# 1 线程池

## 1.1 线程封装

对Linux下的有关线程的接口进行封装

与线程有关的还有互斥锁和条件变量

## 1.2 生产者-消费者问题

一个多线程同步问题的经典案例

该问题描述了两个共享固定大小缓冲区的线程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。

生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。

与此同时，消费者也在缓冲区消耗这些数据。

该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

## 1.3 线程池

面向对象编程中，创建和销毁对象是很费时的，因为创建一个对象要获取内存资源或者其它更多资源。

所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁。

基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

一个简单的线程池实现:

线程池组件：

线程池管理器(ThreadPool):用于创建并管理线程池

工作线程(WorkThread): 线程池中的线程

任务接口(Task):每个任务必须实现的接口，以供工作线程调度任务的执行。

任务队列(Queue):用于存放没有处理的任务。提供一种缓冲机

# 2 并发服务器架构简析

## 2.1 read/write和recv/send区别

int recv(int sockfd,void \*buf,int len,int flags)

int send(int sockfd,void \*buf,int len,int flags)

ssize\_t read(int fildes, void \*buf, size\_t nbyte);

ssize\_t write(int fildes, void \*buf, size\_t nbyte);

前面的三个参数和read,write相同，第四个参数能够是0或是以下的组合：

MSG\_DONTROUTE--不查找路由表

MSG\_OOB--接受或发送带外数据

MSG\_PEEK--查看数据,并不从系统缓冲区移走数据

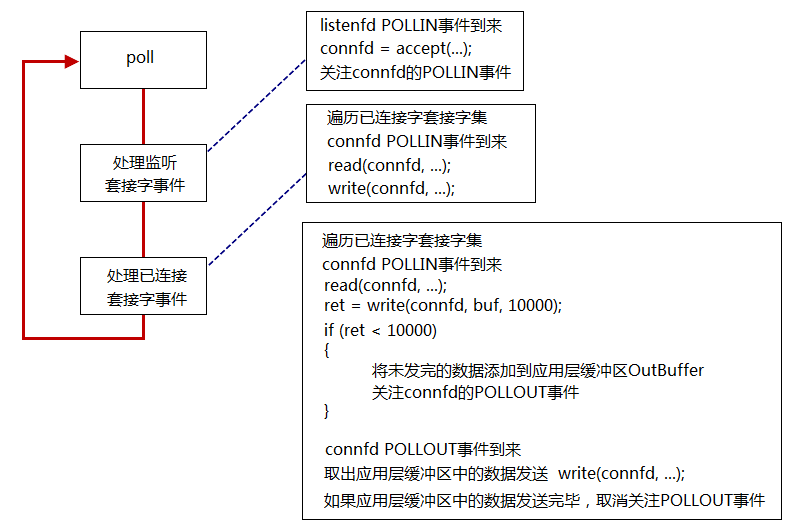
MSG\_WAITALL--等待任何数据

如果flags为0，则和read,write一样的操作

在linux中，所有的设备都可以看成是一个文件，所以可以用read/write来读写socket数据。

## 2.2 IO复用机制

### 2.2.1 poll



函数原型：

#include <poll.h>

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

struct pollfd

{

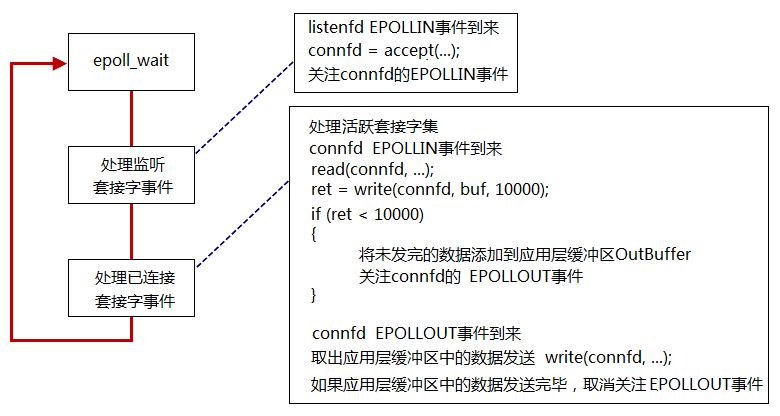
int fd; /\* file descriptor \*/

short events; /\* requested events \*/

short revents; /\* returned events \*/

};

### 2.2.2 epoll



函数原型：

#include <sys/epoll.h>

int epoll\_create(int size);

int epoll\_create1(int flags);

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout);

struct epoll\_event

{

uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

typedef union epoll\_data

{

void \*ptr;

int fd;

uint32\_t u32;

uint64\_t u64;

} epoll\_data\_t;

### 2.2.3 select/poll/epoll对比

|  |  |
| --- | --- |
| 原理 | |
| select | select本质上是通过设置或者检查存放fd标志位的数据结构来进行下一步处理。缺点是：  （1）单个进程可监视的fd数量被限制  （2）需要维护一个用来存放大量fd的数据结构，使得用户空间和内核空间在传递该结构时复制开销大  （3）对socket进行扫描时是线性扫描 |
| poll | poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。  它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：  大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。  poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。 |
| epoll | 在前面说到的复制问题上，epoll使用mmap减少复制开销。  还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知 |

|  |  |
| --- | --- |
| 一个进程所能打开的最大连接数 | |
| select | 单个进程所能打开的最大连接数有FD\_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32\*32，同理64位机器上FD\_SETSIZE为32\*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。 |
| poll | poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的 |
| epoll | 虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接，(cat /proc/sys/fs/file-max) 通过命令查看 |

