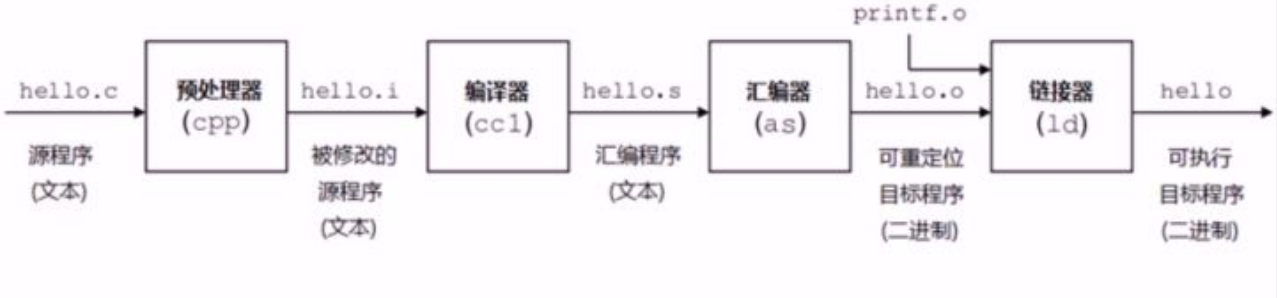
# 什么是gcc

**gcc的全称是GNU Compiler Collection，它是一个能够编译多种语言的编译器。**

最开始gcc是作为C语言的编译器（GNU C Compiler），现在除了c语言，还支持C++、java、Pascal等语言。

# gcc工作流程

- 预处理（--E）
    - 宏替换
    - 头文件展开
    - 去掉注释
    - .c文件变成了.i文件（本质上还是.c文件，只不过#include中的程序给链接进去）
- 编译（--S）
    - gcc调用不同语言的编译器
    - .i文件编程.s（汇编文件）
    - 生成汇编文件
- 汇编（-c）
    - .s文件转化成.o文件
    - 翻译成机器语言指令
    - 二进制文件
- 链接
    - .o文件变成可执行文件，一般不加后缀



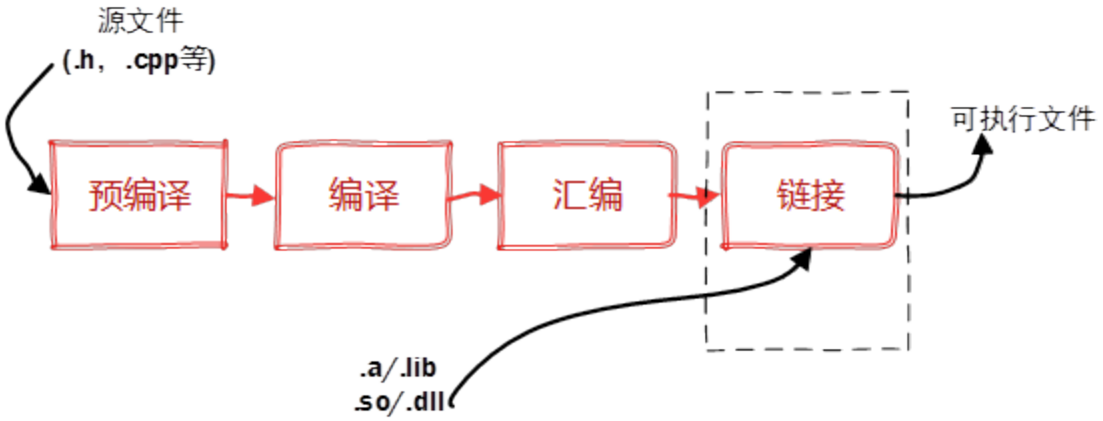

**预处理**实际上是将头文件、宏进行展开。

> **编译阶段**gcc调用不同语言的编译器。gcc实际上是个工具链，在编译程序的过程中调用不同的工具。
>
> **汇编阶段**gcc调用汇编器进行汇编。汇编语言是一种低级语言，在不同的设备中对应着不同的机器语言指令，一种汇编语言专用于某种计算机体系结构，可移植性比较差。通过相应的汇编程序可以将汇编语言转换成可执行的机器代码这一过程叫做汇编过程。汇编器生成的是可重定位的目标文件，在源程序中地址是从0开始的，这是一个相对地址，而程序真正在内存中运行时的地址肯定不是从0开始的，而且在编写源代码的时候也不能知道程序的绝对地址，所以**重定位**能够将源代码的代码、变量等定位为内存具体地址。
>
> **链接过程**会将程序所需要的目标文件进行链接成可执行文件。

# gcc常用参数

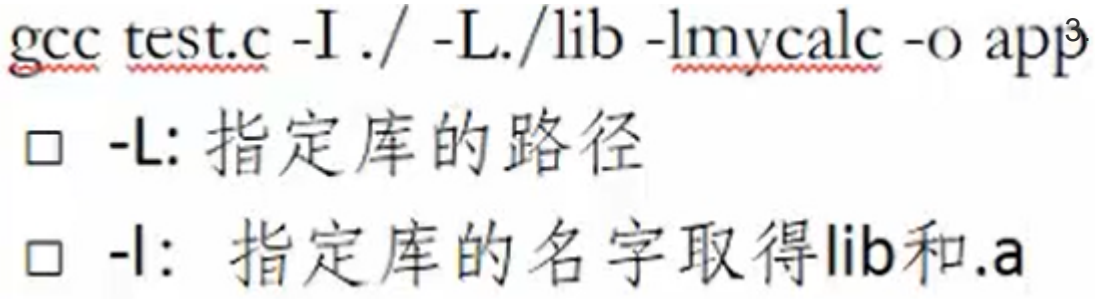
- -v/--version：查看gcc的版本
- -I：编译的时候指定头文件路径，不然头文件找不到
- -c：将汇编文件转换成二进制文件，得到.o文件
- -g：gdb调试的时候需要加
- -D：编译的时候指定一个宏（调试代码的时候需要使用例如printf函数，但是这种函数太多了对程序性能有影响，因此如果没有宏，则#ifdefine的内容不起作用）
- -wall：添加警告信息
- -On：-O是优化代码，n是优化级别：1，2，3

# 静态库和动态库




静态库、动态库区别来自【链接阶段】如何处理库，链接成可执行程序。分别称为静态链接方式、动态链接方式。

1. 什么是库？
    - 库是写好的现有的，成熟的，可以复用的代码。
    - 现实中每个程序都要依赖很多基础的底层库，不可能每个人的代码都从零开始，因此库的存在意义非同寻常。
    - 库是二进制文件，.o文件
    - 将源代码变成二进制的源代码
    - 主要起到加密的作用，为了防止泄露
2. 静态库的制作
    - 原材料：源代码（.c或.cpp文件）
    - 将.c文件生成.o文件（ex：g++ a.c -c）
    - 将.o文件打包
        - ar rcs 静态库名称 原材料(ex: ar rcs libtest.a a.0)
    - 态库的使用




3 动态库的制作

(so代表动态库)

- 命名规则：libxxx.so

- 制作步骤
    - 将源文件生产.o文件（gcc a.c -c -fpic）
    - 打包（gcc -shared a.o -o libxxx.so）
- 动态库的使用
    - 跟静态库一样
- 动态库无法加载的问题
    - 使用环境变量（临时设置和全局设置）

# gdb相关问题

- gdb 不能显示代码（No symbol table is loaded. Use the "file" command）
    - 要是用-g 比如： g++ map_test.cpp -g -o mao

# linux权限相关问题

对任意一个文件使用ls -l命令，如下图所示：

```
1 drwxr-xr-x   2 root root  4096 2009-01-14 17:34 bin
2 drwxr-xr-x   3 root root  4096 2009-01-14 14:36 boot
3 drwxr-xr-x  12 root root 14080 2009-07-20 14:13 dev
4 lrwxrwxrwx   1 root root    11 2009-01-14 10:05 cdrom -> media/cdrom
```

**任意取一行，如：drwxr-xr-x 2 root root 4096 2009-01-14 17:34 bin**

**用序列表示为：0123456789**

- 第一列
    - d：代表目录
    - -：代表文件
    - l：代表链接，如同windows的快捷方式
- 第一到九列
    - r：可读
    - w：可写
    - x：可执行文件
    - 0：代表文件类型

        - 123：表示文件所有者的权限
        - 456：表示同组用户的权限
        - 789：表示其他用户的权限
- 权限的数字表示
        - 读取的权限等于4，用r表示
        - 写入的权限等于2，用w表示
        - 执行的权限等于1，用x表示
- 改变文件权限命令

  > chmod 权限数字（如777） filename

- 改变目录下所有的文件的权限命令

  > chmod -R 权限数字（如777） 目录(如/home)

# 套接字类型

下面是创建套接字所用的socket所用的函数

```
#include<sys/socket.h>
int socket(int domain,int type,int protocol);
```

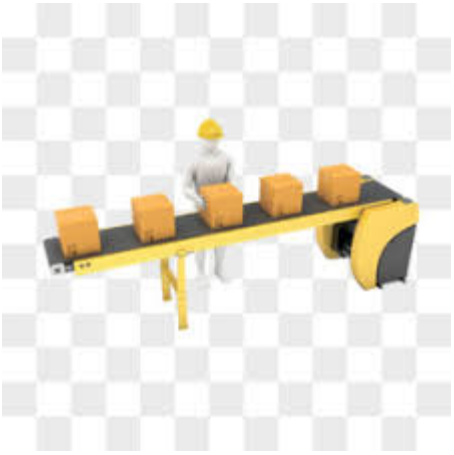
- **协议簇（Protocol Family）（int domain）**

  > 套接字有许多不同的通信协议分类，由socket函数第一个参数进行传递。其中最重要的是PF_INET（IPv4互联网协议族）

- **套接字类型（Type）（int type）**

  > socket函数的第二个参数决定套接字数据传输的方式。
  >
  > 协议族并不能决定数据传输方式，因为一个协议族也有很多数据传输方式

        - 面向连接的套接字（SOCK_STREAM）
          **面向连接的套接字类似于传送带**



有如下特点

a. 传输的过程中数据不会丢失
b. 按需传送数据
c. 传输数据不存在数据边界问题
d. 两端套接字必须一一对应

> 收发的套接字内部有缓冲，就是有字节数组。因此通过套接字传输的数据将保存在数组，因此数据可以填满缓冲一次被读取，也可以分段被读取，不存在数据边界问题。当缓冲区占满后，套接字无法接受数据，停止传输。所以不存在数据丢失问题

- 面向消息的套接字（SOCK_DGRAM）

  **面向消息的套接字类似于快递**

  有如下特点

  a. 强调快速传送而非顺序传送
  b. 传输的数据可能丢失也可能损坏
  c. 传输的数据有边界
  d. 限制每次传输的数据大小

- **协议信息（int protocol）**

> 由于socket函数前两个参数的存在，大部分情况可以向第三个参数传递0。但若同一个协议族中存在多个数据传输方式相同的协议，即数据传输方式相同，协议不同，需要第三个参数制定具体协议

```
//IPv4中面向连接的套接字
int tcp_socket=(PF_INET,SOCK_STREAM,IPPROTO_TCP)
//IPv4中面向消息的套接字
int tcp_socket=(PF_INET,SOCK_STREAM,IPPROTO_UDP)
```

# 实现基于TCP/IP的客户端服务端

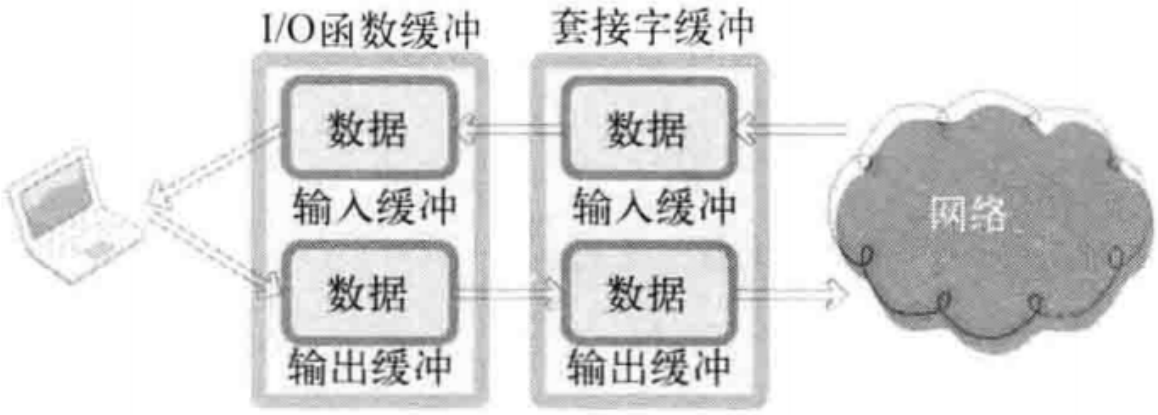
- **TCP服务端默认函数调用顺序**
    i. socket() 创建套接字
    ii. bind() 分配套接字地址
    iii. listen() 等待连接请求状态
    iv. accpet() 允许连接
    v. read()/write() 交换数据
    vi. close() 断开连接
- **TCP客户端默认函数调用顺序**
    i. socket() 创建套接字
    ii. connect() 请求连接
    iii. read()/write() 交换数据
    iv. close() 断开连接

    > 实现服务器端重要/必经过程就是给套接字分配IP和端口号（bind），但是客户端实现过程并未出现套接字的分配，而是创建套接字后立即调用了connect函数，为什么？
    >
    > 答：客户端其实是分配了IP和端口号的。在调用connect的时候分配的（何时），在操作系统内核中进行分配（何地），IP用的是主机的IP，端口号随机分配（何种方式）

- **注意事项**
    - 服务器端创建套接字后连续调用bind、listen函数进入等待状态，客户端通过connect函数发起连接请求
    - 客户端只能等到服务端调用listen后才能调用connect函数
    - 客户端调用connect函数前，服务器可能率先调用accept函数，然后服务端进入阻塞状态，直到客户端调用connect为止
- **TCP套接字中的I/O缓冲**

    > write函数调用后并不是立刻传送数据，read函数调用后也不是马上接收数据。而是将这些数据移到输入和输出缓冲中

    如下图所示：




I/O缓冲有以下特点：

i. I/O缓冲在每个TCP套接字中单独存在
ii. I/O缓冲在创建套接字时自动生成
iii. 即使关闭套接字也会继续传递输出缓冲中遗留的数据
iv. 关闭套接字将丢失输入缓冲中的数据

- **套接字的断开**

> close()函数表示完全断开套接字链接，并且不能收发任何数据。很显然这样做是不好的，若A主机断开连接后，完全无法调用接收数据和发送数据相关函数，这会导致B向A发送数据，A必须接受的数据也被销毁

