1. 概念理解

在进行网络编程时，我们常常见到同步(Sync)/异步(Async)，阻塞(Block)/非阻塞(Unblock)四种调用方式：

同步/异步主要针对 C 端:

同步：

所谓同步，就是在 c 端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。**也就是必须一件一件事做**,等前一件做完了才能做下一件事。

例如普通 B/S 模式（同步）：提交请求->等待服务器处理->处理完毕返回 这个期间客户端浏览器不能干任何事

异步：

异步的概念和同步相对。当 c 端一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

例如 ajax 请求（异步）: 请求通过事件触发->服务器处理（这是浏览器仍然可以作其他事情）->处理完毕

阻塞/非阻塞主要针对 S 端:

阻塞

阻塞调用是指调用结果返回之前，当前线程会被挂起（线程进入非可执行状态，在这个状态下，cpu 不会给线程分配时间片，即线程暂停运行）。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。例如，我们在 socket 中调用 recv 函数，如果缓冲区中没有数据，这个函数就会一直等待，直到有数据才返回。而此时，当前线程还会继续处理各种各样的消息。

快递的例子：比如到你某个时候到 A 楼一层（假如是内核缓冲区）取快递，但是你不知道快递什么时候过来，你又不能干别的事，只能死等着。但你可以睡觉（进程处于休眠状态），因为你知道快递把货送来时一定会给你打个电话（假定一定能叫醒你）。

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。

还是等快递的例子：如果用忙轮询的方法，每隔 5 分钟到 A 楼一层(内核缓冲区）去看快递来了没有。如果没来，立即返回。而快递来了，就放在 A 楼一层，等你去取。

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。

1. 同步，就是我客户端（c 端调用者）调用一个功能，该功能没有结束前，我（c 端调用者）死等结果。
2. 异步，就是我（c 端调用者）调用一个功能，不需要知道该功能结果，该功能有结果后通知我（c 端调用者）即回调通知。

同步/异步主要针对 C 端，但是跟 S 端不是完全没有关系，同步/异步机制必须 S 端配合才能实现.同步/异步是由 c 端自己控制,但是 S 端是否阻塞/非阻塞，C 端完全不需要关心.

3. 阻塞，　　就是调用我（s 端被调用者，函数），我（s 端被调用者，函数）没有接收完数据或者没有得到结果之前，我不会返回。
4. 非阻塞，　就是调用我（s 端被调用者，函数），我（s 端被调用者，函数）立即返回，通过 select 通知调用者

同步 IO 和异步 IO 的区别就在于：数据访问的时候进程是否阻塞！

阻塞 IO 和非阻塞 IO 的区别就在于：应用程序的调用是否立即返回！

同步和异步都只针对于本机 SOCKET 而言的。

同步和异步,阻塞和非阻塞,有些混用,其实它们完全不是一回事,而且它们修饰的对象也不相同。

阻塞和非阻塞是指当 server 端的进程访问的数据如果尚未就绪,进程是否需要等待,简单说这相当于函数内部的实现区别,也就是未就绪时是直接返回还是等待就绪;

而同步和异步是指 client 端访问数据的机制,同步一般指主动请求并等待 I/O 操作完毕的方式,当数据就绪后在读写的时候必须阻塞(区别就绪与读写二个阶段,同步的读写必须阻塞),异步则指主动请求数据后便可以继续处理其它任务,随后等待 I/O,操作完毕的通知,这可以使进程在数据读写时也不阻塞。(等待"通知")

Node.js 里面的描述：

[html] view plain copy print?

1. 线程在执行中如果遇到磁盘读写或网络通信（统称为 I/O 操作），通常要耗费较长的时间，这时操作系统会剥夺这个线程的 CPU 控制权，使其暂停执行，同时将资源让给其他的工作线程，这种线程调度方式称为 阻塞。当 I/O 操作完毕时，操作系统将这个线程的阻塞状态解除，恢复其对 CPU 的控制权，令其继续执行。这种 I/O 模式就是通常的同步式 I/O（Synchronous I/O）或阻塞式 I/O （Blocking I/O）。
2. 相应地，异步式 I/O （Asynchronous I/O）或非阻塞式 I/O （Non-blocking I/O）则针对所有 I/O 操作不采用阻塞的策略。当线程遇到 I/O 操作时，不会以阻塞的方式等待 I/O 操作的完成或数据的返回，而只是将 I/O 请求发送给操作系统，继续执行下一条语句。当操作系统完成 I/O 操作时，以事件的形式通知执行 I/O 操作的线程，线程会在特定时候处理这个事件。为了处理异步 I/O，线程必须有事件循环，不断地检查有没有未处理的事件，依次予以处理。阻塞模式下，一个线程只能处理一项任务，要想提高吞吐量必须通过多线程。而非阻塞模式下，一个线程永远在执行计算操作，这个线程所使用的 CPU 核心利用率永远是 100%，I/O 以事件的方式通知。在阻塞模式下，多线程往往能提高系统吞吐量，因为一个线程阻塞时还有其他线程在工作，多线程可以让 CPU 资源不被阻塞中的线程浪费。而在非阻塞模式下，线程不会被 I/O 阻塞，永远在利用 CPU。多线程带来的好处仅仅是在多核 CPU 的情况下利用更多的核，而 Node.js 的单线程也能带来同样的好处。这就是为什么 Node.js 使用了单线程、非阻塞的事件编程模式。

