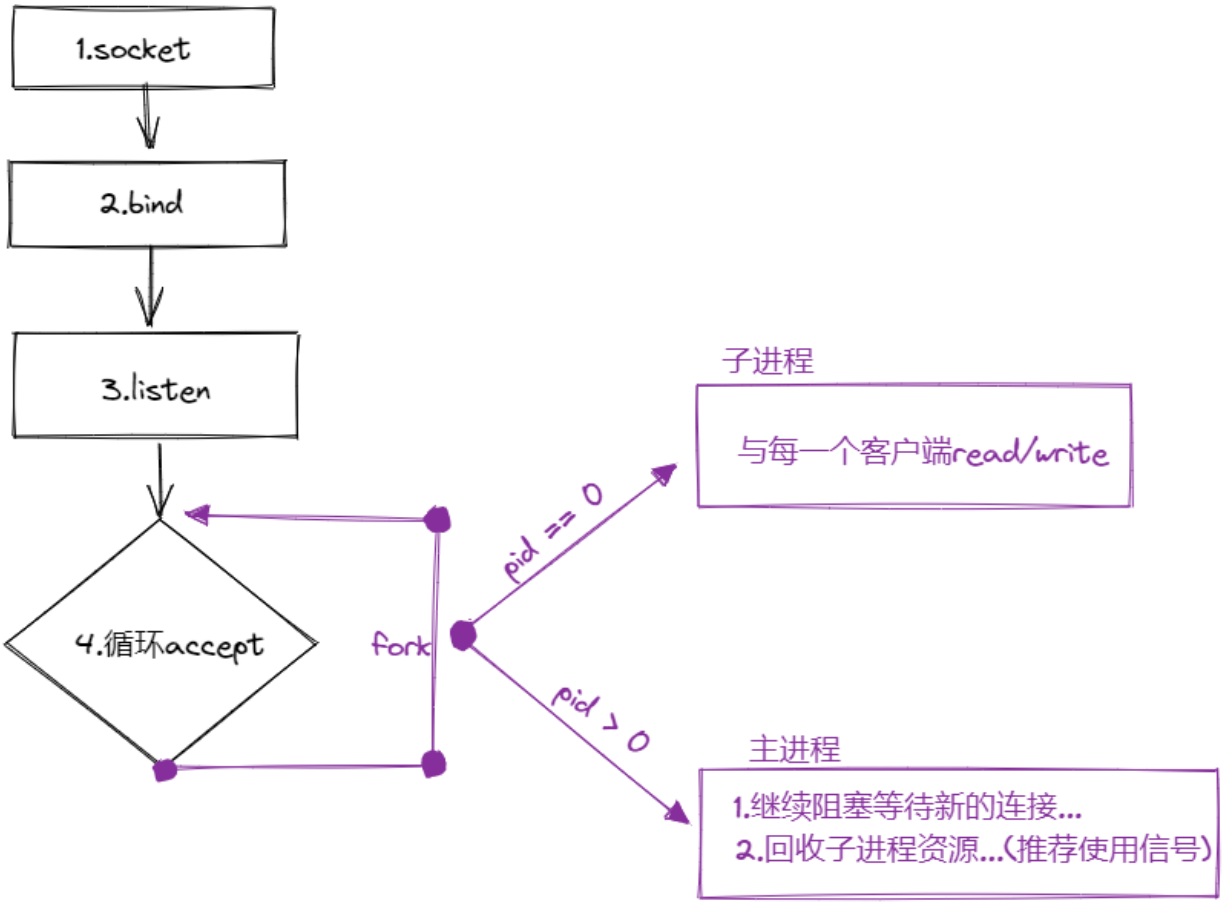
```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
sigaction()函数有三个参数：
signum参数:表示所要处理的信号的编号。
act参数:是一个指向struct sigaction类型的结构体指针，该结构体定义了对该信号的处理方式，
包括指定处理函数、处理标志等。如果该参数为NULL，则忽略信号。
oldact参数是一个指向struct sigaction类型的结构体指针，用于保存原来的信号处理方式。
如果该参数为NULL，则忽略原来的信号处理方式。
返回值：
sigaction()函数成功时返回0，失败时返回-1。

在使用注册信号函数的时候，注意，注册信号的进程不要提前退出。不然进程中的回调函数是无法得到执行的
因为回调函数的执行，也需要使用此进程的栈空间哦。

sigaction结构体内部结构如下：
struct sigaction {
    void (*sa_handler)(int); // 处理程序的地址或者 SIG_IGN、SIG_DFL
    sigset_t sa_mask;        // 处理程序执行期间要阻塞的信号集
    int sa_flags;            // 处理程序的标志，如 SA_RESTART、SA_SIGINFO 等
    void (*sa_sigaction)(int, siginfo_t *, void *); // 处理程序的地址（如果 sa_flags 中包含
};
```

```
首先了解一下sigaction的结构体：
sigaction {
    void (*sa_handler)(int);          // 信号处理函数指针
    void (*sa_sigaction)(int, siginfo_t *, void *); // 替代的信号处理函数指针
    sigset_t sa_mask;                 // 额外屏蔽的信号集
    int sa_flags;                     // 用于指定信号处理的标志
    void (*sa_restorer)(void);         // 过时的恢复函数指针
};
以下是各个字段的说明：
sa_handler：这是一个函数指针，用于处理信号。当信号发生时，操作系统会调用这个函数进行处理。函数的
```

```
sa_sigaction 字段中的函数指针二选一，可以将其中一个设置为
NULL。
sa_sigaction：这也是一个函数指针，与 sa_handler 字段类似，用于处理信号。
与 sa_handler 不同的是，它接受三个参数：一个整数表示信号编号，一个
siginfo_t 类型的指针表示信号的附加信息，以及一个
void 类型的指针，指向当前的上下文。这个字段与
sa_handler 二选一，可以将其中一个设置为 NULL。
sa_mask：这是一个信号集（sigset_t），用于指定在处理当前信号时要屏蔽的信号集。
当处理当前信号时，如果其他被屏蔽的信号发生了，它们将被挂起并等待当前信号处理完毕后再被处理。
通常情况下，我们会将当前信号自身添加到 sa_mask 中，以避免在处理当前信号时再次触发同样的信号。
sa_flags：这是一个整数，用于指定信号处理的一些标志。常见的标志包括：
SA_RESTART：当信号处理函数返回时，系统调用会自动重启，而不是中断。这对于某些系统调用非常重要，以
SA_NOCLDSTOP：如果进程的子进程停止（但不终止），则不会向父进程发送 SIGCHLD 信号。
SA_SIGINFO：如果设置了这个标志并使用了 sa_sigaction 函数指针，将会调用 sa_sigaction 而不是 sa
```