套接字中会生成两个I/O流，如下图：

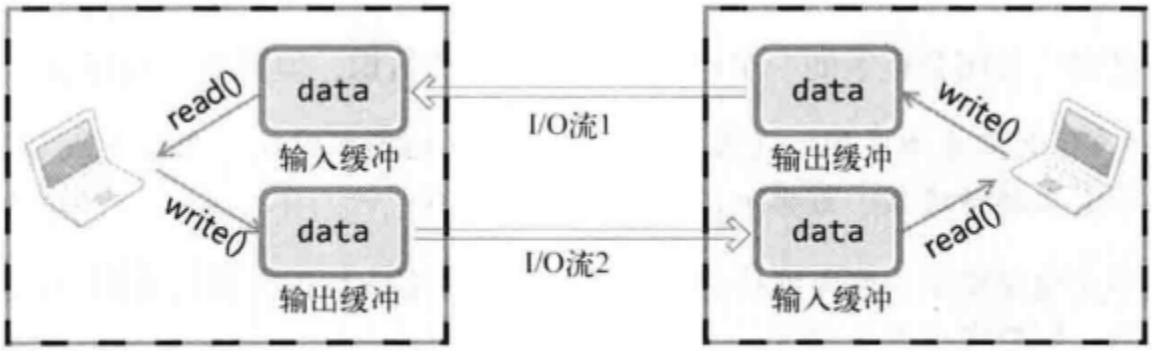



图7-2　套接字中生成的两个流

一旦两台主机间建立了套接字链接，每个主机就会拥有单独的输入流和输出流

# Linux操作系统中断

- **中断概念**

> 中断是指在CPU正常运行期间，由于内外部事件或由程序预先安排的事件引起的CPU暂时停止正在运行的程序，转而为该内部或外部事件或预先安排的事件服务的程序中去，服务完毕后再返回去继续运行被暂时中断的程序。这样的中断机制极大的提高了CPU运行效率。

根据CSAPP中所描述，中断是异常的一种类别。（p504）什么是异常？比如说在处理器中，依次执行对应的指令流程，这样的控制转移序列叫做控制流。但是系统会出现一些变化，系统必须对这种变化做出反应，而且这种变化是随机的，也不一定跟执行当前的程序关联，比若说子进程终止时父进程必须得到通知，硬盘定时器定期产生一个信号。这种叫做**控制流发生突变**，这些突变成为**异常控制流**。计算机的各个层次都会发生异常。(p502)
异常可以分为：中断、陷阱、故障、终止

| 类别 | 原因 | 同步/异步 | 返回行为 |
| --- | --- | --- | --- |
| 中断 | 来自I/O设备的信号 | 异步 | 总是返回到下一条指令 |
| 陷阱 | 有意的异常 | 同步 | 总是返回到下一条指令 |
| 故障 | 潜在可恢复的错误 | 同步 | 可能返回到当前指令 |
| 终止 | 不可恢复的错误 | 同步 | 不会返回 |

- **硬中断**
  硬中断是我们通常所说的中断的概念。硬中断是由硬件产生的，比如，像磁盘，网卡，键盘，时钟等。每个设备或设备集都有它自己的IRQ（中断请求）。内核中维护了一个IDT（中断描述符表），表中是中断处理函数的地址和一些其他的控制位。0-31号中断号位系统为预定义的中断和异常保留的，用户不得使用，所以硬件中断号从32开始分发。每当CPU接收到一个中断或者异常信号，CPU首先要做的决定是否响应这个中断（具体由中断控制器根据中断优先级决定是否给CPU发送中断信号），如果决定响应，就终止当前运行进程的运行，根据IDTR寄存器获取中断描述符表基地址，然后根据中断号定位具体的中断描述符。
- **软中断**
  软中断是由当前正在运行的进程所产生的。软中断并不会直接中断CPU。这种中断是一种需要内核为正在运行的进程去做一些事情（通常为I/O）的请求。

> 中断在本质上是软件或者硬件发生了某种情形而通知处理器的行为，处理器进而停止正在运行的指令流，去转去执行预定义的中断处理程序。软件中断也就是通知内核的机制的泛化名称，目的是促使系统切换到内核态去执行异常处理

> 程序。这里的异常处理程序其实就是一种系统调用，软件中断可以当做一种异常。总之，软件中断是当前进程切换到系统调用的过程。

# 系统调用知识点

## 用户态和内核态

l链接

## 系统调用过程

用户空间的程序无法直接执行内核代码，它们不能直接调用内核空间中的函数，因为内核驻留在受保护的地址空间上。如果进程可以直接在内核的地址空间上读写的话，系统安全就会失去控制。所以，应用程序应该以某种方式通知系统，告诉内核自己需要执行一个系统调用，希望系统切换到内核态，这样内核就可以代表应用程序来执行该系统调用了。

通知内核的机制是靠软件中断实现的。首先，用户程序为系统调用设置参数。其中一个参数是系统调用编号。参数设置完成后，程序执行“系统调用”指令。这个指令会导致一个异常：产生一个事件，这个事件会致使处理器切换到内核态并跳转到一个新的地址，并开始执行那里的异常处理程序。此时的异常处理程序实际上就是系统调用处理程序。它与硬件体系结构紧密相关。

> **系统调用的过程：**首先将API函数参数压到栈上，然后将函数内调用系统调用的代码放入寄存器，通过陷入中断，进入内核将控制权交给操作系统，操作系统获得控制后，将系统调用代码拿出来，跟操作系统一直维护的一张系统调用表做比较，已找到该系统调用程序体的内存地址，接着访问该地址，执行系统调用。执行完毕后，返回用户程序

## 系统调用和函数调用区别

**库函数调用**

函数调用主要通过压栈出栈的操作，面向应用开发。库函数顾名思义是把函数放到库里。是把一些常用到的函数编完放到一个文件里，供别人用。别人用的时候把它所在的文件名用#include<>加到里面就可以了。一般是指编译器提供的可在c源程序中调用的函数。可分为两类，一类是c语言标准规定的库函数，一类是编译器特定的库函数。(由于版权原因，库函数的源代码一般是不

可见的，但在头文件中你可以看到它对外的接口)

**系统调用**

系统调用就是用户在程序中调用操作系统所提供的一个子功能，也就是系统API，系统调用可以被看做特殊的公共子程序。通俗的讲是操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务，比如用户可以通过文件系统相关的调用请求系统打开文件、关闭文件或读写文件，可以通过时钟相关的系统调用获得系统时间或设置定时器等。系统中的各种共享资源都由操作系统统一掌管，因此在用户程序中，凡是与资源有关的操作（如存储分配、进行I/O传输及管理文件等），都必须通过系统调用方式向操作系统提出服务请求，并由操作系统代为完成。

---

# 对事件的两种处理方式

## reactor

如果要让服务器服务多个客户端，那么最直接的方式就是为每一条连接创建线程。其实创建进程也是可以的，原理是一样的，进程和线程的区别在于线程比较轻量级些，线程的创建和线程间切换的成本要小些，为了描述简述，后面都以线程为例。处理完业务逻辑后，随着连接关闭后线程也同样要销毁了，但是这样不停地创建和销毁线程，不仅会带来性能开销，也会造成浪费资源，而且如果要连接几万条连接，创建几万个线程去应对也是不现实的。要这么解决这个问题呢？我们可以使用「资源复用」的方式。也就是不用再为每个连接创建线程，而是创建一个「线程池」，将连接分配给线程，然后一个线程可以处理多个连接的业务。不过，这样又引来一个新的问题，线程怎样才能高效地处理多个连接的业务？

当一个连接对应一个线程时，线程一般采用「read -> 业务处理 -> send」的处理流程，如果当前连接没有数据可读，那么线程会阻塞在 `read` 操作上（ socket 默认情况是阻塞 I/O），不过这种阻塞方式并不影响其他线程。但是引入了线程池，那么一个线程要处理多个连接的业务，线程在处理某个连接的 `read` 操作时，如果遇到没有数据可读，就会发生阻塞，那么线程就没办法继续处理其他连接的业务。要解决这一个问题，最简单的方式就是将 socket 改成非阻塞，然后线程不断地轮询调用 `read` 操作来判断是否有数据，这种方式虽然该能够解决阻塞的问题，但是解决的方式比较粗暴，因为轮询是要消耗 CPU 的，而且随着一个 线程处理的连接越多，轮询的效率就会越低。

有没有办法在只有当连接上有数据的时候，线程才去发起读请求呢？答案是有的，实现这一技术的就是 I/O 多路复用。线程池复用+IO复用就形成了reactor模式。

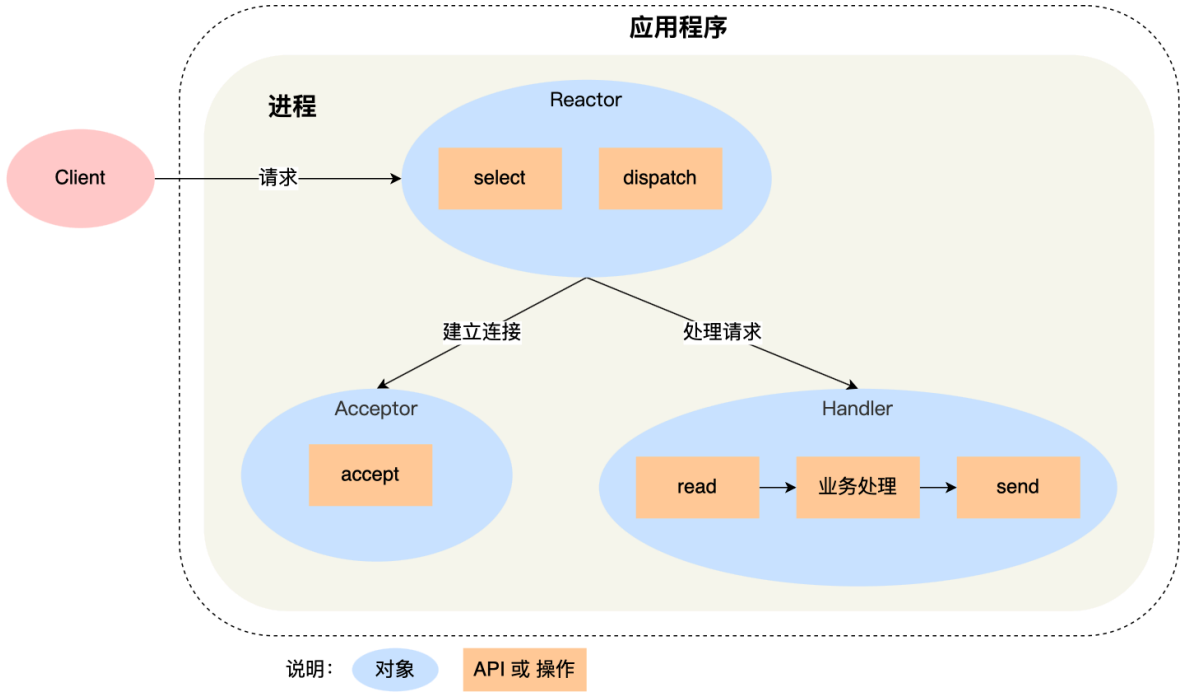
**Reactor的定义：**是一种接收多个输入事件的服务器事件驱动处理模式。服务器端通过IO多路复用来处理这些输入事件，并将这些事件同步分派给对应的处理线程。其实reactor模式也叫做Dispatcher 模式，本质上就是将收到的事件进行分发处理。Reactor模式中有两个关键的组成：①主反应堆reactor在一个单独的线程中运行，负责监听和分发事件，将接收到的事件分为监听socket和连接socket，连接socket放入任务队列让线程池线程去抢占式调度。②Handlers或Accepter，处理任务队列中具体的逻辑或者建立连接socket。

根据 Reactor 的数量和处理资源池线程的数量不同，有 3 种典型的实现：

1. 单 Reactor 单线程；
2. 单 Reactor 多线程；
3. 主从 Reactor 多线程。

接下来一一介绍：

- **单 Reactor 单线程**
  顾名思义，一个主反应堆reactor，一个accepter或者handler来处理接收的事件。




  **优点：**模型简单，没有多线程、进程通信、竞争的问题，全部都在一个线程中完成。
  **缺点：**性能问题，只有一个线程，无法完全发挥多核 CPU 的性能。Handler 在处理某个连接上的业务时，整个进程无法处理其他连接事件，很容易导致性能瓶

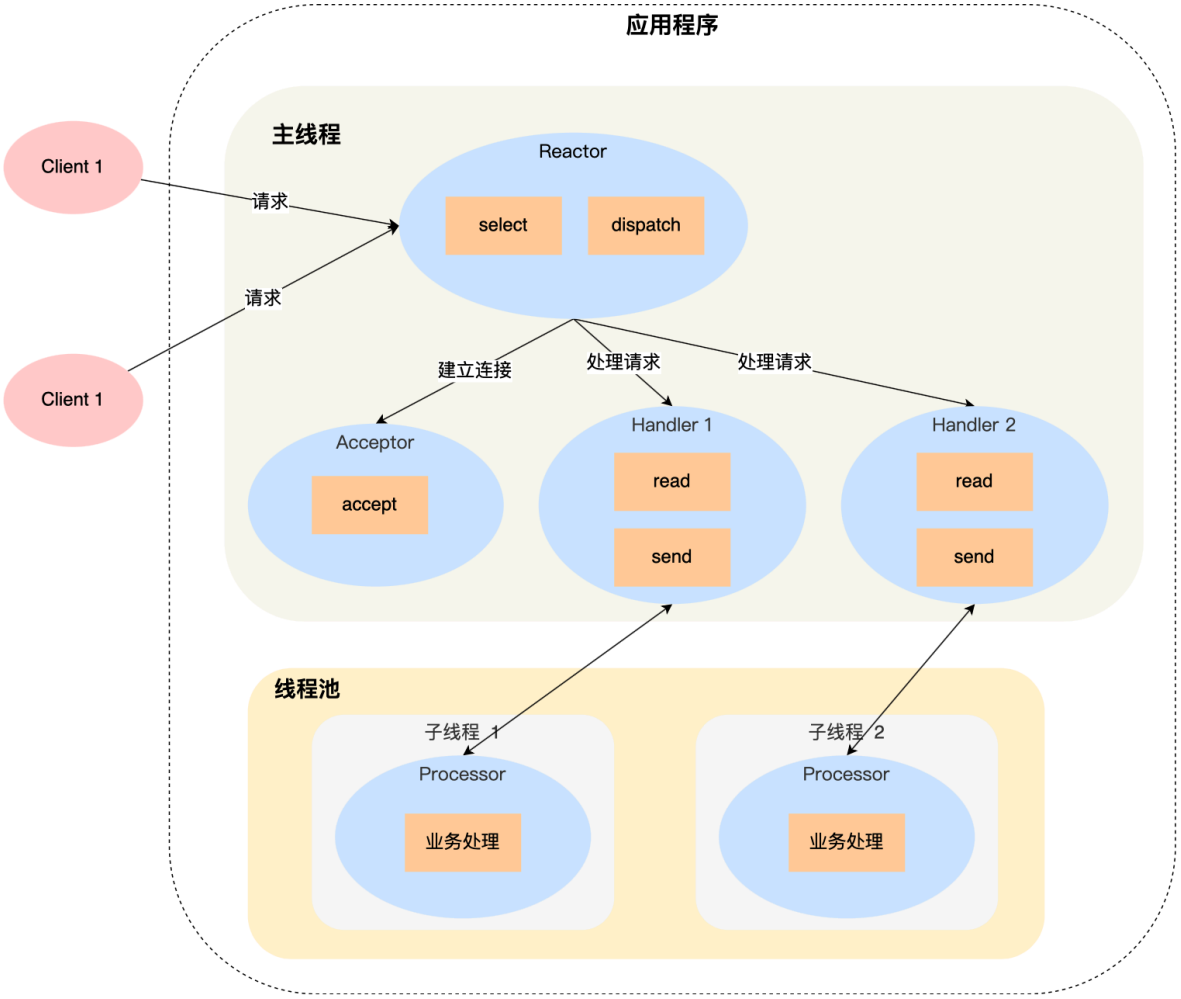
颈。

可靠性问题，线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障。

**使用场景：**客户端的数量有限，业务处理非常快速，比如 Redis，业务处理的时间复杂度 O(1)，因为 Redis 业务处理主要是在内存中完成，操作的速度是很快的，性能瓶颈不在 CPU 上，所以 Redis 对于命令的处理是单进程的方案。