2. Linux 下的五种 I/O 模型

1)阻塞 I/O（ blocking I/O ）

2)非阻塞 I/O （ nonblocking I/O ）

3) I/O 复用(select 和 poll) （ I/O multiplexing ）

4)信号驱动 I/O （ signal driven I/O (SIGIO) ）

5)异步 I/O （ asynchronous I/O (the POSIX aio_functions) ）

前四种都是同步，只有最后一种才是异步 IO。

阻塞 I/O 模型：

简介：进程会一直阻塞，直到数据拷贝完成

应用程序调用一个 IO 函数，导致应用程序阻塞，等待数据准备好。 如果数据没有准备好，一直等待....数据准备好了，从内核拷贝到用户空间,IO 函数返回成功指示。

我们 第一次接触到的网络编程都是从 listen()、send()、recv()等接口开始的。使用这些接口可以很方便的构建服务器 /客户机的模型。

阻塞 I/O 模型图：在调用 recv()/recvfrom（）函数时，发生在内核中等待数据和复制数据的过程。

当调用 recv()函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从**系统缓冲区**复制到用户空间，然后该函数返回。在套接应用程序中，当调用 recv()函数时，未必用户空间就已经存在数据，那么此时 recv()函数就会处于等待状态。

当使用 socket()函数和 WSASocket()函数创建套接字时，默认的套接字都是阻塞的。这意味着当调用 Windows Sockets API 不能立即完成时，线程处于等待状态，直到操作完成。

并不是所有 Windows Sockets API 以阻塞套接字为参数调用都会发生阻塞。例如，以阻塞模式的套接字为参数调用 bind()、listen()函数时，函数会立即返回。将可能阻塞套接字的 Windows Sockets API 调用分为以下四种:

1．输入操作： recv()、recvfrom()、WSARecv()和 WSARecvfrom()函数。以阻塞套接字为参数调用该函数接收数据。如果此时套接字缓冲区内没有数据可读，则调用线程在数据到来前一直睡眠。

2．输出操作： send()、sendto()、WSASend()和 WSASendto()函数。以阻塞套接字为参数调用该函数发送数据。如果套接字缓冲区没有可用空间，线程会一直睡眠，直到有空间。

3．接受连接：accept()和 WSAAcept()函数。以阻塞套接字为参数调用该函数，等待接受对方的连接请求。如果此时没有连接请求，线程就会进入睡眠状态。

4．外出连接：connect()和 WSAConnect()函数。对于 TCP 连接，客户端以阻塞套接字为参数，调用该函数向服务器发起连接。该函数在收到服务器的应答前，不会返回。这意味着 TCP 连接总会等待至少到服务器的一次往返时间。