## 2.3 网络IO模型

### 2.3.1 五种IO模型

IO模型分为同步IO（阻塞式IO、非阻塞式IO、IO复用(IO multiplexing)、信号驱动式IO）和异步IO(POSIX的aio\_系列函数)

### 2.3.2 同步异步与阻塞非阻塞的区别

#### 2.3.2.1 同步与异步

同步/异步, 它们是消息的通知机制

（1）同步

同步就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。

按照这个定义，其实绝大多数函数都是同步调用。但是一般而言，我们在说同步、异步的时候，特指那些需要其他部件协作或者需要一定时间完成的任务。

最常见的例子就是 SendMessage。该函数发送一个消息给某个窗口，在对方处理完消息之前，这个函数不返回。当对方处理完毕以后，该函数才把消息处理函数所返回的值返回给调用者。

（2）异步

异步的概念和同步相对。当一个异步过程调用发出后，调用者不会立刻得到结果。实际处理这个调用的部件是在调用发出后，通过状态、通知来通知调用者，或通过回调函数处理这个调用。

执行部件和调用者可以通过三种途径返回结果：状态、通知、回调函数。

如果执行部件用状态来通知，那么调用者就需要每隔一定时间检查一次，效率就很低，有些初学多线程编程的人，总喜欢用一个循环去检查某个变量的值，这其实是一种很严重的错误。

如果是使用通知的方式，效率则很高，因为执行部件几乎不需要做额外的操作。

至于回调函数，和通知没太多区别。

（3）举例说明

理解这两个概念，可以用去银行办理业务(可以取钱，也可以存钱)来比喻:

当到银行后,可以去ATM机前排队等候 -- (排队等候)就是同步等待消息

可以去大厅拿号,等到排到我的号时，柜台会通知轮到我办理业务. -- (等待别人通知)就是异步等待消息。

在异步消息通知机制中,等待消息者(在这个例子中就是等待办理业务的人)往往注册一个回调机制,在所等待的事件被触发时由触发机制(在这里是柜台的人)通过某种机制(在这里是写在小纸条上的号码)找到等待该事件的人。

在select/poll等IO复用机制中就是fd,当消息被触发时,触发机制通过fd找到处理该fd的处理函数。

（4）在实际的程序中,

同步消息通知机制：就好比简单的read/write 操作,它们需要等待这两个操作成功才能返回;

同步, 是由处理消息者自己去等待消息是否被触发;

异步消息通知机制：类似于select/poll 之类的多路复用IO 操作；

当所关注的消息被触发时,由消息触发机制通知触发对消息的处理。

异步, 由触发机制来通知处理消息者;

还是回到上面的例子：轮到你办理业务, 这个就是你关注的消息,

而办理什么业务, 就是对这个消息的处理,

两者是有区别的。而在真实的IO 操作时: 所关注的消息就是 该fd是否可读写,

而对消息的处理是 对这个fd 进行读写.

同步/异步仅仅关注的是如何通知消息,它们对如何处理消息并不关心,好比说,银行的人仅仅通知你轮到你办理业务了,而办理业务什么业务(存钱还是取钱)他们是不知道的。

#### 2.3.2.2 阻塞与非阻塞

阻塞/非阻塞, 它们是程序在等待消息(无所谓同步或者异步)时的状态.

（1）阻塞

阻塞调用是指调用结果返回之前，**当前线程会被挂起**。函数只有在得到结果之后才会返回。

有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。

socket接收数据函数recv是一个阻塞调用的例子。当socket工作在阻塞模式的时候，如果没有数据的情况下调用该函数，则当前线程就会被挂起，直到有数据为止。

（2）非阻塞

非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程**，而会立刻返回**。

（3）对象的阻塞模式和阻塞函数调用

对象是否处于阻塞模式和函数是不是阻塞调用有很强的相关性，但是并不是一一对应的。

阻塞对象上可以有非阻塞的调用方式，我们可以通过一定的API去轮询状态，在适当的时候调用阻塞函数，就可以避免阻塞。而对于非阻塞对象，调用特殊的函数也可以进入阻塞调用。函数select就是这样的一个例子。

（4）举例说明

继续上面的那个例子,

不论是排队等待，还是使用号码等待通知,如果在这个等待的过程中：

等待者除了等待消息之外不能做其它的事情,那么该机制就是阻塞的, 表现在程序中,也就是该程 序一直阻塞在该函数调用处不能继续往下执行。

相反,有的人喜欢在银行办理这些业务的时候一边打打电话发发短信一边等待,这样的状态就是非阻塞的, 因为他(等待者)没有阻塞在这个消息通知上,而是一边做自己的事情一边等待。

#### 2.3.2.3 易混淆的点

很多人也会把异步和非阻塞混淆,因为异步操作一般都不会在真正的IO 操作处被阻塞,比如如果用select 函数,当select 返回可读时再去read 一般都不会被阻塞。就好比当你的号码排到时一般都是在你之前已经没有人了,所以你再去柜台办理业务就不会被阻塞。可见,同步/异步与阻塞/非阻塞是两组不同的概念,它们可以共存组合,

而很多人之所以把同步和阻塞混淆,我想也是因为没有区分这两个概念,比如阻塞的read/write 操作中,其实是把消息通知和处理消息结合在了一起,在这里所关注的消息就是fd 是否可读/写,而处理消息则是对fd 读/写。当我们将这个fd 设置为非阻塞的时候,read/write 操作就不会在等待消息通知这里阻塞,如果fd 不可读/写则操作立即返回。.