- **单 Reactor 多线程**

  一个主反应堆reactor和一个线程池，线程池用来处理分发的事件

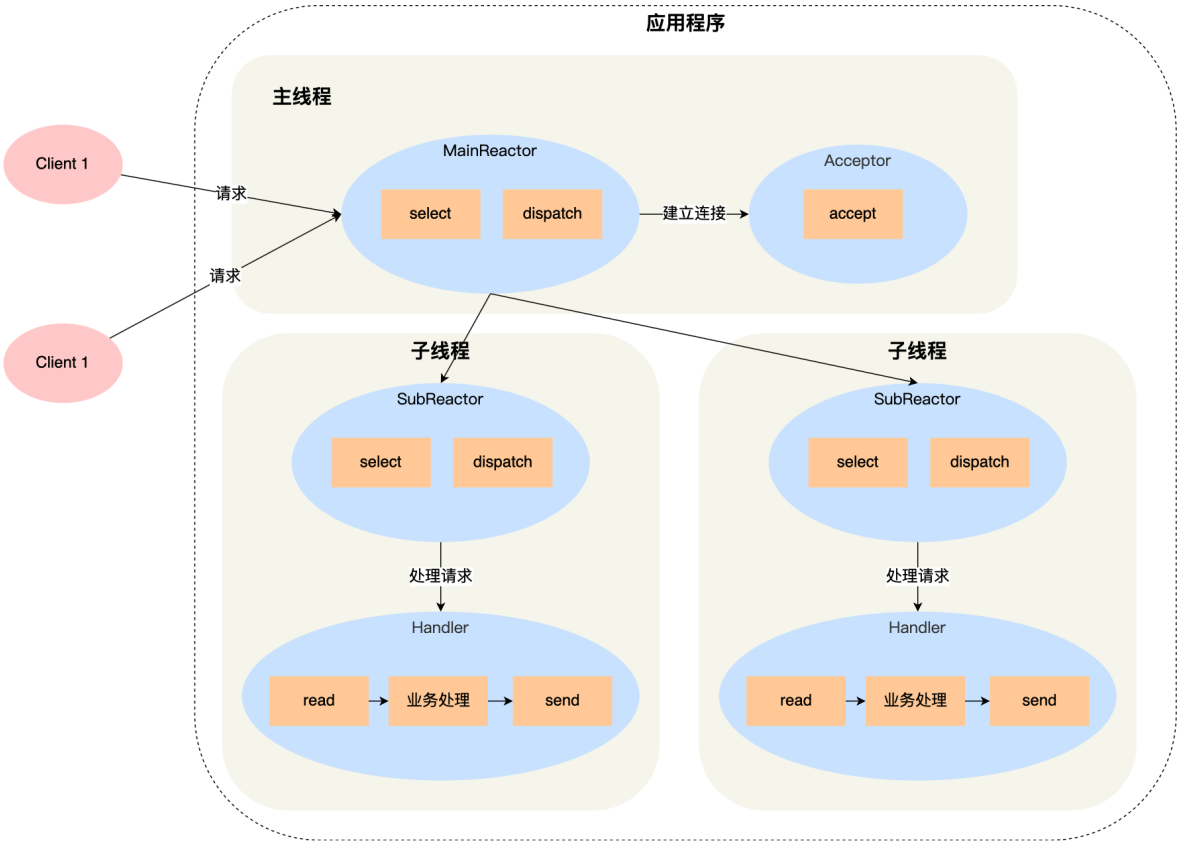



**优点：**可以充分利用多核 CPU 的处理能力。

**缺点：**多线程数据共享和访问比较复杂；Reactor 承担所有事件的监听和响应，在单线程中运行，高并发场景下容易成为性能瓶颈。

- **主从 Reactor 多线程**

  就是游双书里面的半同步/半反应堆模型，给这个归到了代码逻辑层面。




主线程和子线程分工明确，主线程只负责接收新连接，子线程负责完成后续的业务处理。

主线程和子线程的交互很简单，主线程只需要把新连接传给子线程，子线程无须返回数据，直接就可以在子线程将处理结果发送给客户端。

**优点：**父线程与子线程的数据交互简单职责明确，父线程只需要接收新连接，子线程完成后续的业务处理。

父线程与子线程的数据交互简单，Reactor 主线程只需要把新连接传给子线程，子线程无需返回数据。

这种模型在许多项目中广泛使用，包括 Nginx 主从 Reactor 多进程模型，Memcached 主从多线程，Netty 主从多线程模型的支持。

# preactor

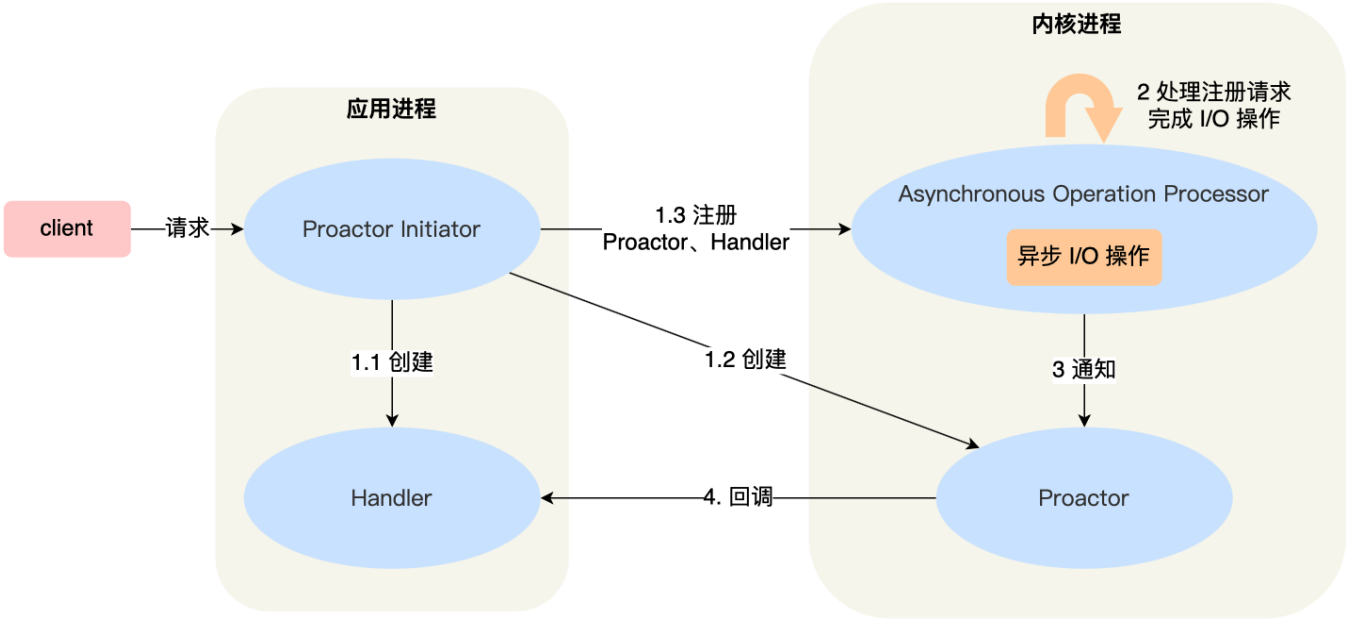
在 Reactor 模式中，Reactor 等待某个事件或者可应用或者操作的状态发生（比如文件描述符可读写，或者是 Socket 可读写）。

然后把这个事件传给事先注册的 Handler（事件处理函数或者回调函数），由后者来做实际的读写操作。

其中的读写操作都需要应用程序同步操作，所以 Reactor 是非阻塞同步网络模型。

如果把 I/O 操作改为异步，即交给操作系统来完成就能进一步提升性能，这就是异步网络模型 Proactor。

preactor模型如下图所示：




工作流程如下：

1. Proactor Initiator 负责创建 Proactor 和 Handler 对象，并将 Proactor 和 Handler 都通过
   Asynchronous Operation Processor 注册到内核；
2. Asynchronous Operation Processor 负责处理注册请求，并处理 I/O 操作；
3. Asynchronous Operation Processor 完成 I/O 操作后通知 Proactor；
4. Proactor 根据不同的事件类型回调不同的 Handler 进行业务处理；
5. Handler 完成业务处理；

理论上 Proactor 比 Reactor 效率更高，异步 I/O 更加充分发挥 DMA(Direct Memory Access，直接内存存取)的优势。
但是Proactor有如下缺点：

1. 编程复杂性，由于异步操作流程的事件的初始化和事件完成在时间和空间上都是相互分离的，因此开发异步应用程序更加复杂。应用程序还可能因为反向的流控而变得更加难以 Debug；
2. 内存使用，缓冲区在读或写操作的时间段内必须保持住，可能造成持续的不确定性，并且每个并发操作都要求有独立的缓存，相比 Reactor 模式，在 Socket 已经准备好读或写前，是不要求开辟缓存的；

3. 操作系统支持，Windows 下通过 IOCP 实现了真正的异步 I/O，而在 Linux 系统下，Linux 2.6 才引入，目前异步 I/O 还不完善。
4. Reactor处理耗时长的操作会造成事件分发的阻塞，影响到后续事件的处理；

> 可惜的是，在 Linux 下的异步 I/O 是不完善的，
> `aio` 系列函数是由 POSIX 定义的异步操作接口，不是真正的操作系统级别支持的，而是在用户空间模拟出来的异步，并且仅仅支持基于本地文件的 aio 异步操作，网络编程中的 socket 是不支持的，这也使得基于 Linux 的高性能网络程序都是使用 Reactor 方案。
>
> 而 Windows 里实现了一套完整的支持 socket 的异步编程接口，这套接口就是 `IOCP`，是由操作系统级别实现的异步 I/O，真正意义上异步 I/O，因此在 Windows 里实现高性能网络程序可以使用效率更高的 Proactor 方案。

# 文件(不带缓存的)I/O和标准(带缓存的)I/O

**首先要明确一个问题：有无缓存是相对于用户层面来说的，而不是系统内核层面。在系统内核层面，一直都存在有“内核高速缓存”**

- 不带缓存的概念

  > 所谓不带缓存，并不是指内核不提供缓存，而是在用户进程层次没有提供缓存。不带缓存的I/O只存在系统调用(write和read函数)，不是函数库的调用。**系统内核对磁盘的读写都会提供一个块缓冲（在有些地方也被称为内核高速缓存）**，当用write函数对其写数据时，直接调用系统调用，将数据写入到块缓存进行排队，当块缓存达到一定的量时，才会把数据写入磁盘。因此所谓的不带缓存的I/O是指用户进程层面不提供缓存功能（但内核还是提供缓冲的）。
  >
  > 文件I/O以文件标识符（整型）作为文件唯一性的判断依据。这种操作与系统有关，移植有一定的问题。

- 带缓存的概念

  > 与之相对的就是带缓存I/O。而带缓存的是在不带缓存的基础之上封装了一层，在用户进程层次维护了一个输入输出缓冲区，使之能跨OS，成为ASCI标准，称为标准IO库。其实就是在用户层再建立一个缓存区，这个缓存区的分配和优化长度等细节都是标准IO库代你处理好了，不用去操心。第一次调用带缓存的文件操作函数时标准库会自动分配内存并且读出一段固定大小的内容存储在缓存中。所以以后每次的读写操作并不是针对硬盘上的文件直接进行的，而是针

> 对内存中的缓存的。
>
> 不带缓存的文件操作通常都是系统提供的系统调用， 更加低级，直接从硬盘中读取和写入文件，由于IO瓶颈的原因，速度并不如意，而且原子操作需要程序员自己保证，但使用得当的话效率并不差。另外标准库中的带缓存文件IO 是调用系统提供的不带缓存IO实现的。

    - 因此，标准I/O函数有两个优点：
      1. 具有良好的移植性
      2. 利用用户层提供的缓存区（流缓冲）提高性能
    - 标准I/O函数缺点
        a. 不容易进行双向通信
        b. 有时可能频繁调用fflush函数
        c. 需要以FILE结构体指针的形式返回文件描述符

- 举例说明

> **带缓冲的I/O在往磁盘写入相同的数据量时，会比不带缓冲的I/O调用系统调用的次数要少。**比如内核缓冲存储器长度为100字节，在进行写操作时每次写入10个字节，则你需要调用10次write函数才能把内核缓冲区写满。但是要注意此时数据还在缓冲区，不在磁盘中，缓冲区满时才进行实际的I/O操作，把数据写入到磁盘，这样调用了太多次系统调用，显得效率很低。但是若调用标准I/O函数，假设用户层缓冲区为50字节（称为流缓存），则用fwrite将数据写入到流缓存，等到流缓存区存满之后再进入内核缓存区，在调用write函数将数据写入到内核缓冲区中，若内核缓冲区满或执行fflush操作，那么内核缓冲区的数据会写入到磁盘中

    - 无缓存IO操作数据流向路径：**数据——内核缓存区——磁盘**
    - 标准IO操作数据流向路径：**数据——流缓存区——内核缓存区——磁盘**

- apue三种io的总结

  在apue中有三种io类型，如下：

    i. 文件I/O（不带缓冲的I/O）：open、read、write、lseek、close
    ii. 标准I/O（带缓冲的I/O）：标准I/O库由ISO C标准说明
    iii. 高级I/O：非阻塞I/O、I/O多路转接、异步I/O、readv和writev