2.3.下面正式进入多进程服务器的构建：

2.31多进程并发服务器模型图解：



2.32多进程服务器框架示例代码：

2.加入fork多进程与信号回收的逻辑之后的多进程并发服务器：

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>
#include <wait.h>
#include <stdlib.h>

//回收资源的函数:
void recyle_res()
{
    int pid = waitpid(-1,NULL,0);
    printf("回收了子进程%d的资源\n",pid);
}

int main(int argc, char const *argv[])
{
    // 1.创建流式监听套接字类型:
    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd == -1)
    {
        perror("socket() err:");
        return -1;
    }
    // 2.定义一个网络地址信息结构体:
    struct sockaddr_in serverInfo;
    memset(&serverInfo, 0, sizeof(serverInfo));
    serverInfo.sin_family = AF_INET;
    // 端口号都是2字节, short类型, 所以一定要注意字节序的问题:
    serverInfo.sin_port = htons(9999);
    serverInfo.sin_addr.s_addr = inet_addr("192.168.250.100");
    // 3.绑定网络地址信息结构体:
    int ret = bind(listen_fd, (const struct sockaddr *)&serverInfo, sizeof(serverInfo));
    if (ret == -1)
    {
        perror("bind err:");
        return -1;
    }
```

```
    // 4.设置监听的状态:
    ret = listen(listen_fd, 1);
    if (ret == -1)
    {
        perror("listen err:");
        return -1;
    }
    printf("多进程服务器启动\n");
    while (true)
    {
        int connect_fd = accept(listen_fd,NULL,NULL);
        if(connect_fd == -1)
        {
            perror("connect err:");
            return -1;
        }
        //创建子进程来处理链接套接字中的事务:
        int pid = fork();
        if(pid == 0)//子进程
        {
            char buf[128] = {0};
            while (true)
            {
                memset(buf,0,sizeof(buf));
                int nbytes = read(connect_fd,buf,sizeof(buf)-1);
                if(nbytes == -1)
                {
                    perror("read err:");
                    continue;
                }
                if(nbytes == 0)
                {
                    printf("对方断开链接\n");
                    close(connect_fd);
                    exit(0);//SIGCHILD
                }
                printf("客户端发来的数据: %s \n",buf);
                //回显示:
                nbytes = write(connect_fd,buf,strlen(buf));
                if(nbytes == -1)
                {
                    perror("write err:");
                    continue;
                }
```

```
            }
        }