使用阻塞模式的套接字，开发网络程序比较简单，容易实现。当希望能够立即发送和接收数据，且处理的套接字数量比较少的情况下，使用阻塞模式来开发网络程序比较合适。

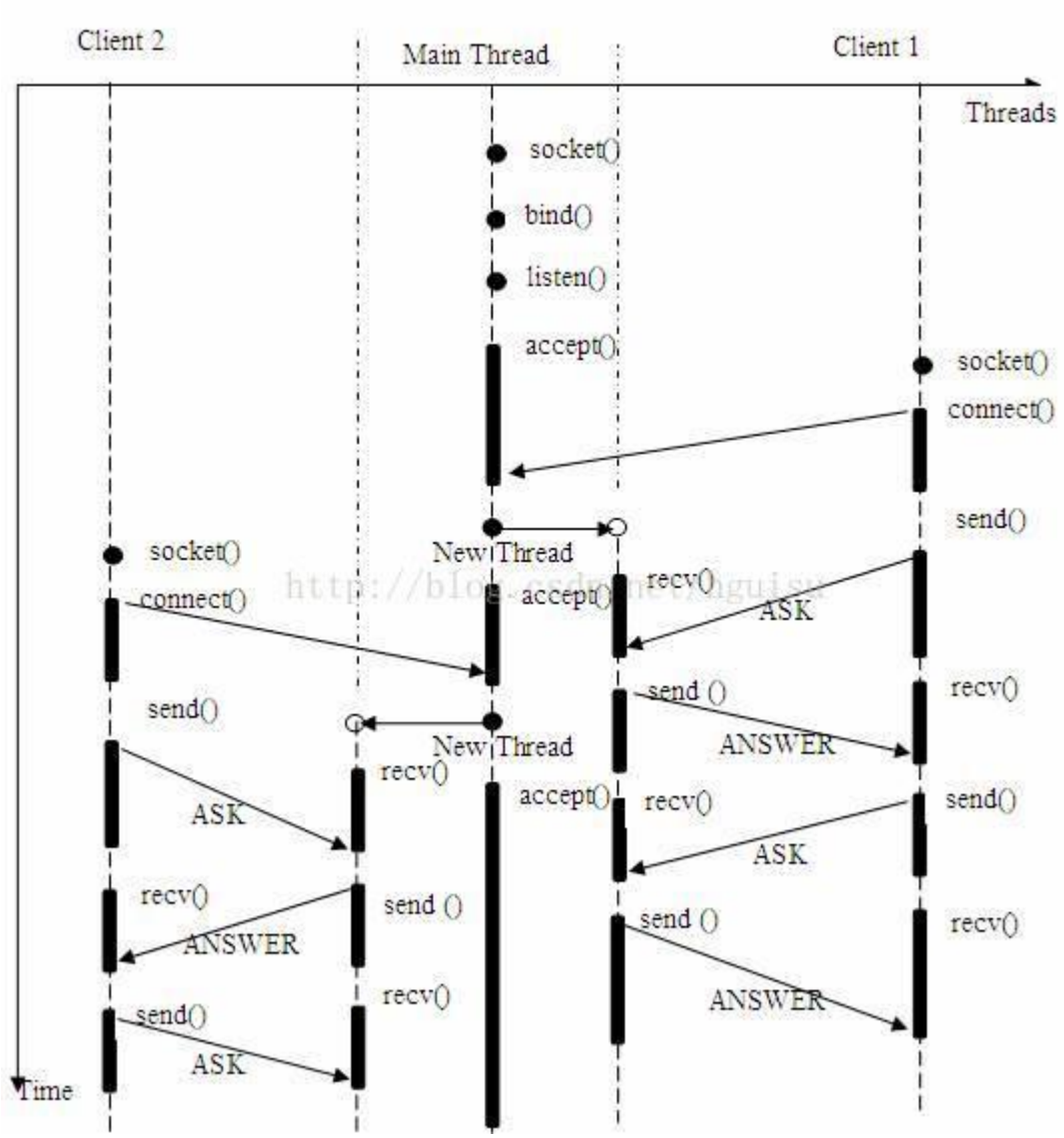
阻塞模式套接字的不足表现为，在大量建立好的套接字线程之间进行通信时比较困难。当使用“生产者-消费者”模型开发网络程序时，为每个套接字都分别分配一个读线程、一个处理数据线程和一个用于同步的事件，那么这样无疑加大系统的开销。其最大的缺点是当希望同时处理大量套接字时，将无从下手，其扩展性很差.

阻塞模式给网络编程带来了一个很大的问题，如在调用 send()的同时，线程将被阻塞，在此期间，线程将无法执行任何运算或响应任何的网络请求。这给多客户机、多业务逻辑的网络编程带来了挑战。这时，我们可能会选择多线程的方式来解决这个问题。

应对多客户机的网络应用，最简单的解决方式是在服务器端使用多线程（或多进程）。多线程（或多进程）的目的是让每个连接都拥有独立的线程（或进程），这样任何一个连接的阻塞都不会影响其他的连接。

具体使用多进程还是多线程，并没有一个特定的模式。传统意义上，进程的开销要远远大于线程，所以，如果需要同时为较多的客户机提供服务，则不推荐使用多进程；如果单个服务执行体需要消耗较多的 CPU 资源，譬如需要进行大规模或长时间的数据运算或文件访问，则进程较为安全。通常，使用 pthread_create () 创建新线程，fork() 创建新进程。

