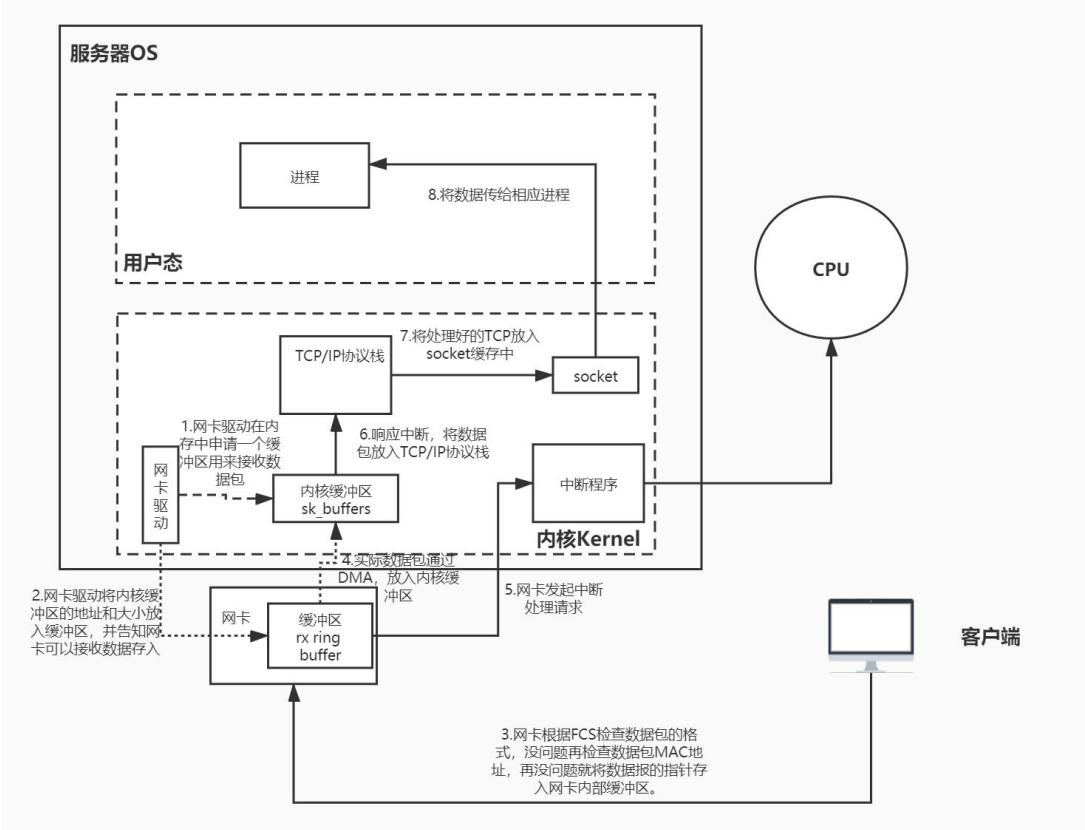
目录

I/O 总结

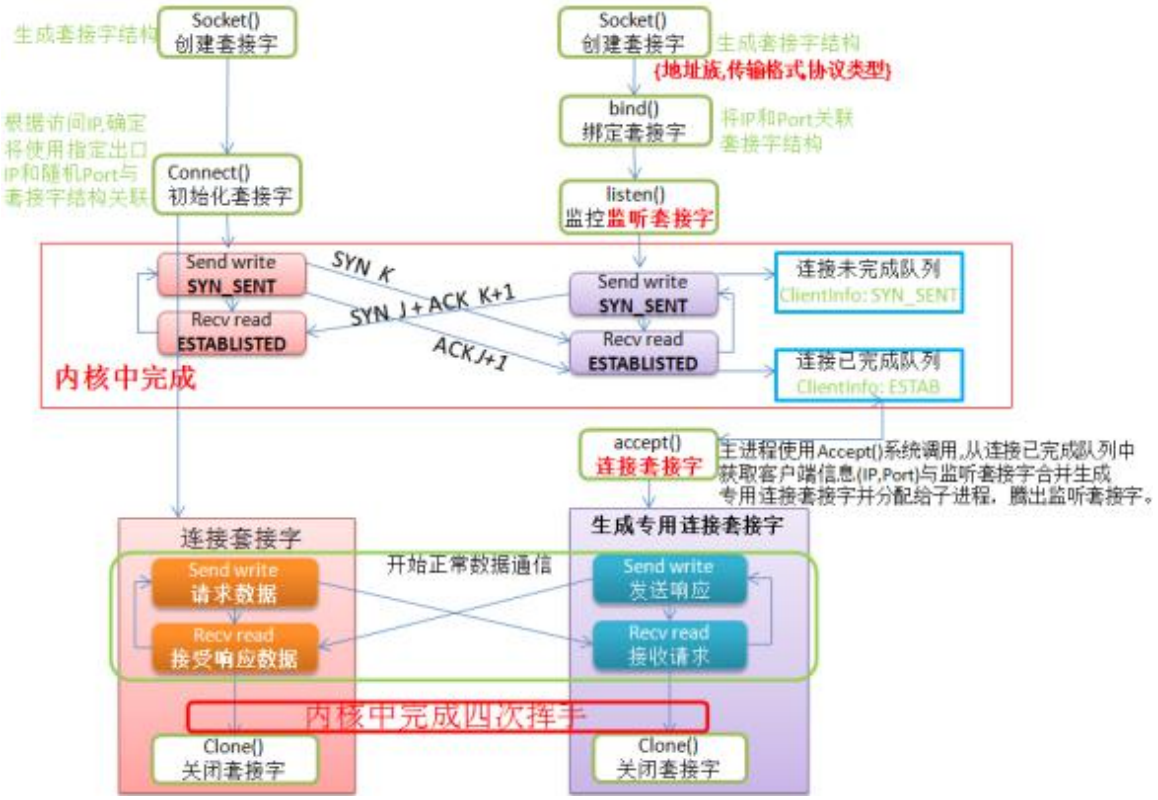
I/O 设备称为输入/输出设备，计算机发送数据是输出，接收数据是输入。
操作系统分为内核态和用户态两个空间。在操作系统的模型中，直接访问硬件的是内核（Kernel）。
一个服务端接收客户端的请求如下图所示：