        else if(pid > 0)//父进程:
        {
            //认识信号注册函数sigaction:
            struct sigaction sa = {0};
            sa.sa_handler = recyle_res;
            sa.sa_flags = SA_RESTART;
            //异步信号处理子进程的资源回收的问题:
            sigaction(SIGCHLD,&sa,NULL);
            continue;
        }
        else{
            perror("fork err:");
            return -1;
        }
    }
    return 0;
}
```

我们可看一下最大可并发的数据，即文件描述符最大值：

ulimit -a 命令输出的常见资源限制信息：

> - core file size（核心转储文件大小限制）：用于限制 core dump（核心转储）文件的最大大小，单位为 blocks。如果该值为 0，则表示禁用核心转储文件。
> - data seg size（数据段大小限制）：用于限制进程数据段的最大大小，单位为 kbytes。
> - scheduling priority（调度优先级限制）：用于限制进程的调度优先级，取值范围为 -20 到 19。
> - file size（文件大小限制）：用于限制单个文件的最大大小，单位为 blocks。
> - open files（打开文件数限制）：用于限制进程能够同时打开的文件描述符数量。
> - pipe size（管道大小限制）：用于限制单个管道的最大大小，单位为 bytes。
> - stack size（进程堆栈大小限制）：用于限制进程堆栈的最大大小，单位为 kbytes。
> - cpu time（CPU 时间限制）：用于限制进程占用 CPU 的最大时间，单位为 seconds。
> - max user processes（最大用户进程数限制）：用于限制用户能够同时运行的进程数量。

由ulimit -a可以看到 Linux创建的多进程并发服务器的理论上限为为15400个。实际上在4核16线程的i7处理器上，可能会最多3000个左右，就会有明显的卡顿现象。

至此，第一阶段的多进程服务模型构建完毕。

下一阶段：多线程服务器。

第三节、多线程并发服务器

3.1多线程前置知识回顾:

我们来了解一下，在Linux中一个进程可以创建子线程的数量可以通过以下路径的配置文件查看：

> $ cat /proc/sys/kernel/threads-max
> 也可以通过修改这个配置文件中的数量增加线程量。

理论上一个进程，可以创建很多线程但一个进程可以创建的最大线程数受到许多因素的影响，例如系统硬件资源、进程的配置、线程的堆栈大小、进程的虚拟内存限制等等。在实际应用中需要根据测试结果进行适当的调整，以避免出现性能瓶颈或者资源耗尽等问题。

因为：

> 在 Linux 中，每个线程都有一个独立的内核栈，用于存储线程的执行上下文、局部变量、函数调用栈等信息。每个线程的内核栈的大小是固定的，并且由系统内核在创建线程时分配和初始化。
> **每个线程都有两种类型的栈：用户栈和内核栈**。线程的用户栈用于存储线程的执行上下文、局部变量、函数调用栈等信息，是线程执行代码时的主要工作区域。线程的内核栈用于存储线程在内核中执行时所需要的状态信息，例如系统调用、中断处理等。内核栈和用户栈是两个独立的栈，分别用于不同的目的。
> 线程的内核栈和用户栈在概念上是不同的，但在实际实现中，它们可能使用同一块物理内存。例如，在 x86 架构上，线程的内核栈和用户栈都是在同一块物理内存区域中，但是它们使用不同的段寄存器来访问这个内存区域。
> 需要注意的是，线程的内核栈和用户栈的大小是可以独立设置的。在大多数情况下，线程的用户栈的大小由编译器和链接器决定，而线程的内核栈的大小可以通过调用 pthread_attr_setstacksize 函数来设置。线程的用户栈和内核栈在使用上有一些不同之处，但它们共同构成了线程的执行环境。
>
> 线程的内核栈的大小通常在创建线程时指定，可以使用 pthread_attr_setstacksize 函数来设置线程的内核栈大小。如果没有指定内核栈的大小，则使用系统默认值。在大多数 Linux 系统中，线程的内核栈的大小通常在 2MB 左右。
> 需要注意的是，线程的内核栈的大小是与线程的数量有关的。如果系统中创建了大量的线程，并且每个线程的内核栈大小都很大，那么可能会导致系统资源不足，从而影响系统的性能和稳定性。因此，在创建线程时，需要根据实际应用的需求和系统的硬件资源情况来设置线程的内核栈大小，以达到最佳的性能和可靠性。

1.pthread_create()创建线程:

pthread_create() 函数是用于创建线程的 POSIX 标准库函数。它的原型如下：

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