多线程/进程服务器同时为多个客户机提供应答服务。模型如下：



主线程持续等待客户端的连接请求，如果有连接，则创建新线程，并在新线程中提供为前例同样的问答服务。

上述多线程的服务器模型似乎完美的解决了为多个客户机提供问答服务的要求，但其实并不尽然。如果要同时响应成百上千路的连接请求，则无论多线程还是多进程都会严重占据系统资源，降低系统对外界响应效率，而线程与进程本身也更容易进入假死状态。

由此可能会考虑使用“**线程池**”或“**连接池**”。“线程池”旨在减少创建和销毁线程的频率，其维持一定合理数量的线程，并让空闲的线程重新承担新的执行任务。“连接池”维持连接的缓存池，尽量重用已有的连接、减少创建和关闭连接的频率。这两种技术都可以很好的降低系统开销，都被广泛应用很多大型系统，如 apache，**MySQL 数据库**等。

但是，“线程池”和“连接池”技术也只是在一定程度上缓解了频繁调用 IO 接口带来的资源占用。而且，所谓“池”始终有其上限，当请求大大超过上限时，“池”构成的系统对外界的响应并不比没有池的时候效果好多少。所以使用“池”必须考虑其面临的响应规模，并根据响应规模调整“池”的大小。

对应上例中的所面临的可能同时出现的上千甚至上万次的客户端请求，“线程池”或“连接池”或许可以缓解部分压力，但是不能解决所有问题。

非阻塞 IO 模型 ：

简介：非阻塞 IO 通过进程反复调用 IO 函数（多次系统调用，并马上返回）；在数据拷贝的过程中，进程是阻塞的；

我们把一个 SOCKET 接口设置为非阻塞就是告诉内核，当所请求的 I/O 操作无法完成时，**不要将进程睡眠**，而是返回一个错误。这样我们的 I/O 操作函数将不断的**测试**数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用 CPU 的时间。

把 SOCKET 设置为非阻塞模式，即通知系统内核：在调用 Windows Sockets API 时，不要让线程睡眠，而应该让函数立即返回。在返回时，该函数返回一个错误代码。图所示，一个非阻塞模式套接字多次调用 recv()函数的过程。前三次调用 recv()函数时，内核数据还没有准备好。因此，该函数立即返回 WSAEWOULDBLOCK 错误代码。第四次调用 recv()函数时，数据已经准备好，被复制到应用程序的缓冲区中，recv()函数返回成功指示，应用程序开始处理数据。

当使用 socket()函数和 WSASocket()函数创建套接字时，默认都是阻塞的。在创建套接字之后，通过调用 ioctlsocket()函数，将该套接字设置为非阻塞模式。Linux 下的函数是:fcntl().

套接字设置为非阻塞模式后，在调用 Windows Sockets API 函数时，调用函数会立即返回。大多数情况下，这些函数调用都会调用“失败”，并返回 WSAEWOULDBLOCK 错误代码。说明请求的操作在调用期间内没有时间完成。通常，应用程序需要重复调用该函数，直到获得成功返回代码。

需要说明的是并非所有的 Windows Sockets API 在非阻塞模式下调用，都会返回 WSAEWOULDBLOCK 错误。例如，以非阻塞模式的套接字为参数调用 bind()

函数时，就不会返回该错误代码。当然，在调用 WSAStartup()函数时更不会返回该错误代码，因为该函数是应用程序第一调用的函数，当然不会返回这样的错误代码。

要将套接字设置为非阻塞模式，除了使用 ioctlsocket()函数之外，还可以使用 WSAAsyncselect()和 WSAEventselect()函数。当调用该函数时，套接字会自动地设置为非阻塞方式。

由于使用非阻塞套接字在调用函数时，会经常返回 WSAEWOULDBLOCK 错误。所以在任何时候，都应仔细检查返回代码并作好对“失败”的准备。应用程序连续不断地调用这个函数，直到它返回成功指示为止。上面的程序清单中，在 While 循环体内不断地调用 recv()函数，以读入 1024 个字节的数据。这种做法很浪费系统资源。

要完成这样的操作，有人使用 MSG_PEEK 标志调用 recv()函数查看缓冲区中是否有数据可读。同样，这种方法也不好。因为该做法对系统造成的开销是很大的，并且应用程序至少要调用 recv()函数两次，才能实际地读入数据。较好的做法是，使用套接字的“I/O 模型”来判断非阻塞套接字是否可读可写。