#### 2.3.2.4 同步/异步与阻塞/非阻塞的组合分析

（1）同步阻塞形式:

效率是最低的，拿上面的例子来说，就是你专心排队，什么别的事都不做。

实际程序中就是未对fd 设置O\_NONBLOCK 标志位的read/write 操作,

（2）异步阻塞形式:

如果在银行等待办理业务的人采用的是异步的方式去等待消息被触发,也就是领了一张小纸条, 假如在这段时间里他不能离开银行做其它的事情,那么很显然,这个人被阻塞在了这个等待的操作上面; 异步操作是可以被阻塞住的,只不过它不是在处理消息时阻塞,而是在等待消息被触发时被阻塞。

比如select 函数, 假如传入的最后一个timeout 参数为NULL,那么如果所关注的事件没有一个被触发, 程序就会一直阻塞在这个select 调用处。

（3）同步非阻塞形式:

实际上是效率低下的,想象一下你一边打着电话一边还需要抬头看到底队伍排到你了没有,如果把打电话和观察排队的位置看成是程序的两个操作的话, 这个程序需要在这两种不同的行为之间来回的切换,效率可想而知是低下的;

很多人会写阻塞的read/write 操作,但是别忘了可以对fd 设置O\_NONBLOCK 标志位,这样就可以将同步操作变成非阻塞的了。

（4）异步非阻塞形式:

效率更高, 因为打电话是你(等待者)的事情,而通知你则是柜台(消息触发机制)的事情,程序没有在两种不同的操作中来回切换。

比如说，这个人突然发觉自己烟瘾犯了,需要出去抽根烟,

于是他告诉大堂经理说,排到我这个号码的时候麻烦到外面通知我一下(注册一个回调函数),那么他就没有被阻塞在这个等待的操作上面,自然这个就是异步+非阻塞的方式了。

如果使用异步非阻塞的情况,比如aio\_\*组的操作,当发起一个aio\_read 操作时,函数会马上返回不会被阻塞,当所关注的事件被触发时会调用之前注册的回调函数进行处理。

### 2.3.3 用户态与内核态

当一个任务（进程）执行系统调用而**陷入内核代码**中执行时，我们就称进程处于内核运行态（或简称为内核态）。此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。当进程在**执行用户自己的代码**时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。这与处于内核态的进程的状态有些类似。

内核态与用户态是操作系统的两种运行级别,跟intel cpu没有必然的联系, intel cpu提供Ring0-Ring3三种级别的运行模式，Ring0级别最高，Ring3最低。Linux使用了Ring3级别运行用户态，Ring0作为 内核态，没有使用Ring1和Ring2。Ring3状态不能访问Ring0的地址空间，包括代码和数据。Linux进程的4GB地址空间，3G-4G部分大家是共享的，是内核态的地址空间，这里存放在整个内核的代码和所有的内核模块，以及内核所维护的数据。用户运行一个程序，该程序所创建的进程开始是运行在用户态的，如果要执行文件操作，网络数据发送等操作，必须通过write，send等系统调用，这些系统调用会调用内核中的代码来完成操作，这时，必须切换到Ring0，然后进入3GB-4GB中的内核地址空间去执行这些代码完成操作，完成后，切换回Ring3，回到用户态。这样，用户态的程序就不能随意操作内核地址空间，具有一定的安全保护作用。至于说保护模式，是说通过内存页表操作等机制，保证进程间的地址空间不会互相冲突，一个进程的操作不会修改另一个进程的地址空间中的数据。

#### 2.3.3.1 用户态和内核态的概念区别

究竟什么是用户态，什么是内核态，这两个基本概念以前一直理解得不是很清楚，根本原因个人觉得是在于因为大部分时候我们在写程序时关注的重点和着眼的角度放在了实现的功能和代码的逻辑性上，先看一个例子：

（1）例子

void testfork()

{

if(0 = = fork())

{

printf(“create new process success!\n”);

}

printf(“testfork ok\n”);

}

这段代码很简单，从功能的角度来看，就是实际执行了一个fork()，生成一个新的进程，从逻辑的角度看，就是判断了如果fork()返回的是0则打印相关语句，然后函数最后再打印一句表示执行完整个testfork()函数。代码的执行逻辑和功能上看就是如此简单，从上到下一句一句执行而已，完全看不出来哪里有体现出用户态和进程态的概念。