```

> 其中，**thread** 是一个指向线程标识符的指针，**attr** 是一个指向线程属性的指针，**start_routine** 是一个指向线程函数的指针，**arg** 是传递给线程函数的参数。
> 参数：
> 调用 **pthread_create()** 函数将创建一个新的线程，并使线程开始执行。新线程的 ID 将存储在 **thread** 参数指向的变量中。
> **attr** 参数可以用来指定线程的属性，如调度策略、栈大小等。如果不需要指定属性，可以将其设置为 NULL。
> **start_routine** 参数是指向线程函数的指针，该函数将在新线程中执行。
> **arg** 参数是传递给线程函数的参数。
> 返回值：
> 成功返回0，不成功返回非0值，并置位错误码。

2.pthread_join(),主线程阻塞等待子线程的执行结束，并回收子线程的线程资源。

pthread_join() 是一个 POSIX 线程库函数，它用于等待一个线程结束。其函数原型如下：

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

> 参数：
> 其中，**thread** 参数是待等待的线程标识符，**retval** 是一个指向指针的指针，用于获取线程的返回值。如果线程没有返回值，可以将其设置为 NULL。
> 调用 **pthread_join()** 函数将阻塞当前线程，直到被等待的线程结束为止。如果被等待的线程已经结束，**pthread_join()** 函数将立即返回。如果等待的线程还没有结束，调用 **pthread_join()** 函数将会使当前线程进入阻塞状态，直到被等待的线程结束为止。
> 当被等待的线程结束时，它的返回值将被存储在 **retval** 指向的内存中。如果不需要获取线程的返回值，可以将 **retval** 参数设置为 NULL
> **pthread_detach()，主线程与子线程分离执行各自的逻辑，主线程不再阻塞，子线程将自动回收其线程资源。**
> **返回值:**
> **在正常情况下，如果被等待的线程已经结束，pthread_join() 函数应该总是能够成功等待，并返回 0。如果被等待的线程尚未结束，pthread_join() 函数将会阻塞当前线程，直到被等待的线程结束为止。**

3.pthread_detach() 是一个 POSIX 线程库函数，

它用于将一个线程标记为“可分离的”（detached）。**可分离的线程是一种特殊的线程，当它结束时，系统会自动回收它所占用的资源，而不需要其他线程调用 pthread_join() 函数来等待它结束并回收资源。**