# 阻塞非阻塞，同步异步的区别

参考

参考2，这个比较符合我的想法

进程通讯层面，阻塞就是同步，非阻塞就是异步，一个意思

IO层面，就不一样。要记住，IO操作只有两个阶段：

1. 数据准备阶段
2. 内核缓冲区（内核空间）复制数据到用户进程缓冲区（用户空间）阶段

对于数据准备阶段，是阻塞和非阻塞的层面。对于数据从内核转移到用户空间，就是同步异步阶段。

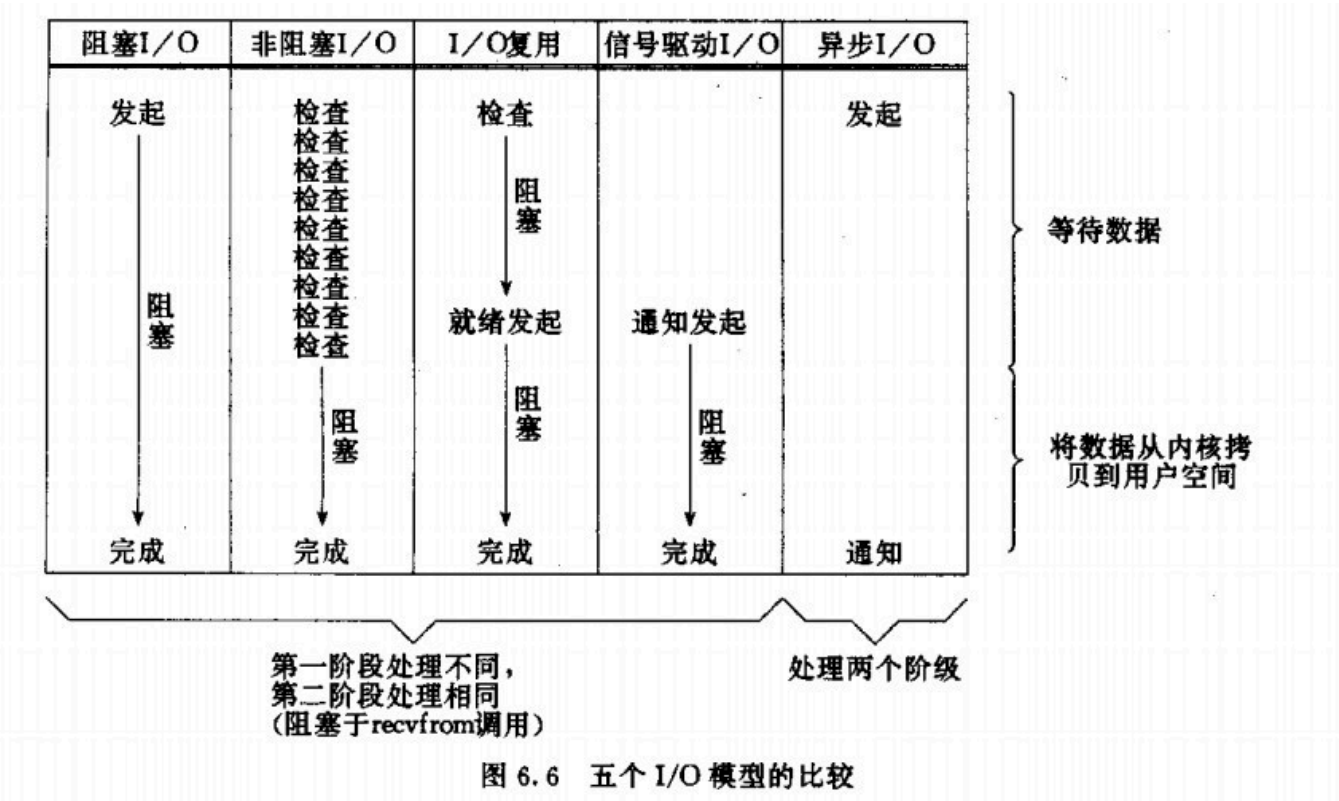



图 6.6　五个 I/O 模型的比较

阻塞和非阻塞的区别在于内核数据还没准备好时，用户进程在一阶段数据准备时是否会阻塞；

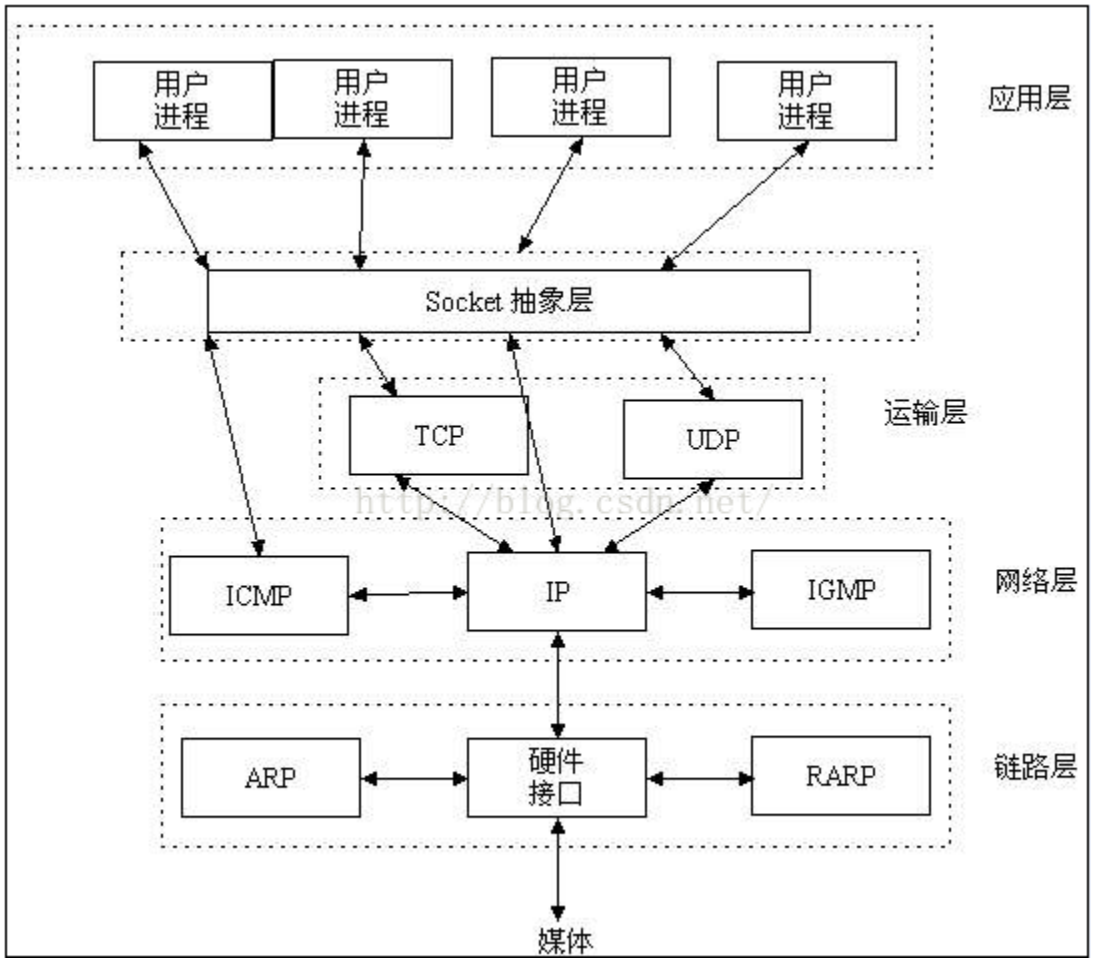
同步IO与异步IO的区别在于当数据从内核 `copy` 到用户空间时，用户进程是否会阻参与第二阶段的数据读写，是由用程序完成还是由内核完成。

# 对于套接字socket的理解

这里我类比一下插座和套接字，为什么这样类比呢？因为套接字的中文有插座的含义....

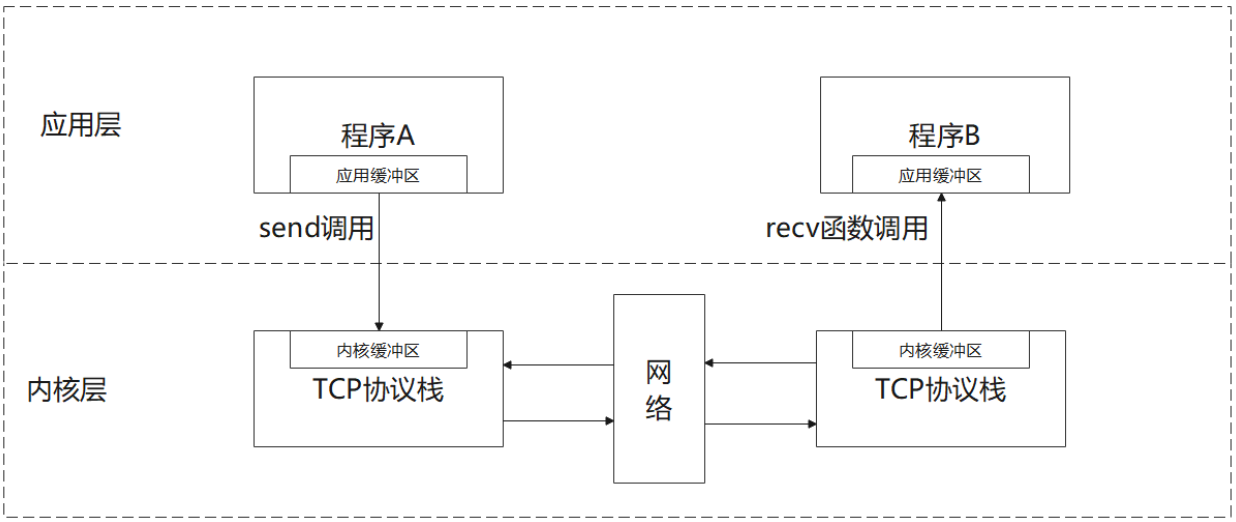
电器如何才能供电？电器需要连接上电网。如何连接到电网？需要把电器插销插到插座上，通过插座连接到电网，这样电器就有电可以工作

计算机如何收发消息？计算机需要连接上互联网。如何连接互联网？硬件层面我们可以拉一根网线，软件层面需要套接字。通过套接字连接到互联网，进而可以和互利网上的所有主机进行通信。




*socket 其实就是操作系统提供给程序员操作「网络协议栈」的接口，说人话就是，你能通过 socket* 的接口，来控制协议找工作，从而实现网络通信**

# C++网络通信中send和receive的为什么会阻塞




使用tcp协议进行通讯的双方，都各自有一个发送缓冲区和一个接收缓冲区。而缓冲区是有大小的，因此发生阻塞的本质原因是缓冲区满了，别的字节流消息无法进入缓冲区。

send函数只是将内存中的数据拷贝到内核中tcp的发送缓冲区或者说写缓冲区，但是什么时候发送数据是send无法控制的。同时tcp是一个可靠传输协议，发送端必须收到确认报文信息后才会清空发送缓冲区中的内容。对于读缓冲区来，收到数据放到自己的读缓冲区，同时要给发送端发送一个确认消息表明自己收到了信息。这个时候如果读缓冲区的数据被填满，由于滑动窗口协议，导致接收端不在读取数据，进而写缓冲区也会阻止发送数据。这个时候write函数就会阻塞。总结：**接收端接收数据的速度小于发送端发送数据的速度，导致接收端的读缓冲区填满，接收端发送报文给发送端告诉他我已经满了，先别发。这样发送端的写缓冲区被占满了，导致阻塞**

receive阻塞是因为读缓冲区中没数据。

记住，send是等到写缓冲区被填满之后才发送，但是write只要发现读缓冲区有数据，就将数据拷贝。

# send和receive在阻塞和非阻塞模式下的表现

```
//第一个参数：指定发送端套接字
//第二个参数：指明需要发送数据的缓冲区
//第三个参数：指明实际发送的数据的字节
//第四个参数：一般不写。可以临时设置为非阻塞
int send( SOCKET s, const char *buf, int len, int flags );

//第一个参数：指定接收端文件描述符
//第二个参数：指明一个缓冲区，存放接受来的数据
//第三个参数：指明缓冲区的长度
//第四个参数：一般不写。可以临时设置为非阻塞
int recv( SOCKET s, char *buf, int len, int flags);
```

- recv和send两种模式设置
  对于一个socket是阻塞还是非阻塞有两种方式来处理：
  i. 用fcntl函数

```
flags = fcntl(sockfd, F_GETFL, 0);                    //获取文件的flags值
fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);           //设
fcntl(sockfd,F_SETFL,flags&~O_NONBLOCK);              //设置成
```

  ii. 将recv和send函数的最后一个flag 参数设置成`MSG_DONTWAIT`
  注意！这种方法是临时的，不管之前是否阻塞

```
recv(sockfd, buff, buff_size,MSG_DONTWAIT);      //非阻塞模式的消息发送
send(scokfd, buff, buff_size, MSG_DONTWAIT);    //非阻塞模式的消息接受
```

- 整体来看
  - 当`socket`处于阻塞模式时,继续调用`send/recv`函数,程序会阻塞在`send/recv`调用处
  - 当`socket`处于非阻塞模式时,继续调用`send/recv`函数,会返回错误码
- 阻塞和非阻塞条件下read/recv的区别
  **阻塞和非阻塞的区别在于没有数据到达的时候是否立刻返回**
  读或者收的本质是从底层缓冲的数据copy到我们制定的buffer中

    - 阻塞条件下
        a. 如果读缓冲区中没有数据会一直等待
        b. 如果读缓冲区有数据，则会把数据读到用户指定的缓冲区。如果读取的数据量比函数参数中指定的长度要小，read会返回读到的数据长度。
        一般情况下我们都需要采取循环读的方式读取数据
    - 非阻塞条件下
        a. 如果没有数据直接返回EWOULDBLOCK
        b. 读缓冲区有数据，有多少读多少
- 阻塞和非阻塞条件下send/write的区别
  写或者发的本质是把buffer(用户态)中的数据copy到内核态，然后就返回了。发送操作是由系统底层和tcp协议进行。send返回成功，表示数据已经copy到底层缓冲区，而不是表示数据已经发送
    - 阻塞情况下
      一直等待，直到write将数据发送完(发送过程中可能会中断)。
      读的时候我们并不知道是否有数据，以及数据是何时结束发送，如果一直等待就会造成死循环
      对于写由于长度是已知的，所以可以随便写，直到写完。不过写会被打断，造成一次write只能写一部分，可以用循环write。
    - 非阻塞情况下
      对于本地网络阻塞的情况来说，写缓冲区没有足够的内存来存储buf中的数据，因此会出现写不成功的情况。非阻塞不会等到数据全部发送再返回，而是写多少算多少，
      返回值是WSAEWOULDDBLOCK
- write和read返回值详解

# IO多路复用

IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合：

1. 当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。
2. 如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。

3. 如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。
4. 如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。

与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

# ☆为什么单线程或者单进程下是用select或者epoll就能实现多个客户端的并发？

答：在套接字中有两种文件描述符，一种是用于监听的文件描述符，一种是用于通信的文件描述符。（对服务器端来说用于监听的文件描述符只有1个，用于通信的文件描述符有n个对应n个连接的客户端）且不论是哪一种文件描述符，都有输入缓冲和输出缓冲。对于用于监听的文件描述符，其读缓冲区会存储来自客户端的链接请求，调用accept函数，如果读缓冲区有数据则解除阻塞，返回对应客户端的文件描述符，用来和响应客户端通信。对于通信的文件描述符，其读缓冲区存储客户端发送来的数据，调用read就能从缓冲区读取，如果没数据就阻塞。写缓冲区同理。从上面来看，accept、read、write函数是互斥的，在单进程中如果有一个陷入了阻塞状态，其余的也没办法工作。因此单线程或者单进程情况下服务器端无法阻止阻塞问题。

单进程或单线程下这个问题从使用者或者是用户层面来说无法解决，因此要把这个问题交给内核处理。程序媛不需要维护文件描述符的读缓冲区和写缓冲区，内核可以同时检测多个文 件描述符缓冲区的变化。比如检测读缓冲区，看是否有有数据，写缓冲区是否有剩余空间等等。如果满足，就会形成事件，内核会将满足条件的文件描述符告诉我们。

总结一下就是本来程序员来检测IO的使用情况，使用阻塞函数检查，交给了内核做。只有满足事件的文件描述符才调用阻塞函数，因此就不会形成阻塞。

# select

- 原理
  select函数将许多个文件描述符集中到一起进行监视。
  使用fd_set数组保存被监视的文件描述符的变化，这个数组是以位存储在内核中的。即该数组所在的索引就是对应文件描述符的id。
  首先创建一个保存监视的数组`fd_set set`，然后将所有文件描述符的状态初始化为0 `FD_ZERO(&set)`，再用`FD_ISSET(i,&set)`查找发生状态的文件描述符，循环查找。
- 存在额问题