如果说前面两种是静态观察的角度看的话，我们还可以从动态的角度来看这段代码，即它被转换成CPU执行的指令后加载执行的过程，这时这段程序就是一个动态执行的指令序列。而究竟加载了哪些代码，如何加载就是和操作系统密切相关了。

（2）特权级

熟悉Unix/Linux系统的人都知道，fork的工作实际上是以系统调用的方式完成相应功能的，具体的工作是由sys\_fork负责实施。其实无论是不是Unix或者Linux，对于任何操作系统来说，创建一个新的进程都是属于核心功能，因为它要做很多底层细致地工作，消耗系统的物理资源，比如分配物理内存，从父进程拷贝相关信息，拷贝设置页目录页表等等，这些显然不能随便让哪个程序就能去做，于是就自然引出特权级别的概念，显然，最关键性的权力必须由高特权级的程序来执行，这样才可以做到集中管理，减少有限资源的访问和使用冲突。

特权级显然是非常有效的管理和控制程序执行的手段，因此在硬件上对特权级做了很多支持，就Intel x86架构的CPU来说一共有0~3四个特权级，0级最高，3级最低，硬件上在执行每条指令时都会对指令所具有的特权级做相应的检查，相关的概念有CPL、DPL和RPL，这里不再过多阐述。硬件已经提供了一套特权级使用的相关机制，软件自然就是好好利用的问题，这属于操作系统要做的事情，对于Unix/Linux来说，只使用了0级特权级和3级特权级。也就是说在Unix/Linux系统中，一条工作在0级特权级的指令具有了CPU能提供的最高权力，而一条工作在3级特权级的指令具有CPU提供的最低或者说最基本权力。

（3）用户态和内核态

现在我们从特权级的调度来理解用户态和内核态就比较好理解了，当程序运行在3级特权级上时，就可以称之为运行在用户态，因为这是最低特权级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态；反之，当程序运行在0级特权级上时，就可以称之为运行在内核态。

虽然用户态下和内核态下工作的程序有很多差别，但最重要的差别就在于特权级的不同，即权力的不同。运行在用户态下的程序不能直接访问操作系统内核数据结构和程序，比如上面例子中的testfork()就不能直接调用sys\_fork()，因为前者是工作在用户态，属于用户态程序，而sys\_fork()是工作在内核态，属于内核态程序。

当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态，比如testfork()最初运行在用户态进程下，当它调用fork()最终触发sys\_fork()的执行时，就切换到了内核态。

#### 2.3.3.2 用户态和内核态的转换

（1）用户态切换到内核态的3种方式

a. **系统调用**

这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。

b. **异常**

当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

c. **外围设备的中断**

当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

这3种方式是系统在运行时由用户态转到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。

（2）具体的切换操作

从触发方式上看，可以认为存在前述3种不同的类型，但是从最终实际完成由用户态到内核态的切换操作上来说，涉及的关键步骤是完全一致的，没有任何区别，都相当于执行了一个中断响应的过程，因为系统调用实际上最终是中断机制实现的，而异常和中断的处理机制基本上也是一致的，关于它们的具体区别这里不再赘述。关于中断处理机制的细节和步骤这里也不做过多分析，涉及到由用户态切换到内核态的步骤主要包括：

首先，从当前进程的描述符中提取其内核栈的ss0及esp0信息。

其次，使用ss0和esp0指向的内核栈将当前进程的cs,eip,eflags,ss,esp信息保存起来，这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。

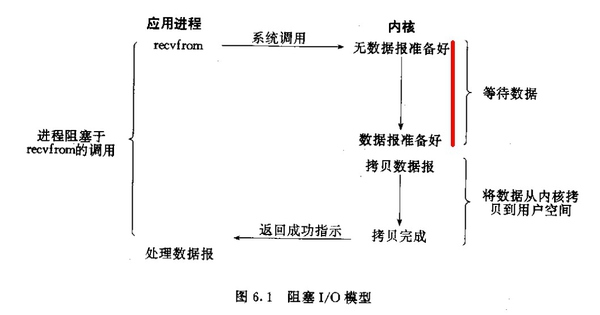
最后，将先前由中断向量检索得到的中断处理程序的cs,eip信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。