非阻塞模式套接字与阻塞模式套接字相比，不容易使用。使用非阻塞模式套接字，需要编写更多的代码，以便在每个 Windows Sockets API 函数调用中，对收到的 WSAEWOULDBLOCK 错误进行处理。因此，非阻塞套接字便显得有些难于使用。

但是，非阻塞套接字在控制建立的多个连接，在数据的收发量不均，时间不定时，明显具有优势。这种套接字在使用上存在一定难度，但只要排除了这些困难，它在功能上还是非常强大的。通常情况下，可考虑使用套接字的“I/O 模型”，它有助于应用程序通过异步方式，同时对一个或多个套接字的通信加以管理。

IO 复用模型：

简介：主要是 select 和 epoll；对一个 IO 端口，两次调用，两次返回，比阻塞 IO 并没有什么优越性；关键是能实现同时对多个 IO 端口进行监听；

I/O 复用模型会用到 select、poll、epoll 函数，这几个函数也会使进程阻塞，但是和阻塞 I/O 所不同的的，这两个函数可以同时阻塞多个 I/O 操作。而且可以同时对多个读操

作，多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调用 I/O 操作函数。

信号驱动 IO

简介：两次调用，两次返回；

首先我们允许套接口进行信号驱动 I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个 SIGIO 信号，可以在信号处理函数中调用 I/O 操作函数处理数据。

异步 IO 模型

简介：数据拷贝的时候进程无需阻塞。

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作

同步 IO 引起进程阻塞，直至 IO 操作完成。
异步 IO 不会引起进程阻塞。
IO 复用是先通过 select 调用阻塞。

5 个 I/O 模型的比较：

3. select、poll、epoll 简介

select 原型说明:http://blog.csdn.NET/hguisu/article/details/38638183#t5

epoll 模型：http://blog.csdn.Net/hguisu/article/details/38638183#t12

epoll 跟 select 都能提供多路 I/O 复用的解决方案。在现在的 Linux 内核里有都能够支持，其中 epoll 是 Linux 所特有，而 select 则应该是 POSIX 所规定，一般操作系统均有实现

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。这样所带来的缺点是：

1、 单个进程可监视的 fd 数量被限制，即能监听端口的大小有限。

一般来说这个数目和系统内存关系很大，具体数目可以 cat /proc/sys/fs/file-max 察看。32 位机默认是 1024 个。64 位机默认是 2048.

2、 对 socket 进行扫描时是线性扫描，即采用轮询的方法，效率较低：

当套接字比较多的时候，每次 select()都要通过遍历 FD_SETSIZE 个 Socket 来完成调度,不管哪个 Socket 是活跃的,都遍历一遍。这会浪费很多 CPU 时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是 epoll 与 kqueue 做的。

3、需要维护一个用来存放大量 fd 的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

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：

1、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。 2
、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:

epoll 支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些 fd 刚刚变为就需态，并且只会通知一次。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

epoll 的优点：

1、没有最大并发连接的限制，能打开的 FD 的上限远大于 1024（1G 的内存上能监听约 10 万个端口）；

2、效率提升，不是轮询的方式，不会随着 FD 数目的增加效率下降。只有活跃可用的 FD 才会调用 callback 函数；

即 Epoll 最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll 的效率就会远远高于 select 和 poll。

3、内存拷贝，利用 mmap()文件映射内存加速与内核空间的消息传递；即 epoll 使用 mmap 减少复制开销。

select、poll、epoll 区别总结：

1、支持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

2、FD 剧增后带来的 IO 效率问题

select	因为每次调用时都会对连接进行线性遍历，所以随着 FD 的增加会造成遍历速度慢的“线性下降性能问题”。
poll	同上

epoll	因为 epoll 内核中实现是根据每个 fd 上的 callback 函数来实现的，只有活跃的 socket 才会主动调用 callback，所以在活跃 socket 较少的情况下，使用 epoll 没有前面两者的线性下降的性能问题，但是所有 socket 都很活跃的情况下，可能会有性能问题。

3、 消息传递方式

select	内核需要将消息传递到用户空间，都需要内核拷贝动作
poll	同上
epoll	epoll 通过内核和用户空间共享一块内存来实现的。

总结：

综上，在选择 select，poll，epoll 时要根据具体的使用场合以及这三种方式的自身特点。

1、表面上看 epoll 的性能最好，但是在连接数少并且连接都十分活跃的情况下，select 和 poll 的性能可能比 epoll 好，毕竟 epoll 的通知机制需要很多函数回调。

2、select 低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善

同步/异步与阻塞/非阻塞经常看到是成对出现：

同步阻塞,异步非阻塞,同步非阻塞