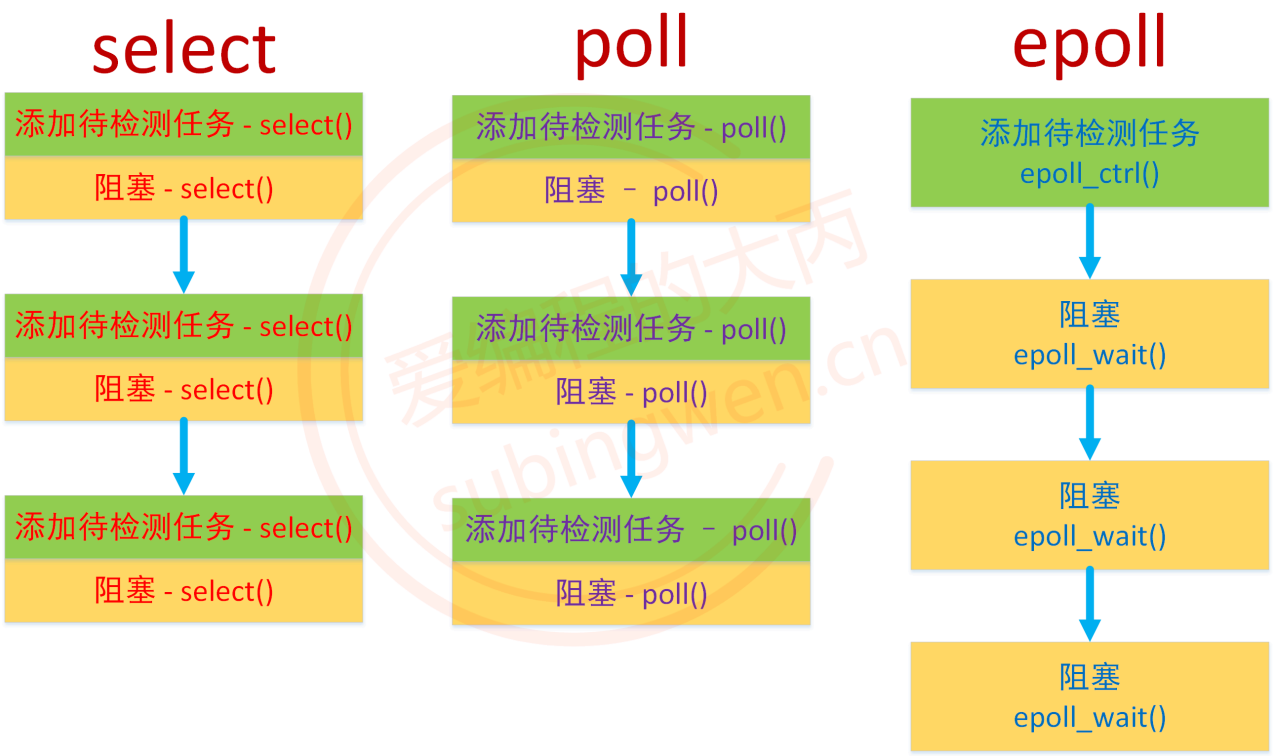
i. 单个进程可监视的文件描述符的数量被限制，即监听的端口有限，这个数目的限制和内存有很大关系，32位默认为1024个，对于64位机默认为2048个
ii. 对这个数组的是线性扫描，时间复杂度为O(n)
iii. 这个是最主要的开销。**每次调用select函数的时候会向操作系统传递监视对象信息。记住，应用程序向操作系统传递数据会造成很大的开销。**select函数是监视套接字变化的函数，但是套接字是由操作系统来管理的，所以select必须要借助操作系统才能完成功能。所以select函数本身就是一个系统调用。
（另一种说法：不是拷贝进内核，而是通过mmap系统调用开辟了一坨内核态用户态的共享空间，数据放在了这个共享空间了）

> 最low的就是在用户代码中自旋实现所有阻塞socket的监听。但是每次判断socket是否产生数据，都涉及到用户态到内核态的切换。
> 于是select改进：将fd_set传入内核态，由内核判断是否有数据返回；
> 然后最low的只能使用自旋来时刻的去判断socket列表中是否有数据达到。
> 于是select改进：使用等待队列，让线程在没有资源时park（阻塞），当有数据到达时唤醒select线程，去处理socket。

# epoll

> epoll就是解耦了select的模型：

> 设想一个场景：有100万用户同时与一个进程保持着TCP连接，而每一时刻只有几十个或几百个TCP连接是活跃的(接收TCP包)，也就是说在每一时刻进程只需要处理这100万连接中的一小部分连接。那么，如何才能高效的处理这种场景呢？进程是否在每次询问操作系统收集有事件发生的TCP连接时，把这100万个连接告诉操作系统，然后由操作系统找出其中有事件发生的几百个连接呢？实际上，在Linux2.4版本以前，那时的select或者poll事件驱动方式是这样做的.
>
> 这里有个非常明显的问题，即在某一时刻，进程收集有事件的连接时，其实这100万连接中的大部分都是没有事件发生的。因此如果每次收集事件时，都把100万连接的套接字传给操作系统(这首先是用户态内存到内核态内存的大量复制)，而由操作系统内核寻找这些连接上有没有未处理的事件，将会是巨大的资源浪费，然后select和poll就是这样做的，因此它们最多只能处理几千个并发连接。而epoll不这样做，它在Linux内核中申请了一个简易的文件系统，把原先的一个select或poll调用分成了3部分：




首先介绍一下epoll的函数，主要由三个：

```
int epoll_create(int size);
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);
```

使用epoll_create**向操作系统申请创建文件描述符的空间**（这个文件描述符都是在内核空间中），即建立一个epoll对象。epoll_ctl将刚创立的socket加入到epoll中进行监控，或者将某个正在监控的socket移除，不在监控。epoll_wait即监控socket有状态发生变化时候，就返回用户态的进程

> 从这三个函数就可以看到epoll函数的优越性。当调用select时需要传递所有监视的socket给系统调用，意味着需要将用户态的fd_set拷贝到内核态，可想而知效率非常低效。但是epoll中内核通过epoll_ctl函数已经拿到监视socket列表。所以，实际上在你调用epoll_create后，内核就已经在内核态开始准备帮你存储要监控的句柄了，每次调用epoll_ctl只是在往内核的数据结构里塞入新的socket句柄。
>
> 1. 调用epoll_create建立一个epoll对象(在epoll文件系统中给这个句柄分配资源)；
> 2. 调用epoll_ctl向epoll对象中添加这100万个连接的套接字；
> 3. 调用epoll_wait收集发生事件的连接。

epoll会开辟出自己的内核高速cache区，用于安置每一个我们想监控的socket，这些socket会以红黑树的形式保存在内核cache里，以支持快速的查找、插入、删除。同时还会建立一个双向链表，每个节点保存着满足读写条件，返回给用户的事件。

epoll高效的原因主要是epoll_wait这个函数。由于我们在调用epoll_create时，内核除了帮我们在epoll文件系统里建了个file结点，在内核cache里建了个红黑树用于存储以后epoll_ctl传来的socket外，还会再建立一个list链表，用于存储准备就绪的事件，当epoll_wait调用时，仅仅观察这个list链表里有没有数据即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll_wait非常高效。而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll_wait仅需要从内核态copy少量的文件描述符到用户态而已，如何能不高效？！

那么，这个准备就绪list链表是怎么维护的呢？**当我们执行epoll_ctl时，除了把socket放到epoll文件系统里对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数**，告诉内核，如果这个文件描述符的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，数据copy到内核中后就来把socket插入到准备就绪链表里了。

**总结：　一颗红黑树，一张准备就绪句柄链表，少量的内核cache，就帮我们解决了大并发下的socket处理问题。**

详解epoll

# select和epoll效率

**select原理概述**：

1. 从用户空间拷贝fd_set到内核空间；
2. 遍历所有fd，只要有事件触发，系统调用返回，将fd_set从内核空间拷贝到用户空间，回到用户态，用户就可以对相关的fd作进一步的读或者写操作了。

**epoll原理概述**：

1. 调用epoll_create
    i. 内核帮我们在epoll文件系统里建了个file结点；
    ii. 在内核cache里建了个红黑树用于存储以后epoll_ctl传来的socket；
    iii. 建立一个list链表，用于存储准备就绪的事件。
2. 调用epoll_ctl

i. 把socket放到epoll文件系统里file对象对应的红黑树上；
ii. 给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。

3. 调用epoll_wait
   观察list链表里有没有数据。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll_wait仅需要从内核态copy少量的句柄到用户态而已。

**select缺点:**

1. 最大并发数限制：使用32个整数的32位，即32*32=1024来标识fd，虽然可修改，但是有以下第二点的瓶颈；
2. 效率低：每次都会线性扫描整个fd_set，集合越大速度越慢；
3. 内核/用户空间内存拷贝问题。

**epoll的提升**：

1. 本身没有最大并发连接的限制，仅受系统中进程能打开的最大文件数目限制；
2. 效率提升：只有活跃的socket才会主动的去调用callback函数；
3. 省去不必要的内存拷贝：epoll通过内核与用户空间mmap同一块内存实现。

**总结：**需要看所有被观察的事件是否都活跃。

假设现在有1024个fd，select 和epoll 都同时维护他，假设这些fd 都是活跃的，这种情况下select一次扫描 可以扫描1024个fd，空闲的fd很少，但是epoll 就有可能不一样了， epoll 是先注册等待回调， 有可能出现1024次回调。所以不好说。

如果select 和epoll 同时维护1024个fd， 但是每次只有一个fd有事件，这种情况下 select 每次都会扫描所有的fd， 对比于epoll 每次只有一个fd 回调。 select 做了很多无用功， 此时应该epoll的效率高吧！！

或者在短连接多的时候， 一个连接使用epoll 会触发epoll_ctrl_add/del 两次系统调用，但是select 只有一次扫描， 此时 也许select 效率性能更高。

# epoll的水平触发模式（LT）

默认模式

在LT（水平触发）模式下，只要这个文件描述符还有数据可读，每次 epoll_wait都会返回它的事件，提醒用户程序去操作。

# epoll的边缘触发模式（ET）

ET（边缘触发）模式下，在它检测到有 I/O 事件时，通过 epoll_wait 调用会得到有事件通知的文件描述符，**对于每一个被通知的文件描述符，如可读，则必须将该文件描述符一直读到空**，让 errno 返回 EAGAIN 为止，否则下次的 epoll_wait 不会返回余下的数据，会丢掉事件。如果ET模式不是非阻塞的，那这个一直读或一直写势必会在最后一次阻塞。

为什么会有ET模式？

答：如果采用EPOLLLT模式的话，系统中一旦有大量你不需要读写的就绪文件描述符，它们每次调用epoll_wait都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率.。而采用EPOLLET这种边缘触发模式的话，当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符。减少epoll_wait的调用次数，系统调用开销是很大的

# tcp 在调用connect失败后要不要重新socket？

connect（套接字默认阻塞）出错返回的情况：

1. 调用connect时内核发送一个SYN分节，若无响应则等待6s后再次发送一个，仍无响应则等待24s再发送一个，若总共等了75s后仍未收到响应则返回ETIMEDOUT错误；
2. 若对客户的SYN的响应是RST，则表示该服务器主机在我们指定的端口上面没有进程在等待与之连接，例如服务器进程没运行，客户收到RST就马上返回ECONNREFUSED错误；
3. 若客户发出的SYN在中间的某个路由上引发了一个“destination unreachable”（目的不可达）ICMP错误，客户主机内核保存该消息，并按1中所述的时间间隔发送SYN，在某个规定的时间（4.4BSD规定75s）仍未收到响应，则把保存的ICMP错误

作为EHOSTUNREACH或ENETUNREACH错误返回给进程。

若connect失败则该套接字不再可用，必须关闭，我们不能对这样的套接字再次调用connect函数。在每次connect失败后，都必须close当前套接字描述符并重新调用socket。

# Linux零拷贝技术

splice( )函数

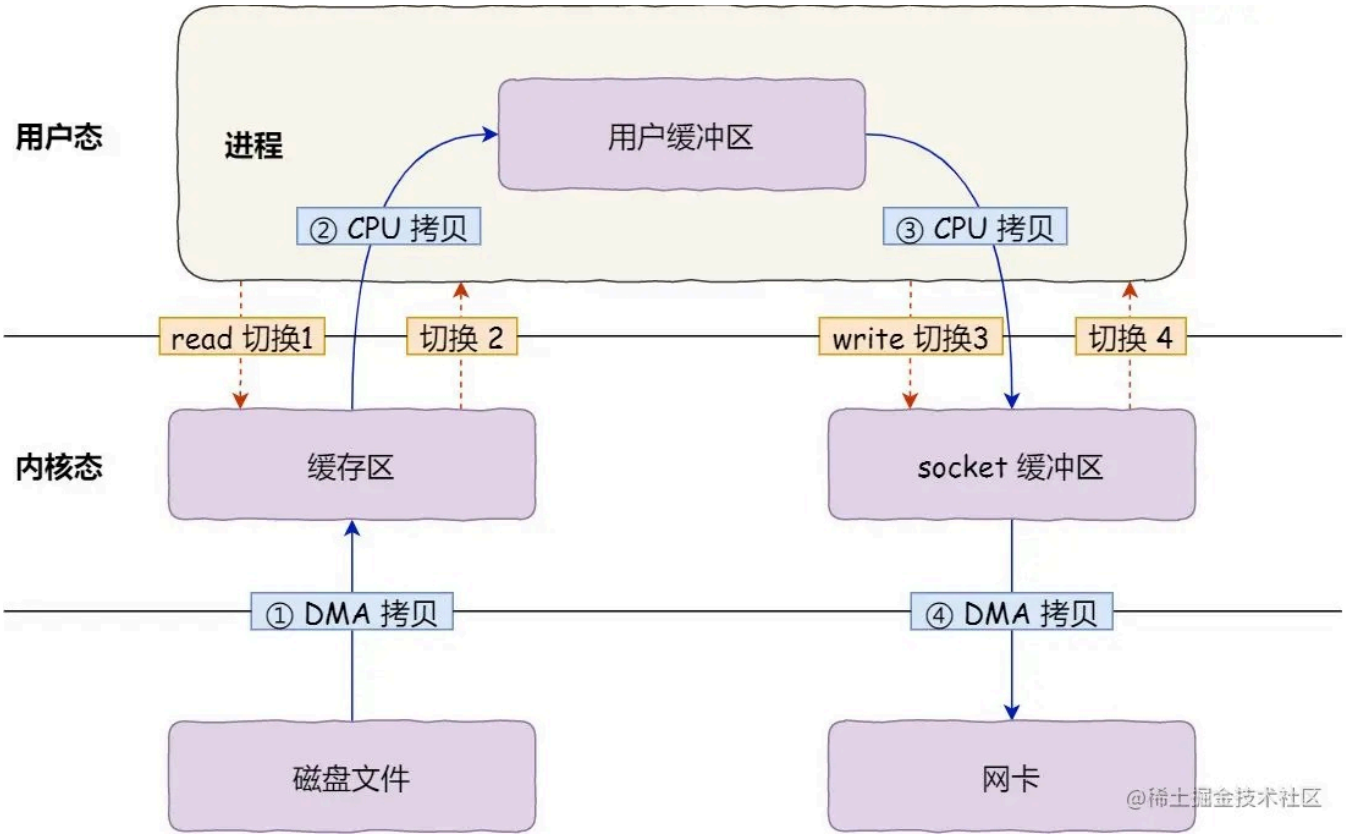
tee( )函数

## 概述：

零拷贝（Zero-Copy）是一种 `I/O` 操作优化技术，可以快速高效地将数据从文件系统移动到网络接口，而不需要将其从内核空间复制到用户空间。其在 `FTP` 或者 `HTTP` 等协议中可以显著地提升性能。