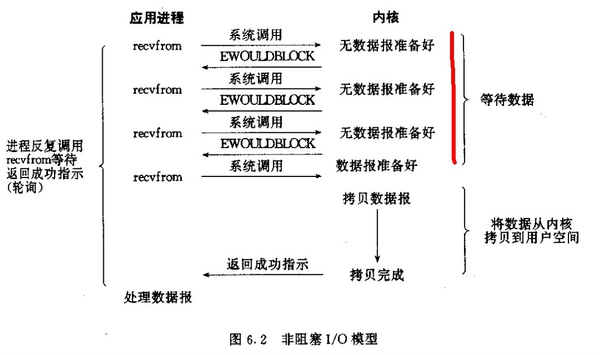
### 2.3.4 各模型的详解

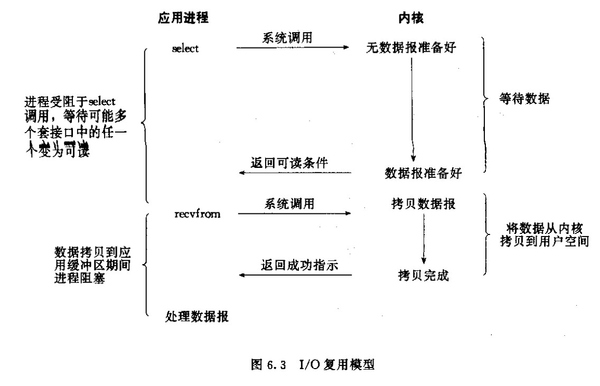
一个网络数据输入操作包括两个不同的阶段：

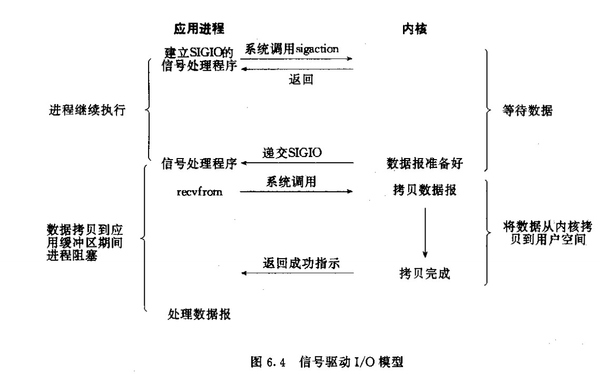
第一步，等待数据准备好

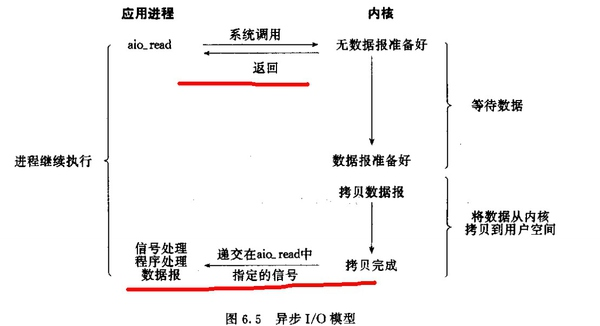
第二步，从内核向用户态复制数据

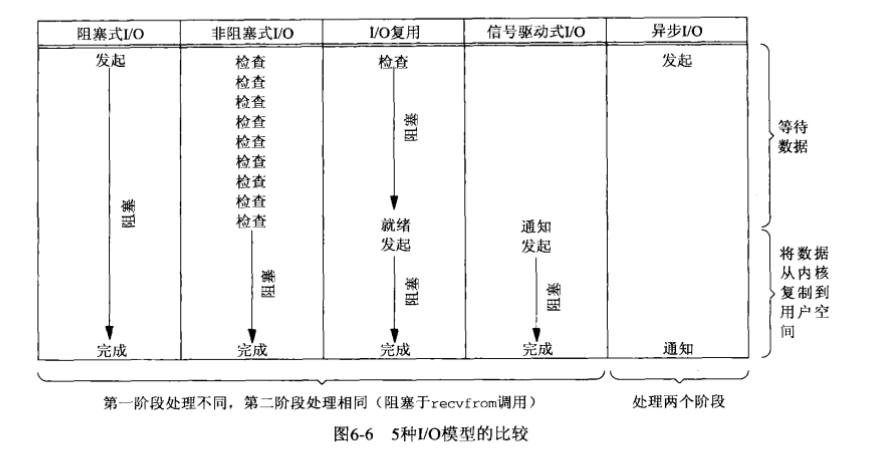












### 2.3.5 总结

阻塞/非阻塞：

关注的是用户态进程/线程(程序)的状态

其要访问的数据是否就绪，进程/线程是否需要等待；

同步/异步：

关注的是消息通信机制　　-- 内核与应用程序的交互

访问数据的方式，同步需要主动读写数据，在读写数据的过程中还是会阻塞；

异步只需要I/O操作完成的通知，并不主动读写数据，由操作系统内核完成数据的读写。

举例：风华去新华书店买《王道机试指南》

（1）如果新华书店没有，就一直等着书店有了书之后才走。 (同步阻塞)

（2）如果新华书店没有，先离开书店；然后每天都去书店逛一次，直到书店到货了，买了就走。(同步非阻塞)

（3）如果新华书店没有，留下电话号码；书店有货时，老板打电话通知他，他再去书店买书。(信号驱动IO 同步非阻塞)

（4）如果新华书店没有，留下地址；书店有货时，老板直接把书送到家(异步非阻塞)

对应于程序：用户进程调用系统调用，用户进程对应风华，内核对应书店老板，书对应数据， 买书就是一个系统调用，而内核拷贝数据到进程这个过程近似于老板送书到风华手中。

## 2.4 并发服务器

### 2.4.1 三点基础知识

（1）一个主机的端口号为所有进程所共享，但普通用户进程绑定bind不了一些特殊端口号如20、80等。多个进程不能同时监听listen同一个端口，会失败。

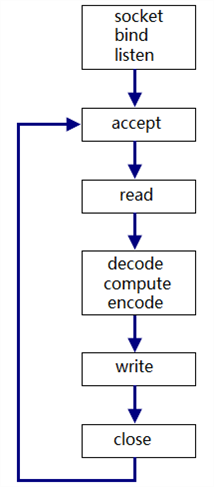
（2）每个进程都有自己的文件描述符（包括file fd, socket fd, timer fd, event fd, signal fd），一般是1024，可以通过ulimit -n 设置，但所有进程打开的文件描述符总数有上限，跟主机的内存有关。