其函数原型如下：

```
int pthread_detach(pthread_t thread);
```

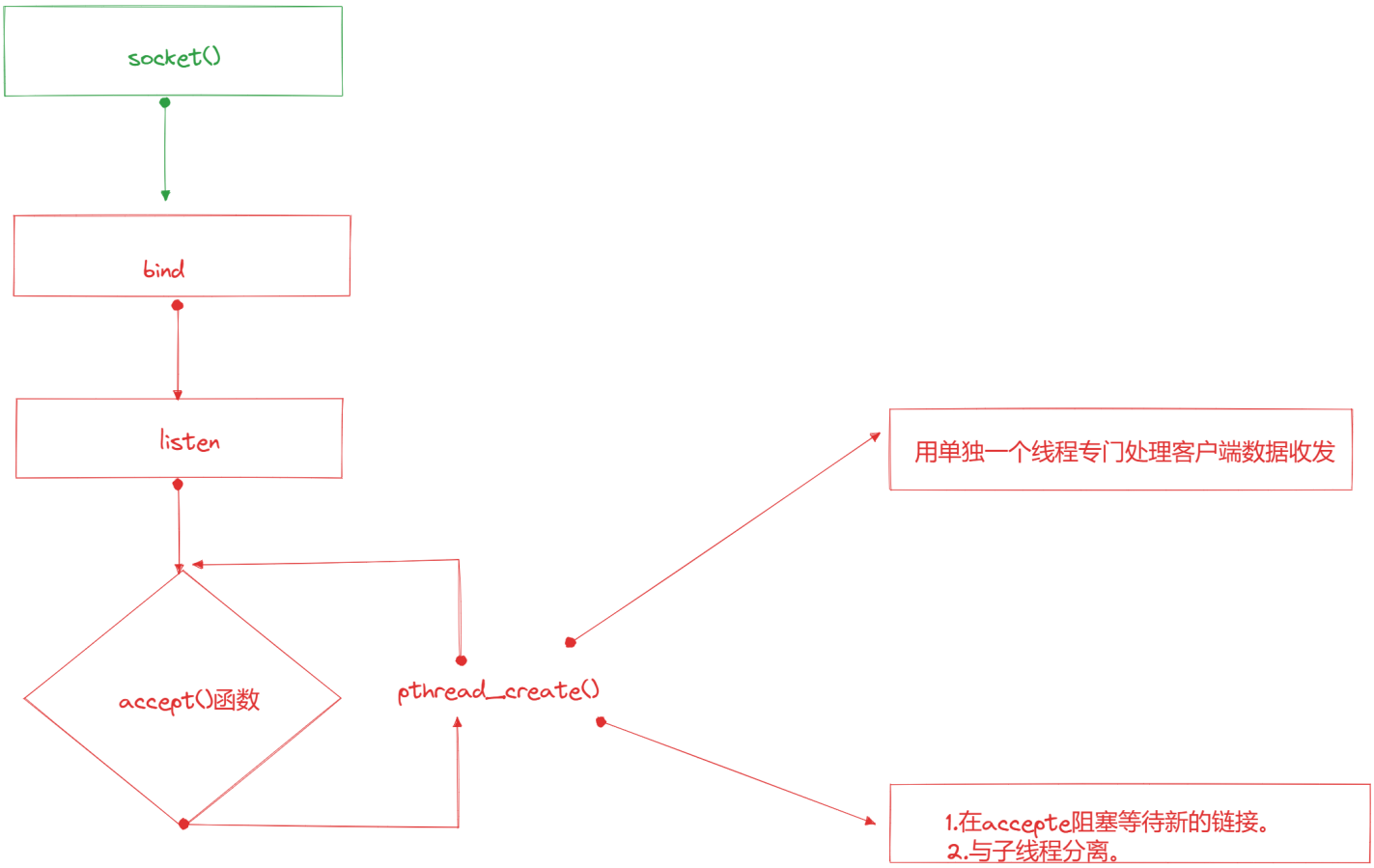
其中，**thread** 参数是要标记为可分离的线程的线程标识符。如果线程已经被标记为可分离的，则调用 pthread_detach() 函数将不会有任何效果。

调用 pthread_detach() 函数将会将指定的线程标记为可分离的。当这个线程结束时，它所占用的资源将会被自动回收。被标记为可分离的线程不需要等待其他线程调用 pthread_join() 函数来回收资源.