## 由来：

如果服务端要提供文件传输的功能，我们能想到的最简单的方式是：将磁盘上的文件读取出来，然后通过网络协议发送给客户端。传统 I/O 的工作方式是，数据读取和写入是从用户空间到内核空间的复制，而内核空间的数据是通过操作系统层面的 I/O 接口从磁盘读取或写入。其过程如下图所示：




可以想想一下这个过程。服务器读从磁盘读取文件的时候，发生一次系统调用，产生用户态到内核态的转换，将磁盘文件拷贝到内核的内存中。然后将位于内核内存中的文件数据拷贝到用户的缓冲区中。用户应用缓冲区需要将这些数据发送到socket缓冲区中，进行一次用户态到内核态的转换，复制这些数据。此时这些数据在内核的socket的缓冲区中，在进行一次拷贝放到网卡上发送出去。

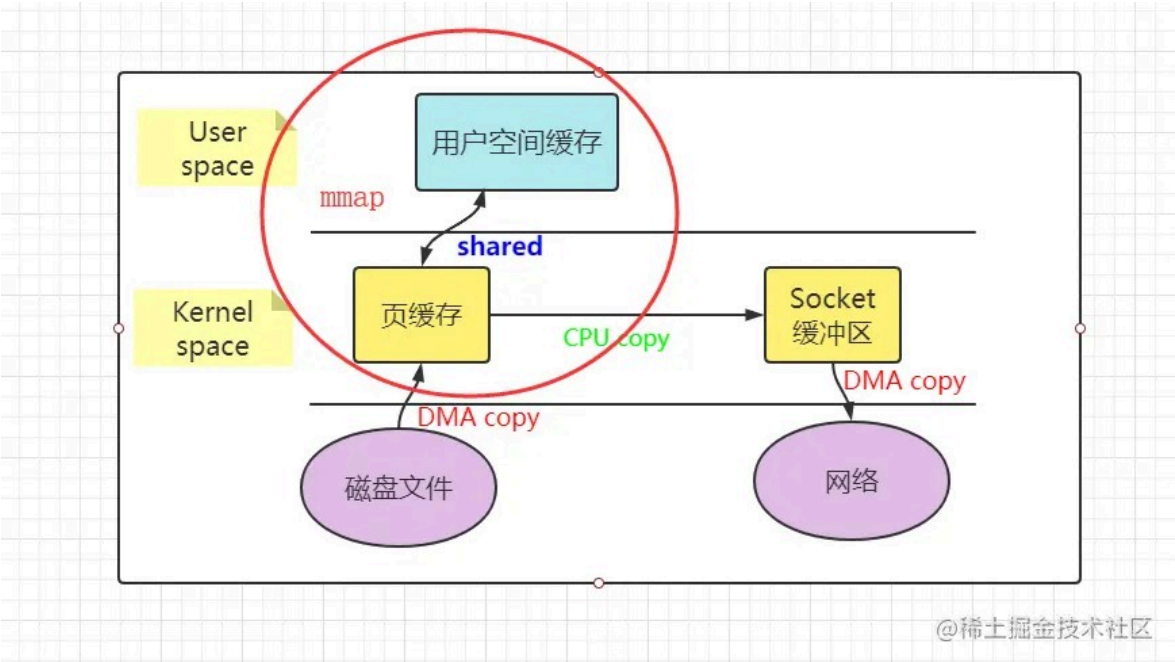
所以整个过程一共进行了四次拷贝，四次内核和用户态的切换。这种简单又传统的文件传输方式，存在冗余的上文切换和数据拷贝，在高并发系统里是非常糟糕的，多了很多不必要的开销，会严重影响系统性能。

# 零拷贝原理

零拷贝主要是用来解决操作系统在处理 I/O 操作时，频繁复制数据的问题。关于零拷贝主要技术有 `mmap+write`、`sendfile`和`splice`等几种方式。

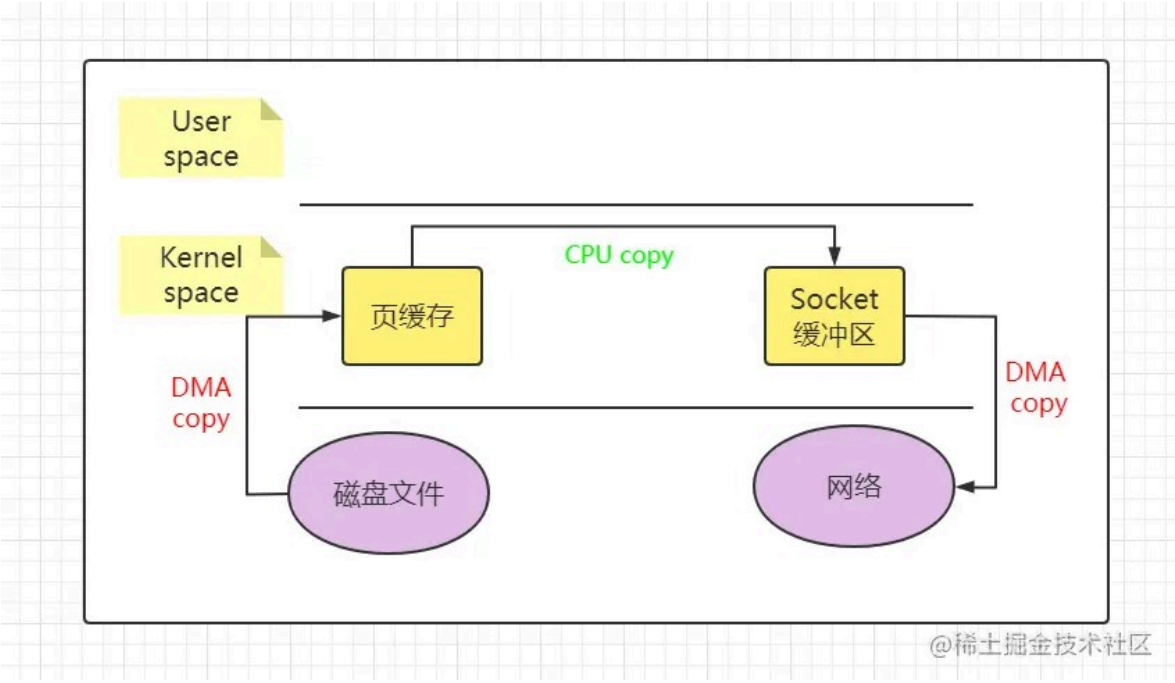
看完下图会发现其实零拷贝就是少了CPU拷贝这一步，磁盘拷贝还是要有的

- mmap/write 方式




把数据读取到内核缓冲区后，应用程序进行写入操作时，直接把内核的`Read Buffer`的数据复制到`Socket Buffer`以便写入，这次内核之间的复制也是需要CPU的参与的。

- sendfile 方式




可以看到使用sendfile后，没有用户空间的参与，一切操作都在内核中进行。但是还是需要1次拷贝

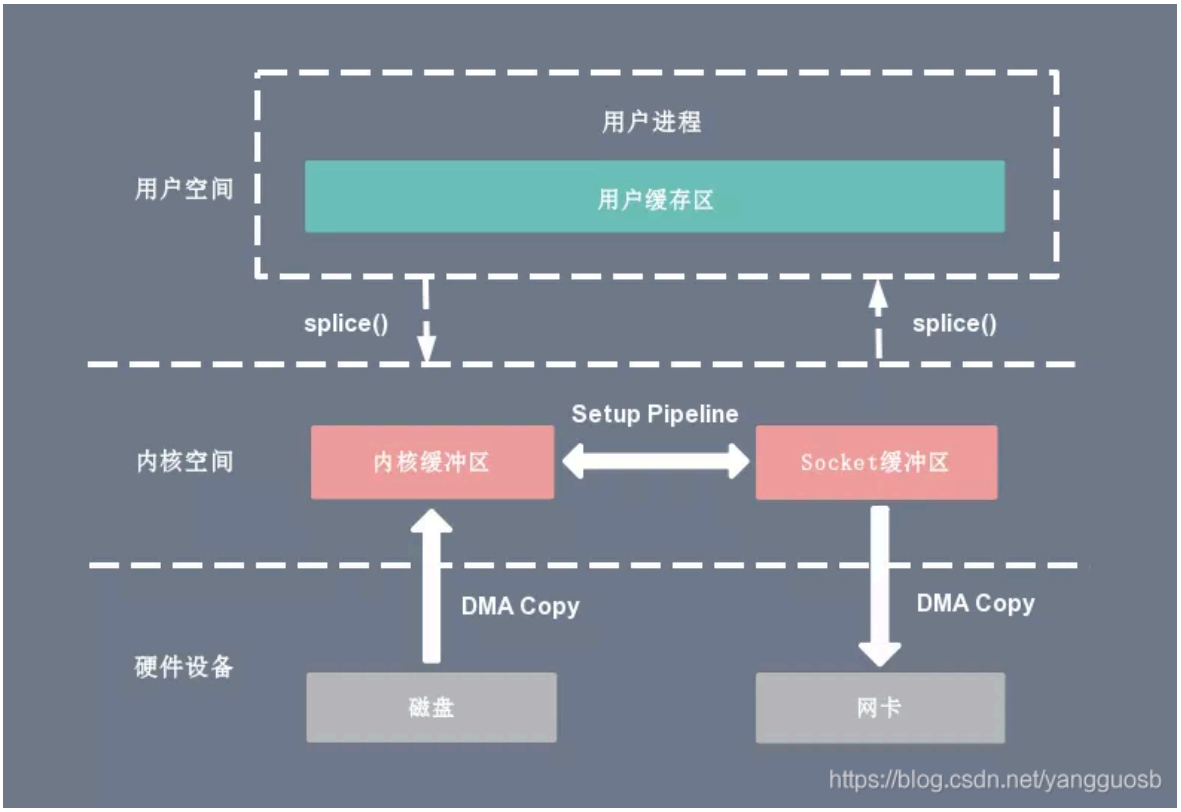
- 带有 scatter/gather 的 sendfile方式

  Linux 2.4 内核进行了优化，提供了带有`scatter/gather`的 sendfile 操作，这个操作可以把最后一次`CPU COPY`去除。其原理就是在内核空间 Read BUffer 和 Socket

Buffer 不做数据复制，而是将 Read Buffer 的内存地址、偏移量记录到相应的 Socket Buffer 中，这样就不需要复制。其本质和虚拟内存的解决方法思路一致，就是内存地址的记录。

**注意： sendfile适用于文件数据到网卡的传输过程，并且用户程序对数据没有修改的场景；**

- splice 方式




其实就是CPU 在内核空间的读缓冲区（read buffer）和网络缓冲区（socket buffer）之间建立管道（pipeline）直接把数据传过去了，不去要CPU复制了

# accept()、connect()发生在三次握手的哪一步？

image看上图就很明白了，刚准备发SYN同步报文时候，connect函数阻塞，然后服务端收到SYN同步报文的时候，调用accept()阻塞，服务器发送SYN+ACK给客户端，此时connect连接上就返回了，然后等服务器收到ACK报文时，accept也返回。

所以只说返回状态的话connect成功返回是在第二步，accept成功返回是在第三步

# accept的队列

处于“LISTENING”状态的TCP socket，有两个独立的队列：

1. SYN队列(SYN Queue)
2. Accept队列(Accept Queue)

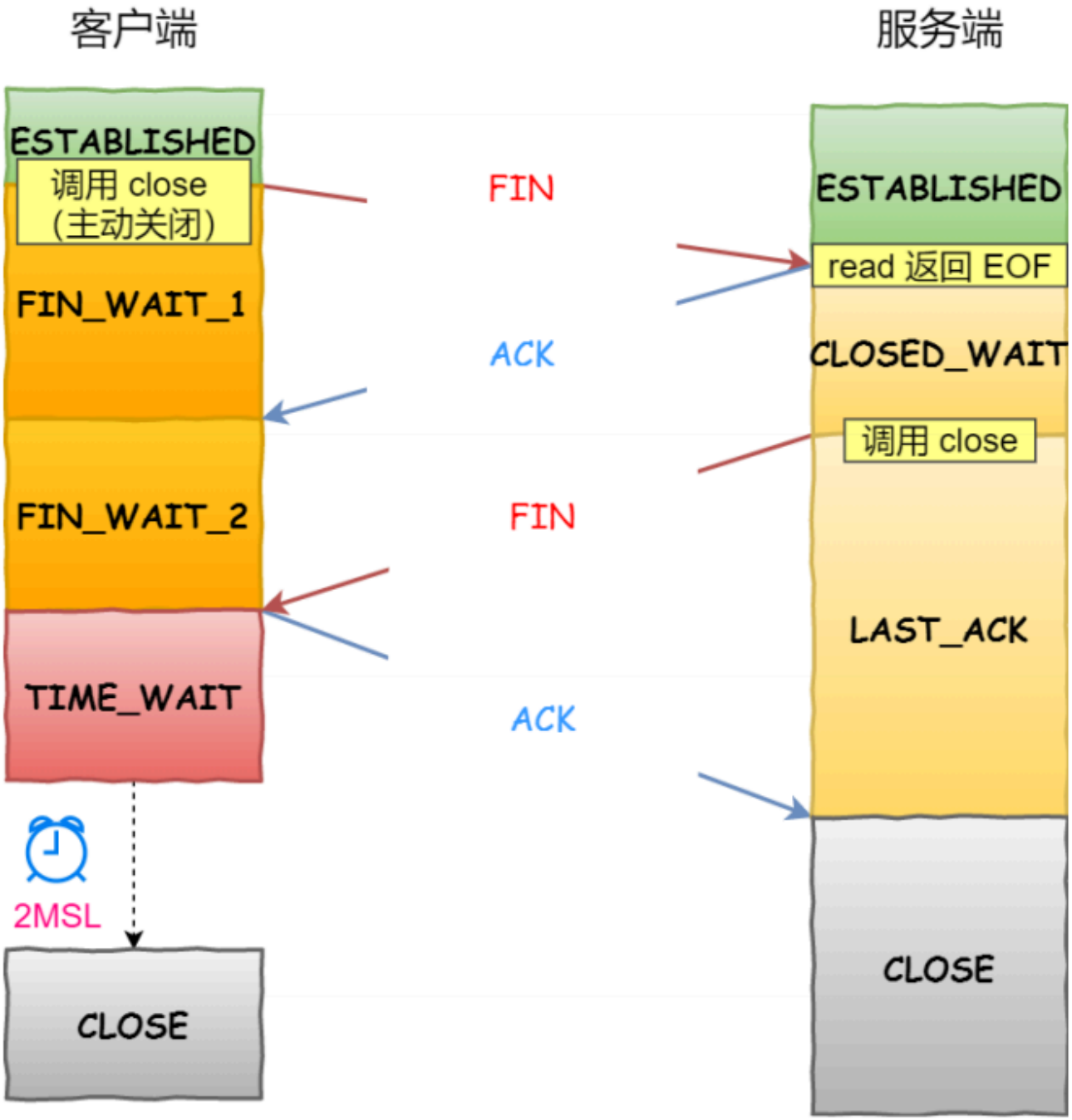
**SYN队列**

SYN队列存储了收到SYN包的连接(对应内核代码的结构体： struct inet_request_sock)。它的职责是回复SYN+ACK包，并且在没有收到ACK包时重传，直到超时。发送完SYN+ACK之后，SYN队列等待从客户端发出的ACK包(也即三次握手的最后一个包)。当收到ACK包时，首先找到对应的SYN队列，再在对应的SYN队列中检查相关的数据看是否匹配，如果匹配，内核将该连接相关的数据从SYN队列中移除