（3）一个进程内的所有线程共享进程的文件描述符。

### 2.4.2 常见并发服务器方案

**（1）循环式/迭代式( iterative )服务器**

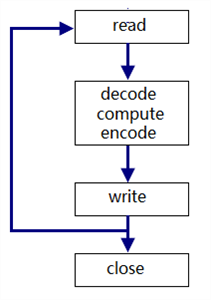
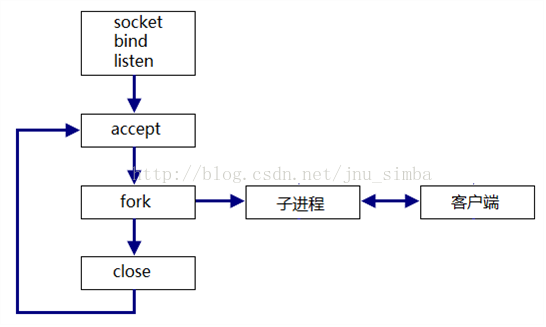
无法充分利用多核CPU，不适合执行时间较长的服务，即适用于短连接。如果是长连接则需要在read/write之间循环，那么只能服务一个客户端。



**（2）并发式(concurrent)服务器**

one connection per process/one connection per thread

适合执行时间比较长的服务

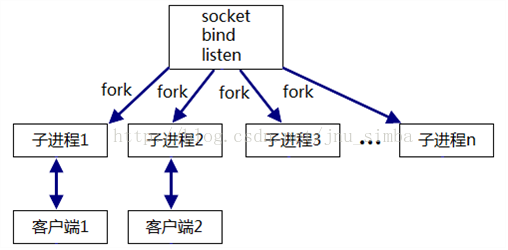


one connection per process : 主进程每次fork 之后要关闭connfd，子进程要关闭listenfd

one connection per thread : 主线程每次accept 回来就创建一个子线程服务，由于线程共享文件描述符，故不用关闭。

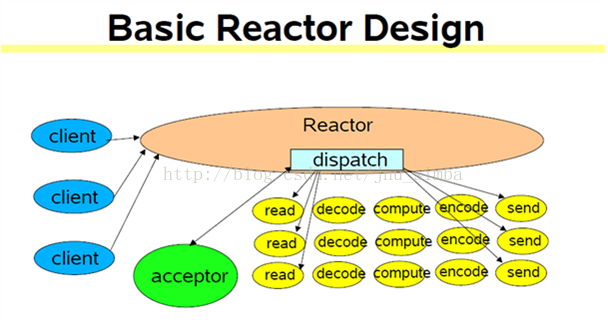
**（3）prefork or pre threaded（UNP2e 第27章）**

容易发生“惊群”现象，即多个子进程都处于accept状态。



**（4）反应式( reactive )服务器 (reactor模式)（select/poll/epoll）**

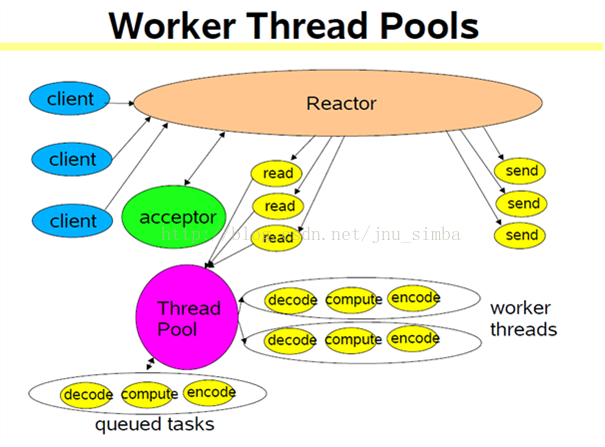
并发处理多个请求，实际上是在一个线程中完成。无法充分利用多核CPU，不适合执行时间比较长的服务，所以为了让客户感觉是在“并发”处理而不是“循环”处理，每个请求必须在相对较短时间内执行。



（5）reactor + thread per request（过渡方案）

（6）reactor + worker thread（过渡方案）

**（7）reactor + thread pool（能适应密集计算）**



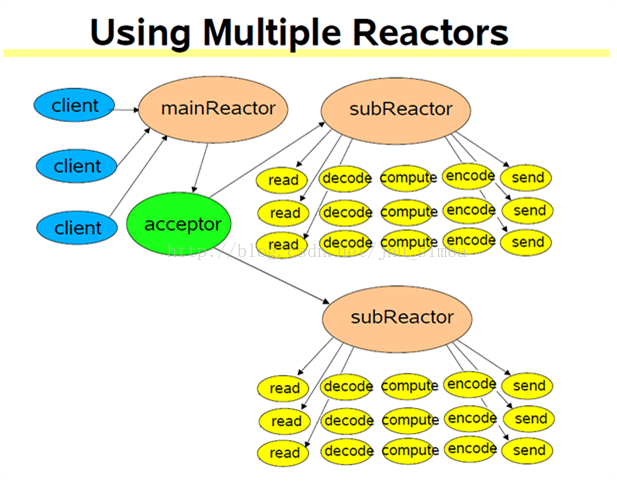
muduo库中的/example/suduku/ 中有这样一个例子，因为数独求解是计算密集型任务。