1. 当数据包存入内核缓冲区时，会触发中断，这里我理解的是硬中断，是网卡发送的中断信号。

2. 上图仅表示已经创建好连接，然后互传信息的过程。

创建套接字的过程伴随着 TCP 的连接过程：



在连接过程中，其实就是 write 和 read 的过程。

对于一次 IO 访问，数据会先被拷贝到内核缓冲区，然后才从缓冲区拷贝到应用程序的地址空间。因此，一般有下面两个阶段：

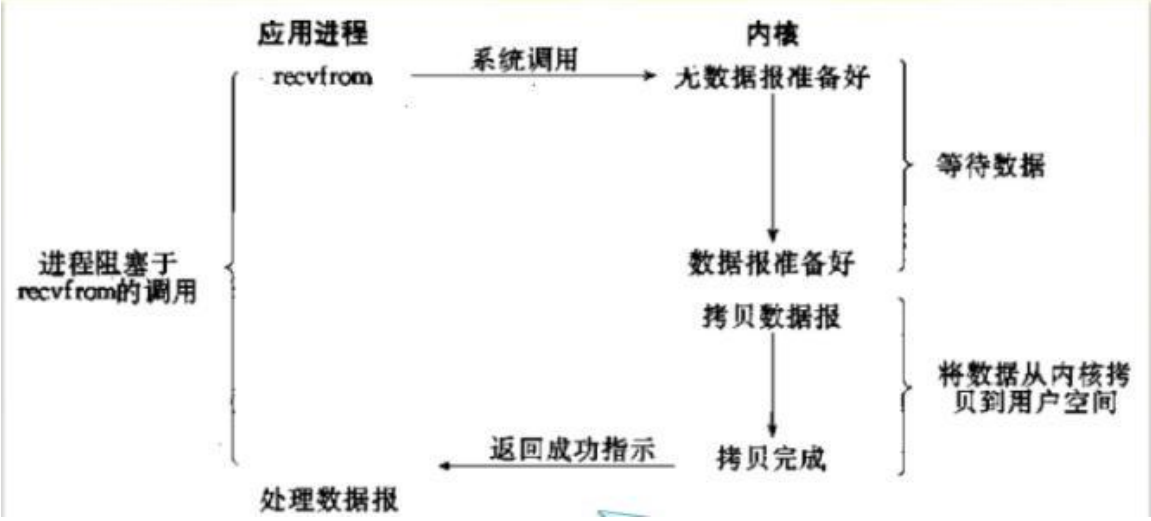
(1) 等待数据准备

(2) 将数据从内核拷贝到进程

因为这两个阶段，linux 产生了下面五种网络模式。

- 阻塞 I/O
- 非阻塞 I/O
- I/O 多路复用
- 信号驱动 I/O
- 异步 I/O

一、阻塞 I/O



在 linux 中，默认所有的 socket 都是阻塞的。

- 当用户进程连接完毕开始接收数据时，如果数据还没有拷贝到内核缓冲区，或内核缓冲区还没有足够的数据时，内核的后续操作会被阻塞。
- 而用户进程这边，在数据拷贝到用户内存的阶段（接收数据），则会一直阻塞，直到操作系统将内核缓冲区的数据拷贝到用户内存中，才回解除阻塞。

因此实际上，在 I/O 操作的两个阶段，都有阻塞。

缺点:每当有一个客户端请求时，服务端都会创建一个新的线程。在高并发环境下，线程会特别多（因为有很多客户端请求，创建了大量线程），这导致多个线程抢占 CPU 资源，引起 CPU 的上下文切换，增加了系统开销。