**accept队列**

内核将该连接相关的数据从SYN队列中移除，创建一个完整的连接(对应内核代码的结构体： struct inet_sock )，并将这个连接加入Accept队列。Accept队列中存放的是已建立好的连接，也即等待被上层应用程序取走的连接。当进程调用accept()，这个socket从队列中取出，传递给上层应用程序。一般来说都是select或者epoll上有事件发生时再取走

# 调用close()在哪一步？




首先注意EOF这个东西。当客户端调用close主动断开时，会在FIN报文后面放入一个文件结束符EOF，会放在服务器端读缓存队列的最后，当接收到EOF时，服务器需要处理这种异常情况，表示以后不会有任何数据到达。因此服务器会返回一个ACK确认报文然后进入closed_wait状态

等服务器处理完后，发送FIN报文后也调用close。

# Posix 信号量与System v信号量的区别

参考

参考

参考

Posix:是“可移植操作系统接口（Portable Operating System Interface ）的首字母简写，但它并不是一个单一的标准，而是一个电气与电子工程学会即IEEE开发的一系列标准，它还是由ISO（国际标准化组织）和IEC（国际电工委员会）采纳的国际标准。

System v是Unix操作系统众多版本的一个分支，它最初是由AT&T在1983年第一次发布，System v一共有四个版本，而最成功的是System V Release 4，或者称为SVR4。这样看来，一个是Unix 的标准之一（另一个标准是Open Group），一个是Unix众多版本的分支之一（其他的分支还有Linux跟BSD），应该来说，Posix标准正变得越来越流行，很多厂家开始采用这一标准。

区别如下：

1. System v信号量指的是计数信号量集；Posix 信号量指的是单个计数信号量。
2. Posix信号量是基于内存的，即信号量值是放在共享内存中的，它是由可能与文件系统中的路径名对应的名字来标识的。System v信号量测试基于内核的，它放在内核里面，相同点都是它们都可以用于进程或者线程间的同步。
3. POSIX信号量常用于线程；system v信号量常用于进程的同步。
4. POSIX 信号量的头文件是 <semaphore.h>，而 System V 信号量的头文件是 <sys/sem.h>。

# 信号通知流程和处理机制

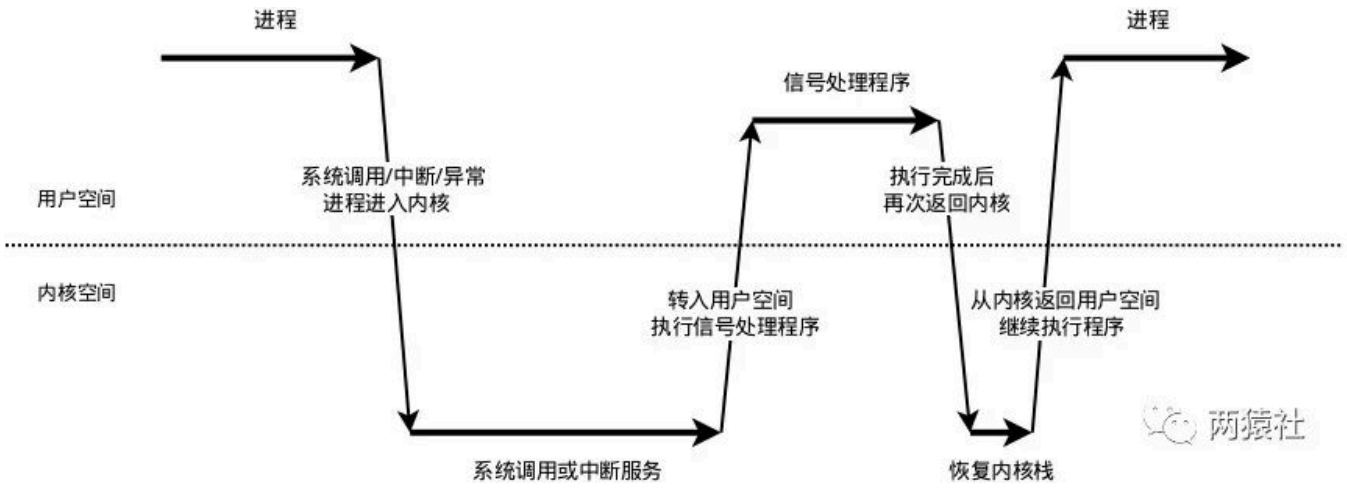
Linux下的信号采用的异步处理机制，信号处理函数和当前进程是两条不同的执行路线。具体的，当进程收到信号时，操作系统会中断进程当前的正常流程，转而进入信号处理函数执行操作，完成后再返回中断的地方继续执行。为避免信号竞态现象发生，信号处理期间系统不会再次

触发它。所以，为确保该信号不被屏蔽太久，信号处理函数需要尽可能快地执行完毕。

一般的信号处理函数需要处理该信号对应的逻辑，当该逻辑比较复杂时，信号处理函数执行时间过长，会导致信号屏蔽太久。提供一种解决方案是，信号处理函数仅仅发送信号通知程序主循环，将信号对应的处理逻辑放在程序主循环中，由主循环执行信号对应的逻辑代码。

> **统一事件源**
>
> 统一事件源，是指将信号事件与其他事件一样被处理。
>
> 具体的，信号处理函数使用管道将信号传递给主循环，信号处理函数往管道的写端写入信号值，主循环则从管道的读端读出信号值，使用I/O复用系统调用来监听管道读端的可读事件，这样信号事件与其他文件描述符都可以通过epoll来监测，从而实现统一处理。

信号处理机制如下图：




- 信号的接收
    - 接收信号的任务是由内核代理的，当内核接收到信号后，会将其放到对应进程的信号队列中，同时向进程发送一个中断，使其陷入内核态。注意，此时信号还只是在队列中，对进程来说暂时是不知道有信号到来的。
- 信号的检测
    - 进程从内核态返回到用户态前进行信号检测
    - 进程在内核态中，从睡眠状态被唤醒的时候进行信号检测
    - 进程陷入内核态后，有两种场景会对信号进行检测：
    - 当发现有新信号时，便会进入下一步，信号的处理。
- 信号的处理

    - ( **内核** )信号处理函数是运行在用户态的，调用处理函数前，内核会将当前内核栈的内容备份拷贝到用户栈上，并且修改指令寄存器（eip）将其指向信号处理函数。
    - ( **用户** )接下来进程返回到用户态中，执行相应的信号处理函数。
    - ( **内核** )信号处理函数执行完成后，还需要返回内核态，检查是否还有其它信号未处理。
    - ( **用户** )如果所有信号都处理完成，就会将内核栈恢复（从用户栈的备份拷贝回来），同时恢复指令寄存器（eip）将其指向中断前的运行位置，最后回到用户态继续执行进程。
- 至此，一个完整的信号处理流程便结束了，如果同时有多个信号到达，上面的处理流程会在第2步和第3步骤间重复进行。

# 服务器端有100万个TCP长连接，可能会出现的问题

参考1

参考2

# epoll是同步还是异步的？

1. IO层面
2. 消息处理层面
3. 从IO层面来看,epoll绝对是同步的；
4. 从消息处理层面来看，epoll是异步的.

select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的（可能通过while循环来检测内核将数据准备的怎么样了，而不是属于内核的一种通知用户态机制），仍然需要read、write去读写数据。

用户线程定期轮询epoll文件描述符上的事件，事件发生后，读取事件对应的epoll_data，该结构中包含了文件fd和数据地址，由于采用了mmap，程序可以直接读取数据（epoll_wait函数）。有人把epoll这种方式叫做同步非阻塞（NIO），因为用户线程需要不停地轮询，自己读取数据，

看上去好像只有一个线程在做事情。也有人把这种方式叫做异步非阻塞（AIO），因为毕竟是内核线程负责扫描fd列表，并填充事件链表的。个人认为真正理想的异步非阻塞，应该是内核线程填充事件链表后，主动通知用户线程，或者调用应用程序事先注册的回调函数来处理数据，如果还需要用户线程不停的轮询来获取事件信息，就不是太完美了，所以也有不少人认为epoll是伪AIO，还是有道理的。

# 原子操作的原理

原子操作（atomic operation）意为”不可被中断的一个或一系列操作”

# signal机制

# 使用epoll或select时需要将socket设为非阻塞吗？

**先说结论**：

需要的。但是一个 socket 是否设置为阻塞模式，只会影响到 connect/accept/send/recv 等四个 socket API 函数，不会影响到 select/poll/epoll_wait 函数，后三个函数的超时或者阻塞时间是由其函数自身参数控制的。

**socket 是否被设置成阻塞模式对下列 API 造成的影响**：

connfd：该端调用 connect 函数主动发起连接；

listenfd：调用 listen 函数发起侦听的一端（服务端）；即监听的socket

clientfd：调用 accept 函数接受连接，由 accept 函数返回的 socket（服务端）。即连接的 socket

当connfd 被设置成阻塞模式时（默认行为，无需设置），connect 函数会一直阻塞到连接成功或超时或出错，超时值需要修改内核参数。当 connfd 被设置成非阻塞模式，无论连接是否建立成功，connect 函数都会立刻返回，那如何判断 connect 函数是否连接成功呢？接下来使用

select 和 epoll_wait 函数去判断 socket 是否可写即可，当然，Linux 系统上还需要额外加一步——使用 getsockopt 函数判断此时 socket 是否有错误(因为select中通知有数据达到但是可能数据error checksum或者discard了)，这就是所谓的异步 connect 或者叫非阻塞 connect。

当 listenfd 设置成阻塞模式（默认行为，无需额外设置）时，如果连接 pending(待办) 队列中有需要处理的连接，accept 函数会立即返回，否则会一直阻塞下去，直到有新的连接到来。当 listenfd 设置成非阻塞模式，无论连接 pending 队列中是否有需要处理的连接，accept 都会立即返回，不会阻塞。如果有连接，则 accept 返回一个大于 0 的值，这个返回值即是我们上文所说的 clientfd；如果没有连接，accept 返回值小于 0，错误码 errno 为 EWOULDBLOCK（或者是 EAGAIN，这两个错误码值相等）。

当 connfd 或 clientfd 设置成阻塞模式时：send 函数会尝试发送数据，如果对端因为 TCP 窗口太小导致本端无法将数据发送出去，send 函数会一直阻塞直到对端 TCP 窗口变大足以发数据或者超时；recv 函数则正好相反，如果此时没有数据可收获，recv函数会一直阻塞直到收取到数据或者超时，有的话，取到数据后返回。send 和 recv 函数的超时时间可以分别使用 SO_SNDTIMEO 和 SO_RCVTIMEO 两个 socket 选项来设置。当 connfd 或 clientfd 设置成非阻塞模式时，send 和 recv 函数都会立即返回，send 函数即使因为对端 TCP 窗口太小发不出去也会立即返回，recv 函数如果无数据可收也会立即返回，此时这两个函数的返回值都是 -1，错误码 errno 是 EWOULDBLOCK（或 EAGIN，与上面同）。

**select/poll/epoll_wait 函数的等待或超时时间**

select、poll、epoll_wait 函数的超时时间分别由传给各自函数的时间参数决定的，我们来看下这三个函数的签名：

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

三个函数最后一个参数是 timeout，只不过 select 函数的 timeout 参数的类型是一个结构体指针

select 函数的 timeout 参数含义有三种：

1. timeout 为 NULL 时，select 函数将一直阻塞下去，直到出错或者绑定其上的 socket 有事件；
2. 当 timeout->tv_sec 和 timeout->tv_usec 同时为 0 时，select 函数会检查一下绑定在其上的 socket 是否有事件，然后立刻返回；
3. 当 timeout->tv_sec 和 timeout->tv_usec 之和大于 0 时，select 函数检测到绑定其

上的 socket 有时间才会返回或者阻塞时长为 timeout->tv_sec + timeout->tv_usec 。

poll 和 epoll_wait 函数的超时时间为毫秒，设置为 0，和 select 函数一样，检测一下绑定其上的 socket 是否有事件，然后立即返回。

**为什么使用epoll时候需要将socket设置成阻塞的？**

首先epoll 模型通常用于服务端，那讨论的 socket 只有 listenfd 和 clientfd 了。

对于listenfd，可以阻塞可以非阻塞。有很多的服务器程序结构确实采用的就是阻塞的 listenfd，为了不让 accept 函数在没有连接时阻塞对程序其他逻辑执行流造成影响，我们通常将 accept 函数放在一个独立的线程中。但是如果不在一个独立线程中获得listenfd的话，如果默认一个监听socket是阻塞的话，有如下场景：客户端通过connect向服务器发起三次握手，三次握手后触发select或者epoll上的事件，但是呢此时客户端发送过来RST报文取消了连接，而此时服务器端调用了accept接收了次连接区去内核队列中取时内核队列中是空的（因为该客户端连接被取消），那么服务器就会阻塞在accept调用，无法响应其他就绪的监听socket，所以我们要吧监听socket即listenfd也设置为阻塞的。

对于clientfd，主要涉及到的是读写的问题。当读取的数据很小，比如有个buffer是1024字节，读取的数据小于1024的话，水平边缘触发搭配阻塞和非阻塞其实都一样。那么很多时候我们接受的数据都很大，一个buffer装不下，就需要while循环多次读。那么这种情况水平触发模式显得很鸡肋，阻塞和非阻塞效果都一样。所以我们主要讨论的是边缘触发情况下clientfd的阻塞和非阻塞。首先是ET+阻塞的情况，当最后一个数据读取完后，程序是无法立刻跳出while循环的，因为阻塞IO会在 while(true){ int len=recv(); }这里阻塞住，除非对方关闭连接或者recv出错，这样程序就无法继续往下执行，因为我就绪的文件描述符都在等着处理，这一次的epoll_wait没有办法处理其它的连接，会造成延迟、并发度下降。其次的话select或者epoll返回可读，和recv去读，这是两个独立的系统调用，两个操作之间是有窗口的，也就是说 select 返回可读，紧接着去 read，不能保证一定可读。man select中说了数据到达了但是可能error checksum或者discard了，如果你用阻塞的，就阻塞了进行不下去，这显然不行。如果是ET+非阻塞IO的话，当读取完数据后，recv会立即返回-1，并将errno设置为EAGAIN或EWOULDBLOCK,这就表示数据已经读取完成，已经没有数据了，可以退出循环了。这样就不会像阻塞IO一样卡在那里，这就减少了不必要的等待时间，性能自然更高。

# 用过哪些linux命令？

## top

top命令是Linux下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于Windows的任务管理器。

常用命令如下：

1. P：按%CPU使用率排行
2. M：按%MEM排行
3. T： 根据时间/累计时间进行排序。

## scp

用于不同linux主机之间复制文件的

```
scp  local_file    remote_username@remote_ip:remote_file
```

## find

找文件名

```
find . -name '[A-Z]*.txt' -print
```

## sar

`sar`（System Activity Reporter 系统活动情况报告）是目前 Linux 上最为全面的系统性能分析工具之一，可以从多方面对系统的活动进行报告，包括：文件的读写情况、系统调用的使用情况、磁盘 I/O、CPU 效率、内存使用状况、进程活动及 IPC 有关的活动等。我们可以使用`sar`命令来获得整个系统性能的报告。这有助于我们定位系统性能的瓶颈，并且有助于我们找出这些烦人的性能问题的解决方法。