（**8）multiple reactors（能适应更大的突发I/O）**

reactors in threads（one loop per thread）

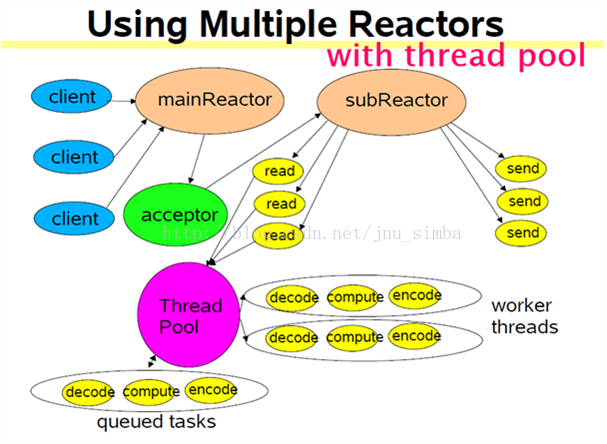
reactors in processes

一般来说一个subReactor适用于一个千兆网口



**（9）multiple reactors + thread pool**（one loop per thread + threadpool）（突发I/O与密集计算）

subReactor可以有多个，但threadpool只有一个。



**10、proactor服务器(proactor模式，基于异步I/O)**

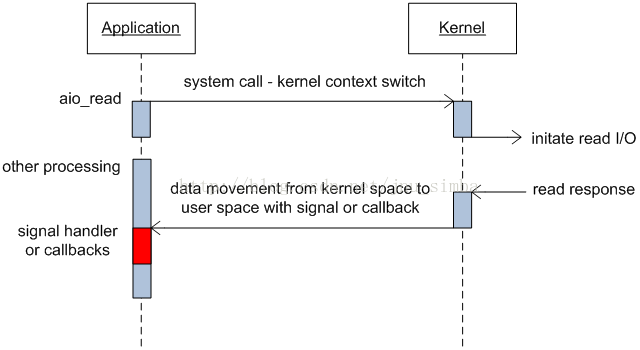
理论上proactor比reactor效率要高一些，异步I/O能够让I/O操作与计算重叠。充分利用DMA特性。

Linux异步IO：

（1）glibc aio（aio\_\*），有bug。

（2）kernel native aio（io\_\*），也不完美。目前仅支持 O\_DIRECT 方式来对磁盘读写，跳过系统缓存。要自已实现缓存，难度不小。

boost asio实现的proactor，实际上不是真正意义上的异步I/O，底层是用epoll来实现的，模拟异步I/O的。



### 2.4.3 常见并发服务器方案比较



### 2.4.4 一些常见问题

**（1）Linux能同时启动多少个线程？**

对于 32-bit Linux，一个进程的地址空间是 4G，其中用户态能访问 3G 左右，而一个线程的默认栈 (stack) 大小是 8M，心算可知，一个进程大约最多能同时启动 350 个线程左右。

**（2）多线程能提高并发度吗？**

如果指的是“并发连接数”，不能。

假如单纯采用 thread per connection 的模型，那么并发连接数大约350，这远远低于基于事件的单线程程序所能轻松达到的并发连接数（几千上万，甚至几万）。所谓“基于事件”，指的是用 IO multiplexing event loop 的编程模型，又称 Reactor 模式。

**（3）多线程能提高吞吐量吗**？

对于计算密集型服务，不能。

如果要在一个8核的机器上压缩100个1G的文本文件，每个core的处理能力为200MB/s，那么“每次起8个进程，一个进程压缩一个文件”与“只启动一个进程（8个线程并发压缩一个文件）”，这两种方式总耗时相当，但是第二种方式能较快的拿到第一个压缩完的文件。

**（4）多线程能提高响应时间吗？**

可以。

**（5）多线程程序日志库要求**

线程安全。

第一种，多个线程并发写日志，两个线程的日志消息不会出现交织。用一个全局的mutex保护IO。但会造成全部线程抢占一个锁（串行写入）

第二种，每个线程单独写一个日志文件。有可能让业务线程阻塞在写磁盘操作上。（磁盘IO时间比较长）。

解决办法：用一个logging线程负责收集日志消息，并写入日志文件，其他业务线程只管往这个“日志线程”发送日志消息（如通过BlockingQueue提供接口），这称为“异步日志”，也是一个经典的生产者消费者模型。

**（6）线程池大小的选择**

如果池中执行任务时，密集计算所占时间比重为P（0<P<=1)，而系统一共有C个CPU，为了让C个CPU跑满而不过载，线程池大小的经验公式T=C/P，即T\*P=C（让CPU刚好跑满 ）