当然，多线程的内容涉及的还有很多，比如，竞态时，要设置同步互斥的机制。在网络编程中，就不深入讲解了。

那么下面我们来看一看，如何使用多线程，创建一个一对多的并发服务器模型：

3.2多线程并发服务器模型构建图解：



3.3加入多线程后，创建多线程并发服务器模型的代码示例：

```
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>
#include <stdlib.h>
#include <pthread.h>

//子线程执行的函数:
void * subThreadTask(void* arg)
{
    int connect_fd = *(int*)arg;
    char buf[128] = {0};
    while (true)
    {
        memset(buf,0,sizeof(buf));
        int nbytes = read(connect_fd,buf,sizeof(buf)-1);
```

```
        if(nbytes == -1)
        {
            continue;
        }
        if(nbytes == 0)
        {
            printf("对方已经关闭了\n");
            break;
        }
        //打印一下:
        printf("客户端发来的数据: %s \n",buf);
        //回显服务器:
        nbytes = write(connect_fd,buf,strlen(buf));
        if(nbytes == -1)
        {
            perror("write err:");
            break;
        }
    }
}
int main(int argc, char const *argv[])
{
    int listen_fd = socket(AF_INET,SOCK_STREAM,0);
    if(listen_fd == -1)
    {
        perror("socket err:");
        return -1;
    }
    //创建网络信息结构体:
    struct sockaddr_in serverInfo = {0};
    serverInfo.sin_family = AF_INET;
    serverInfo.sin_port = htons(8080);
    serverInfo.sin_addr.s_addr = INADDR_ANY;
    //bind:
    int ret = bind(listen_fd,(struct sockaddr*)&serverInfo,sizeof(serverInfo));
    if(ret == -1)
    {
        perror("bind err:");
        return -1;
    }

    //设置套接字为监听套接字:
    ret = listen(listen_fd,5);
    if(ret == -1)
```

```
    {
        perror("liseten err:");
        return -1;
    }
    printf("多线程并发服务器启动\n");
    while (true)
    {
        int connect_fd = accept(listen_fd,NULL,NULL);
        if(connect_fd == -1)
        {
            continue;
        }
        //1.创建子线程:
        pthread_t subThreadId;
        pthread_create(&subThreadId,NULL,subThreadTask,(void*)&connect_fd);
        //2.设定为分离态:
        pthread_detach(subThreadId);
    }
    return 0;
}
```

注意编译时 加上-lpthread库。

3.4总结：多进程并发、多线程并发都是实现并发编程的方式，但它们之间有以下区别：

1. **多进程并发：**每个进程都有自己独立的地址空间和系统资源，如果进程间通信需要通过操作系统提供的IPC（Inter-Process Communication，进程间通信）机制。**多进程并发的优点是稳定性高，一个进程崩溃不会影响其他进程的运行，但缺点是创建进程的开销较大，进程间通信复杂，效率较低。**
2. **多线程并发：**多个线程共享同一个进程地址空间和系统资源，线程间通信可以通过共享内存、信号量、互斥锁等机制实现。**多线程并发的优点是创建线程的开销较小，线程间通信较简单，效率较高，但缺点是稳定性较差，一个线程的崩溃可能会导致整个进程的崩溃。**

如果想避免多进程与多线程的缺点，又想兼容他们的优点的技术有吗？有：那就是IO多路复用技术。

IO多路复用并发：通过操作系统提供的IO多路复用机制（如select、poll、epoll等），可以在单线程内同时处理多个IO事件，从而实现高效的并发编程。相比多进程并发和多线程并发，IO多路复用并发的优点是开销较小、效率较高、稳定性较好，但缺点是编程复杂度较高，需要掌握较为深入的操作系统知识。

综上所述，多进程并发、多线程并发和IO多路复用并发各有优缺点，需要根据实际需求和编程场景选择合适的方式来实现并发编程。**下一节，我们将介绍IO多路复用技术。**