参考

# tar

# df

用来检查linux服务器的文件系统的磁盘空间占用情况。可以利用该命令来获取硬盘被占用了多少空间，目前还剩下多少空间等信息。

# free

显示Linux系统中空闲的、已用的物理内存及swap内存,及被内核使用的buffer。在Linux系统监控的工具中，free命令是最经常使用的命令之一。

total:总计物理内存的大小。

used:已使用多大。

free:可用有多少。

# netstat

Netstat 命令用于显示各种网络相关信息，如网络连接，路由表，实际的网络连接以及每一个网络接口设备的状态信息。Netstat用于显示与IP、TCP、UDP和ICMP协议相关的统计数据，一般用于检验本机各端口的网络连接情况。

- 直接使用netstat
  输出结果可以分为两个部分：一个是Active Internet connections，称为有源TCP连接，其中"Recv-Q"和"Send-Q"指%0A的是接收队列和发送队列。这些数字一般都应该是0。如果不是则表示软件包正在队列中堆积。这种情况只能在非常少的情况见到。另一个是Active UNIX domain sockets，称为有源Unix域套接口(和网络套接字一样，但是只能用于本机通信，性能可以提高一倍)。
  Proto显示连接使用的协议,RefCnt表示连接到本套接口上的进程号,Types显示套接口的类型,State显示套接口当前的状态,Path表示连接到套接口的其它进程使用的路径名。

- 列出所有端口 #netstat -a
- 列出所有 tcp 端口 #netstat -at
- 列出所有 udp 端口 #netstat -au

## traceroute

追踪网络数据包的路由途径，预设数据包大小是40Bytes。

## route

跟路由表相关的命令

可以查看（route），增加（route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0），删除（route del -net 224.0.0.0 netmask 240.0.0.0）

## ip

linux的ip命令和ifconfig类似，但前者功能更强大，并旨在取代后者。使用ip命令，只需一个命令，你就能很轻松地执行一些网络管理任务。ifconfig是net-tools中已被废弃使用的一个命令，许多年前就已经没有维护了。iproute2套件里提供了许多增强功能的命令，ip命令即是其中之一。

# Linux惊群效应详解

参考链接

**定义：**惊群效应（thundering herd）是指多进程（多线程）在同时阻塞等待同一个事件的时候（休眠状态），如果等待的这个事件发生，那么他就会唤醒等待的所有进程（或者线程），但是最终却只能有一个进程（线程）获得这个时间的“控制权”，对该事件进行处理，而其他进程（线程）获取“控制权”失败，只能重新进入休眠状态，这种现象和性能浪费就叫做惊群效应。

**危害**：

1. Linux 内核对用户进程（线程）频繁地做无效的调度、上下文切换等使系统性能大

打折扣。上下文切换（context switch）过高会导致 cpu 像个搬运工，频繁地在寄存器和运行队列之间奔波，更多的时间花在了进程（线程）切换，而不是在真正工作的进程（线程）上面。直接的消耗包括 cpu 寄存器要保存和加载（例如程序计数器）、系统调度器的代码需要执行。间接的消耗在于多核 cache 之间的共享数据。

2. 为了确保只有一个进程（线程）得到资源，需要对资源操作进行加锁保护，加大了系统的开销。目前一些常见的服务器软件有的是通过锁机制解决的，比如 Nginx（它的锁机制是默认开启的，可以关闭）；还有些认为惊群对系统性能影响不大，没有去处理，比如 lighttpd

**Linux 解决方案之 Accept**

Linux 2.6 版本之前，监听同一个 socket 的进程会挂在同一个等待队列上，当请求到来时，会唤醒所有等待的进程。Linux 2.6 版本之后，通过引入一个标记位 WQ_FLAG_EXCLUSIVE，解决掉了 Accept 惊群效应。

# pthread_cond_wait 为什么需要传递 mutex 参数？

本质上这个问题想问这个锁是用来保护什么的？其实这个互斥锁不是用来保护条件变量的内部状态的，而是用来保护外部条件的，就是那个while()循环中的判断。

游双书里面说"pthread_cond_wait函数用于等待目标条件变量，mutex是保护条件变量的互斥锁，以确保pthread_cond_wait的原子性。"在看完这个回答你就该知道，**pthread_cond_wait的原子性指的是while条件判断成立和这个线程调用wait函数进入唤醒队列的原子性。**

- 错误写法

```
//threadA
pthread_mutex_lock(&mutex);
while (false == ready) {
   pthread_cond_wait(&cond, &mutex);
}
pthread_mutex_unlock(&mutex);

//threadB
ready = true;
pthread_cond_signal(&cond);
```

这个写法为什么有错误？如图：

| 执行序列 | Thread A | Thread B |
| --- | --- | --- |
| 1 | 1: pthread_mutex_lock(&mutex); | |
| 2 | 2: while (false == ready) { | |
| 3 | | 1: ready = true; |
| 4 | | 2: pthread_cond_signal(&cond); |
| 5 | 3: pthread_cond_wait(&cond, &mutex); | |
| 6 | 4: } | |



ThreadA进入while循环后，准备进入唤醒队列，此时ThreadB进来横插一脚，把`ready`改成`true`然后提前唤醒，这个时候线程A还没有进入唤醒队列，因此A会丢失唤醒条件进而永久wait

- 正确写法

```
//Thread A
pthread_mutex_lock(&mutex);
while (false == ready) {
    pthread_cond_wait(&cond, &mutex);
}
pthread_mutex_unlock(&mutex);

//Thread B
pthread_mutex_lock(&mutex);
ready = true;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond);
```

参考链接

# 什么是虚假唤醒？

一般来说我们要在等待wait的时候用while循环。因为会产生虚假唤醒。

pthread 的条件变量等待 `pthread_cond_wait` 是使用阻塞的系统调用实现的（比如 Linux 上的 `futex`），这些阻塞的系统调用在进程被信号中断后，通常会中止阻塞、直接返回 EINTR 错误。同样是阻塞系统调用，你从 `read` 拿到 EINTR 错误后可以直接决定重试，因为这通常不影响它本身的语义。而条件变量等待则不能，因为本线程拿到 EINTR 错误和重新调用 `futex` 等待之间，可能别的线程已经通过 `pthread_cond_signal` 或者 `pthread_cond_broadcast` 发过通知了。所以，虚假唤醒的一个可能性是条件变量的等待被信号中断。

在多核处理器下，pthread_cond_signal可能会激活多于一个线程（阻塞在条件变量上的线程）。结果是，当一个线程调用pthread_cond_signal()后，因为多个线程都被唤醒了，很可能其中一个唤醒的线程，先一步改变的condition. 此时另一个线程的condition已经不满足，因此需要加Where再次判断。这种效应成为”**虚假唤醒**”(spurious wakeup)。所以我们把判断条件从if改成while，pthread_cond_wait中的while()不仅仅在等待条件变量**前**检查条件变量，实际上在等待条件变量**后**也检查条件变量。

# 套接字文件描述符和端口号的关系

个socket句柄代表两个地址对 “本地ip:port”--“远程ip:port”

在windows下叫句柄，在linux下叫文件描述符

socket为内核对象，由操作系统内核来维护其缓冲区，引用计数，并且可以在多个进程中使用。

# 进程线程协程

参考链接

# linux栈大小

用命令ulimit -s查看

8M左右

# 服务器集群

# Linux进程、进程组、会话、僵尸

> 研究 Linux 之前，首先要对进程、进程组、会话，线程有个整体的了解：一个会话包含多个进程组，一个进程组包含多个进程，一个进程包含多个线程。

**进程**：

进程是 Linux 操作系统环境的基础，它控制着系统上几乎所有的活动。每个进程都有自己唯一的标识：进程 ID，也有自己的生命周期。进程都有父进程，父进程也有父进程，从而形成了一个以 init 进程 （PID = 1）为根的家族树。除此以外，进程还有其他层次关系：进程组和会话。

有几种比较特殊的进程：

1. 孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程(进程号为1)所收养，并由 init 进程对它们完成状态收集工作。
2. 僵尸进程：一个进程使用 fork 创建子进程，如果子进程先退出，而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。
3. 守护进程：（英语：daemon）是一种在后台执行的程序。此类程序会被以**进程**的形式初始化。**守护进程**程序的名称通常以字母“d”结尾：例如，syslogd 就是指管理系统日志的守护进程。

**进程ID**：

Linux每个进程都会有一个非负整数表示的唯一进程 ID，简称 pid。Linux 提供了getpid 函数来获

取进程的 pid，同时还提供了 getppid 函数来获取父进程的 pid。

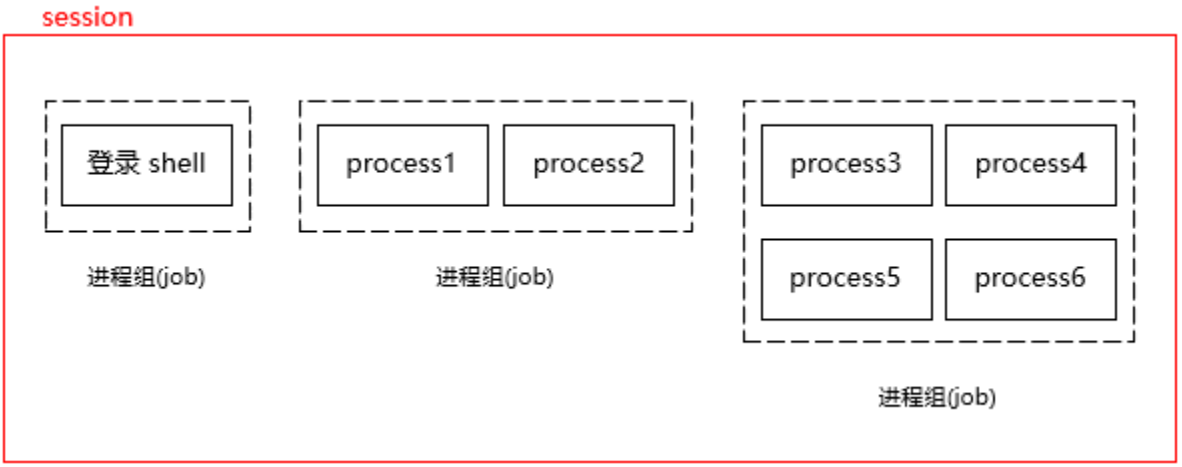
**进程组**：

进程组的概念并不难理解，可以将人与人之间的关系做类比。一起工作的同事，自然比毫不相干的路人更加亲近。shell 中协同工作的进程属于同一个进程组，就如同协同工作的人属于同一个部门一样。引入了进程组的概念，可以更方便地管理这一组进程了。比如这项工作放弃了，不必向每个进程一一发送信号，可以直接将信号发送给进程组，进程组内的所有进程都会收到该信号。

**会话(session)**：

一般是指 shell session。Shell session 是终端中当前的状态，在终端中只能有一个 session。当我们打开一个新的终端时，总会创建一个新的 shell session。

就进程间的关系来说，session 由一个或多个进程组组成。我们可以通过下图来理解进程、进程组和 session 之间的关系：




# 守护进程

**概念**：

守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程是一种很有用的进程。 Linux的大多数服务器就是用守护进程实现的。比如，Internet服务器inetd，Web服务器httpd等。同时，守护进程完成许多系统任务。比如，作业规划进程crond，打印进程lpd等。守护进程的编程本身并不复杂，复杂的是各种版本的Unix的实现机制不尽相同，造成不同 Unix环境下守护进程的编程规则并不一致。

当我们在命令行提示符后输入类似./helloworld程序时，在程序运行时终端被占用，此时无法执行其它操作。即使使用./helloworld &方式后台运行，当连接终端的网络出现问题，那么也会导致运行程序中断。这些因素对于长期运行的服务来说很不友好，而「守护进程」可以很好的解决这个问题。

**特性**：

守护进程最重要的特性是后台运行。其次，守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符，控制终端，会话和进程组，工作目录以及文件创建掩模等。这些环境通常是守护进程从执行它的父进程（特别是shell）中继承下来的。最后，守护进程的启动方式有其特殊之处。它可以在Linux系统启动时从启动脚本/etc/rc.d中启动，可以由作业规划进程crond启动，还可以由用户终端（通常是 shell）执行。总之，除开这些特殊性以外，守护进程与普通进程基本上没有什么区别。因此，编写守护进程实际上是把一个普通进程按照上述的守护进程的特性改造成为守护进程。如果对进程有比较深入的认识就更容易理解和编程了。

**编程**：

setsid()函数主要是重新创建一个session,子进程从父进程继承了SessionID、进程组ID和打开的终端,子进程如果要脱离父进程，不受父进程控制，我们可以用这个setsid命令。想脱离父进程，自己自由自在的活着，就要用这个命令执行后面的操作，简单粗暴。

编写守护进程过程如下：

1. 创建子进程，父进程退出
   进程 fork 后，父进程退出。这么做的原因有 2 点：
   - 如果守护进程是通过 Shell 启动，父进程退出，Shell 就会认为任务执行完毕，这时子进程由 init 收养
   - 子进程继承父进程的进程组 ID，保证了子进程不是进程组组长，因为后边调用setsid()要求必须不是进程组长
2. 子进程创建新会话

调用setsid()创建一个新的会话，并成为新会话组长。这个步骤主要是要与继承父进程的会话、进程组、终端脱离关系。fork()的目的是想成功调用setsid()来建 立新会话，子进程有单独的sid，并且成为新进程组的组长，不关联任何终端。

3. 禁止子进程重新打开终端
   此刻子进程是会话组长，为了防止子进程重新打开终端，再次 fork 后退出父进程，也就是此子进程。这时子进程 2 不再是会话组长，无法再打开终端。其实这一步骤不是必须的，不过加上这一步骤会显得更加严谨。
4. 设置当前目录为根目录
   如果守护进程的当前工作目录是/usr/home目录，那么管理员在卸载/usr分区时会报错的。为了避免这个问题，可以调用chdir()函数将工作目录设置为根目录/。
5. 设置文件权限掩码

文件权限掩码是指屏蔽掉文件权限中的对应位。由于使用 fork()函数新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多的麻烦。因此，把文件权限掩码设置为 0，可以大大增强该守护进程的灵活性。通常使用方法是 umask(0)。

6. 关闭文件描述符

   子进程会继承已经打开的文件，它们占用系统资源，且可能导致所在文件系统无法卸载。此时守护进程与终端脱离，常说的输入、输出、错误描述符也应该关闭。

```
pid_t pid, sid;
int i;
pid = fork(); // 第1步
if (pid < 0)
    exit(-1);
else if (pid > 0)
    exit(0);  // 父进程第一次退出

if ((sid = setsid()) < 0)  // 第2步
{
    syslog(LOG_ERR, "%s\n", "setsid");
    exit(-1);
}

// 第3步 第二次父进程退出
if ((pid = fork()) > 0)
    exit(0);
if ((sid = chdir("/")) < 0) // 第4步
{
    syslog(LOG_ERR, "%s\n", "chdir");
    exit(-1);
}

umask(0); // 第5步

// 第6步：关闭继承的文件描述符
for(i = 0; i < getdtablesize(); i++)
{
    close(i);
}
while(1)
{
    do_something();
}
closelog();
exit(0);
```

# 系统调用的详细过程

**系统调用：**系统调用是操作系统为用户提供的一系列API；系统调用将用户的请求发给内核，内核执行完以后，将结果返回给用户；

参考