假设C=8，P=1.0，线程池的任务完全密集计算，只要8个活动线程就能让CPU饱和

假设C=8，P=0.5，线程池的任务有一半是计算，一半是IO，那么T=16，也就是16个“50%繁忙的线程”能让8个CPU忙个不停。

**（7）线程分类**

I/O线程（这里特指网络I/O）

计算线程

第三方库所用线程，如logging,又比如database

### 2.4.5 Reactor模式

Reactor模式，又叫反应器模式，应用于同步I/O的场景。以读操作为例来看看Reactor中的具体步骤：

读取操作：

（1）应用程序注册读就绪事件和相关联的事件处理器；

（2）事件分离器等待事件的发生；

（3）当发生读就绪事件的时候，事件分离器调用第一步注册的事件处理器；

（4）事件处理器首先执行实际的读取操作，然后根据读取到的内容进行进一步的处理。

### 2.4.6 Proactor模式

Proactor(主动器)模式中读取操作过程：

读取操作：

（1） 应用程序初始化一个异步读取操作，然后注册相应的事件处理器，此时事件处理器不关注读取就绪事件，而是关注读取完成事件，这是区别于Reactor的关键。

（2）事件分离器等待读取操作完成事件

（3）在事件分离器等待读取操作完成的时候，操作系统调用内核线程完成读取操作，并将读取的内容放入用户传递过来的缓存区中。这也是区别于Reactor的一点，Proactor中，应用程序需要传递缓存区。

（4）事件分离器捕获到读取完成事件后，激活应用程序注册的事件处理器，事件处理器直接从缓存区读取数据，而不需要进行实际的读取操作。

### 2.4.7 Reactor vs Proactor

Proactor 感兴趣的事件是读写完成事件。

Reactor和Proactor模式的主要区别：真正的读取和写入操作是由谁来完成的：

Reactor中需要应用(用户)程序自己读取或者写入数据

Proactor模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，内核会读取缓存区或者写入缓存区到真正的IO设备。

### 2.4.8 Reactor模式封装

InetAddress类

Socket类

SocketIO类

Acceptor类

TcpConnection类

EpollPoller类

TcpServer类

# 3 timerfd + eventfd 封装

## 3.1 timerfd

timerfd是Linux提供的一个定时器接口。这个接口基于文件描述符，通过文件描述符的可读事件进行超时通知，所以能够被用于select/poll/epoll的应用场景。timerfd是linux内核2.6.25版本中加入的接口。

#include <sys/timerfd.h>

int timerfd\_create(int clockid, int flags);

功能：该函数生成一个定时器对象，返回与之关联的文件描述符。

参数clockid:可设置为

CLOCK\_REALTIME：相对时间，从1970.1.1到目前的时间。更改系统时间会更改获取的值，它以系统时间为坐标。

CLOCK\_MONOTONIC：绝对时间，获取的时间为系统重启到现在的时间，更改系统时间对齐没有影响。

参数flags: 可设置为

TFD\_NONBLOCK（非阻塞），

TFD\_CLOEXEC（同O\_CLOEXEC）

#include <sys/timerfd.h>

int timerfd\_settime(int fd, int flags, const struct itimerspec \*new\_value, struct itimerspec \*old\_value);

功能：该函数能够启动和停止定时器

参数fd: timerfd对应的文件描述符

参数flags:

0表示是相对定时器

TFD\_TIMER\_ABSTIME表示是绝对定时器

参数new\_value:设置超时时间，如果为0则表示停止定时器。

参数old\_value:一般设为NULL, 不为NULL,则返回定时器这次设置之前的超时时间。

struct timespec

{

time\_t tv\_sec; /\*秒\*/

long tv\_nsec /\*纳秒\*/

}

struct itimerspec

{

struct timespec it\_interval; /\*interval for periodic timer\*/

struct timespec it\_value; /\*initial expiration\*/

}

操作：

read：读取缓冲区中的数据，其占据的存储空间为sizeof(uint\_64)，表示超时次数。

select/poll/epoll：当定时器超时时，会触发定时器相对应的文件描述符上的读操作，IO复用操作会返回，然后再去对该读事件进行处理。

## 3.2 eventfd

从Linux 2.6.27版本开始，新增了不少系统调用，其中包括eventfd，它的主要是用于进程或者线程间通信(如通知/等待机制的实现)。

函数原型：

#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);

返回值：函数返回一个文件描述符，与打开的其他文件一样，可以进行读写操作。

参数flags支持以下标志位：

EFD\_NONBLOCK类似于使用O\_NONBLOCK标志设置文件描述符。

EFD\_CLOEXEC 类似open以O\_CLOEXEC标志打开，O\_CLOEXEC应该表示执行exec()时，之前通过open()打开的文件描述符会自动关闭。

如果是2.6.26或之后版本的内核，flags 必须设置为0。

参数initval：初始化计数器值，该值保存在内核。

操作：

read：如果计数器A的值不为0时，读取成功，获得到该值。如果A的值为0，非阻塞模式时，会直接返回失败，并把error置为EINVAL;如果为阻塞模式，一直会阻塞到A为非0为止。

write：将缓冲区写入的8字节整形值加到内核计数器上，即会增加8字节的整数在计数器A上，如果其值达到0xfffffffffffffffe时，就会阻塞（在阻塞模式下），直到A的值被read。