二、非阻塞 I/O

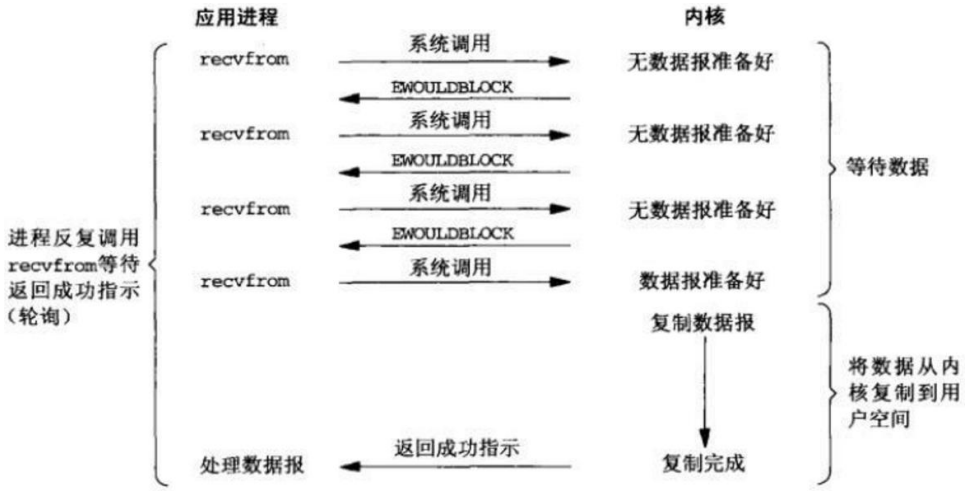


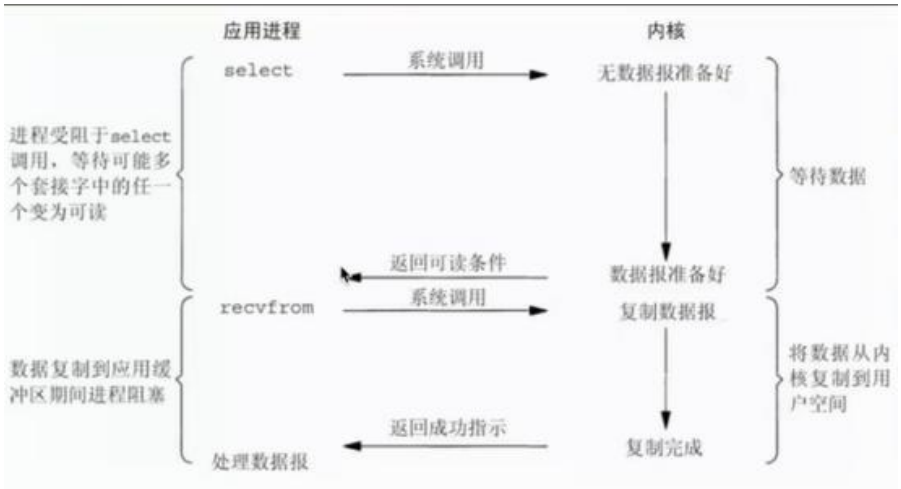
图6-2　非阻塞式I/O模型

Linux 下，可以通过设置 socket 使其变为 non-blocking。

- 当用户进程发出 read 操作时，如果内核中的数据还没准备好，它并不会阻塞住用户进程，而是返回一个 error。从用户进程的角度讲，它并没有等待，而是返回了一个结果。当用户进程发现返回的是 error 后，就再次发送 read 操作。如果内核中的数据准备好了，并且收到了 read 指令，就会将数据拷贝到用户内存。
- 因此，阻塞和非阻塞的主要区别就在于，调用阻塞 IO 会一直阻塞住对应进程直到操作完成，而非阻塞 IO 在内核还在准备数据的时候就会返回一个值，让用户进程不断轮询。
- 相同点在于，内核在拷贝数据到用户空间这个过程，进程始终是在阻塞的。
- 非阻塞 IO 解决了阻塞 IO 一个连接创建一个线程的缺点。在高并发下节省了创建线程的成本。

缺点：在轮询的过程中，会占用大量的 CPU 时间。（这个原因是我自己理解的，查了很多资料也没查到缺点，硬要说缺点的话可能这个算）

三、I/O 多路复用



IO 多路复用就是 select，poll，epoll。select/poll 的好处在于单个进程就可以处理多个网络连接的 IO。基本原理就是 select，poll，epoll 这个 function 会不断轮询所有的 socket，当某个 socket 有数据到了，就会通知用户进程。

当用户进程调用了 select 时，整个进程会被阻塞。内核会监视 select 负责的 socket，一旦有 socket 的数据准备好了，select 就会返回，进程解除阻塞，调用 read 将数据拷贝到用户空间。

这个 IO 多路复用使用 select 其实跟阻塞 IO 差不多，区别在于：

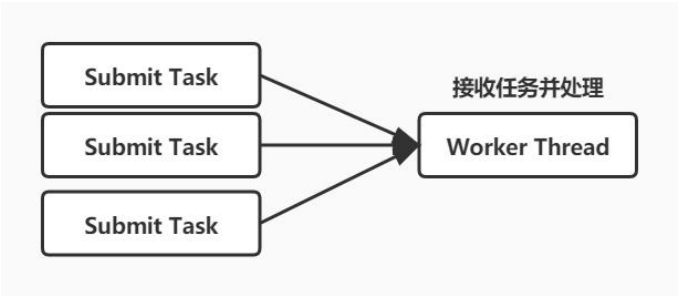
- IO 多路复用可以处理多个连接，而阻塞 IO，一个线程只能处理一个连接。
- 而进程则是一直被阻塞，只不过是被 select 这个函数阻塞。

select，poll，epoll 都是 IO 多路复用的机制。IO 多路复用都是通过这种机制，让内核去监视多个 socket，如果有 socket 完成，就通知进程进行相应的 write/read 指令。但 select，poll，epoll 本质上都是同步 IO，因为他们都需要在读写时间就绪后（socket 就绪）后再负责读写，也就是说，在就绪的这段时间里，进程始终是堵塞的。而异步 IO，只需要进程发出一个 IO 操作指令（write/read），内核就会自己去等待数据进内核缓冲区，再将数据拷贝到用户内存。
所以无论是阻塞还是非阻塞还是多路复用，在内核将数据拷贝到用户空间这个过程，进程始终是处于阻塞状态下的。不同在于第一步数据拷贝到内核缓冲区的过程：

- 阻塞 IO 模型下，进程是被阻塞的。
- 非阻塞 IO 模型下，进程一直轮询，没有被阻塞。
- IO 多路复用模型下，进程被 select 函数阻塞。

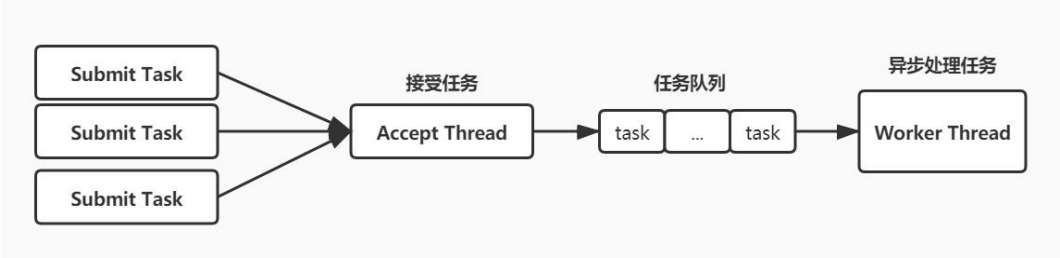
四、模型演变

以基本任务的处理流程为例，首先是串行模型



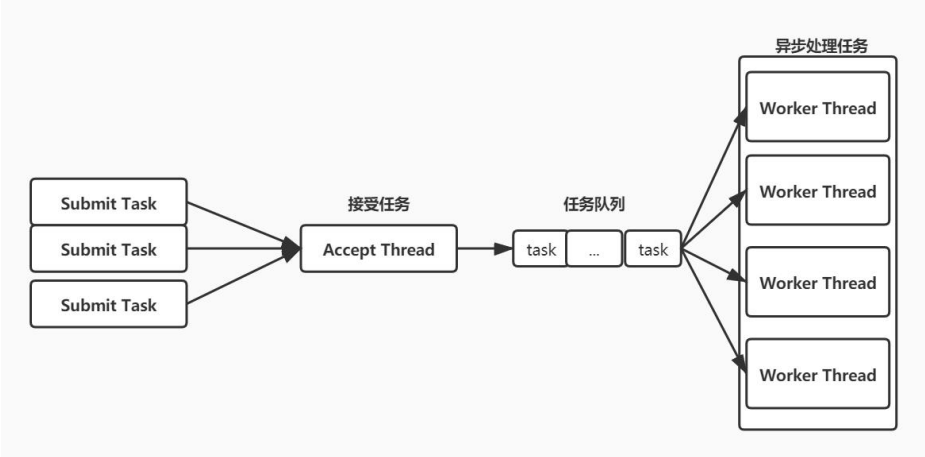
可以看到，每次提交任务时，工作线程会接收该任务并做处理。这种模式的缺点显而易见，在没做完前一个任务之前，也不会接收下一个任务，浪费 CPU 资源。

针对串行模式的缺点，将任务的接收和处理变成两个线程去处理：



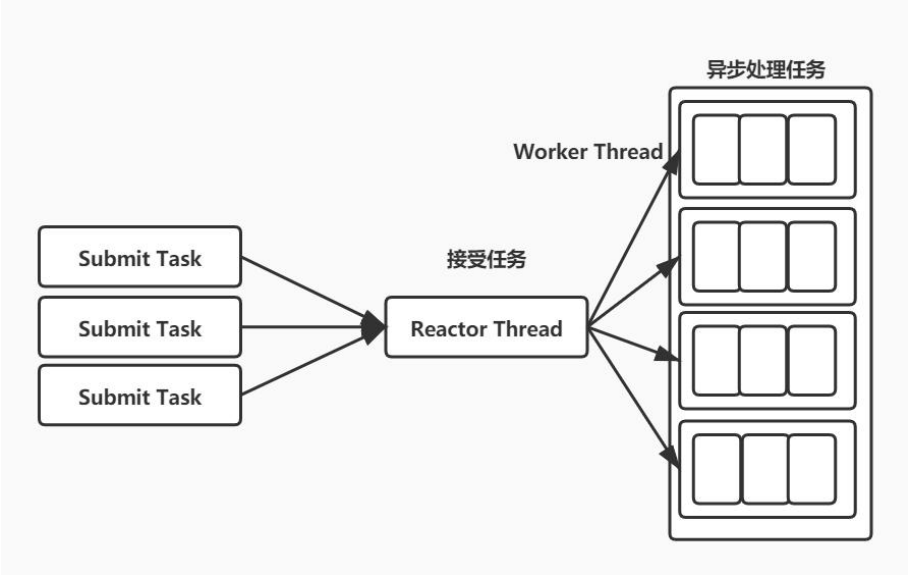
接收线程负责接收，工作线程负责处理，中间维护一个队列用来存放任务。这种模式的缺点，如果接收任务过多，而处理任务过慢，就导致接收任务队列的任务数量会越来越大。

针对上面模式的缺点，将处理任务改成并行，多个线程去处理任务队列的任务：



这种任务模式的缺点在于，由于是多个线程去操作任务队列，导致任务队列必须加锁来保证多线程环境下取数据的正确性。

于是，下面一种 Reactor 模型就出现了:



直接将任务分发给工作线程，每个工作线程自己维护一个任务队列，这种模式避免了锁的竞争。