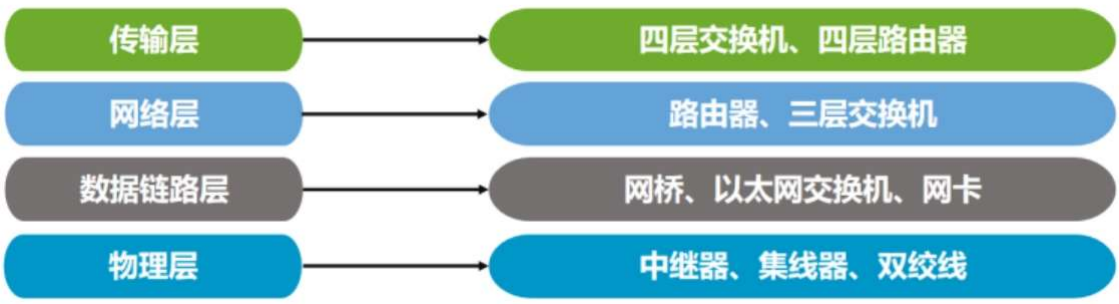
TCP/IP五层模型详解

左侧：TCP/IP五层或TCP/IP四层

右侧：从不同维度分为OSI七层

每层运行的常见设备如下图



1 物理层

物理层，物理层干啥的，就是电脑之间要联网，一般咋弄？类似于说，你有台电脑，现在要联网，咋联？以前N年前，大家记不记得都是在电脑上插根线是吧，然后才能上网，结果现在就是联个wifi就行了，还有中国美国之前联网靠的是海底的光缆。所以物理层就指的这个，就是怎么把各个电脑给联结起来，形成一个网络，这就是物理层的含义，物理层负责传输0和1的电路信号。学过一些计算机的同学，计算机的最最底层，就是0/1，电信号。

2 数据链路层

由于单纯的电平信号“0”和“1”没有任何意义，在实际应用中，我们会将电平信号进行分组处理，多少位一组、每组什么意思，这样数据才有具体的含义。数据链路层的功能就是定义电平信号的分组方式

2.1 以太网协议

数据链路层使用以太网协议进行数据传输，基于MAC地址的广播方式实现数据传输，只能在局域网内广播。早期各个公司都有自己的分组方式，后来形成了统一的标准，即以太网协议Ethernet

2.2 Ethernet

以太网由一组电平信号构成一个数据包，叫作“帧”，每一数据帧由报头Head和数Data两部分组成

Head：固定18字节，其中发送者/源地址6字节，接收者/目标地址6字节，数据类型6字节。

Data：最短46字节，最长1500字节。数据包的具体内容格式为：Head长度+Data长度=最短64字节，最长1518字节

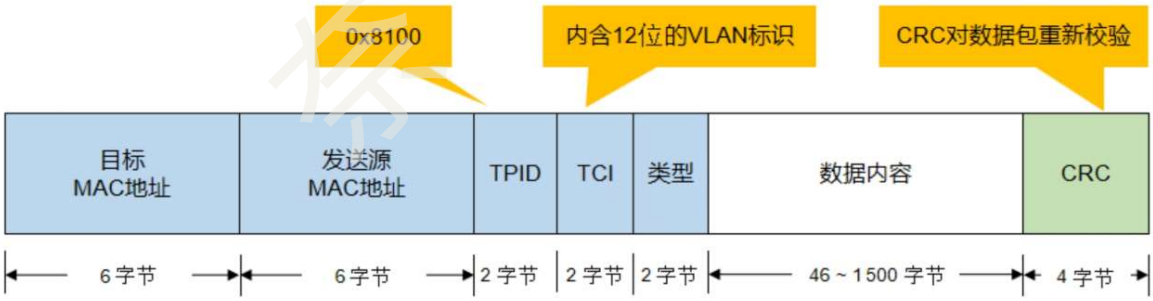
2.3 MAC地址

Head中包含的源地址和目标地址的由来：Ethernet规定接入Internet的设备必须配有网卡，发送端和接收端的地址便是指网卡的地址，即MAC地址。

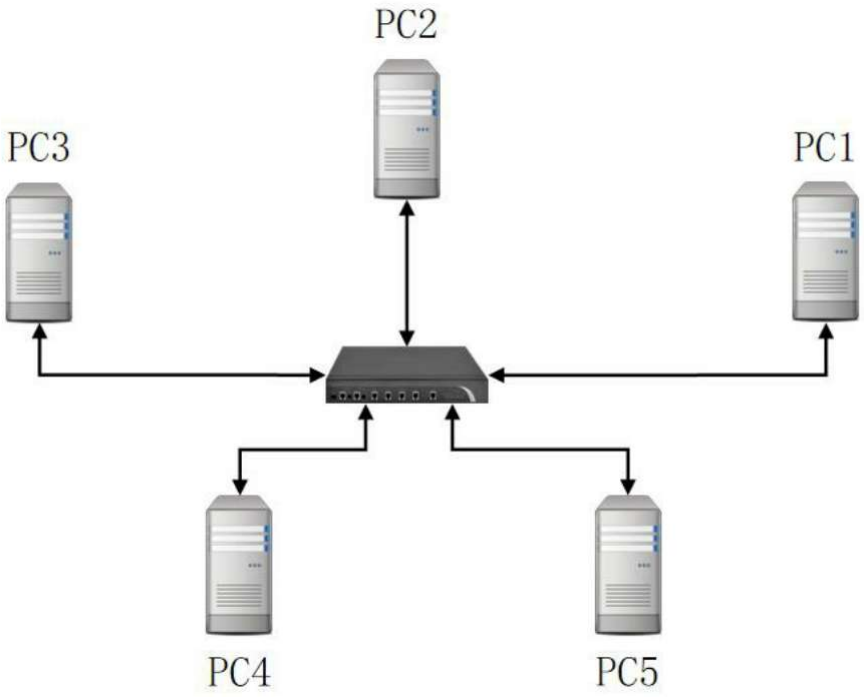
MAC地址：每块网卡出厂时都被印上一个世界唯一的MAC地址，它是一个长度为48位的二进制数，通常用12位十六进制数表示（前6位是厂商编号，后6位是流水线号）

2.4 Broadcast广播

有了MAC地址，同一网络内的两台主机就可以通信了（一台主机通过ARP协议获取另外一台主机的MAC地址），下面是以太网通信数据帧的详细示意图。



PC1按照固定协议格式以广播的方式发送以太网包给PC4，然而，PC2、PC3、PC5都会收到PC1发来的数据包，拆开后如果发现目标MAC地址不是自己就会丢弃，如果是自己就响应



3 网络层

如果所有的通信都采用以太网的广播方式，那么一台机器发送的数据包全世界都会收到，这就不仅仅是效率低的问题了，这会是一种灾难。

全世界的大网络由一个个小的彼此隔离的局域网组成，以太网包只能在一个局域网内发送，一个局域网是一个广播域，跨广播域通信只能通过路由转发。

由此得出结论：必须找出一种方法来区分哪些计算机属于同一广播域。如果是就采用广播的方式发送，如果不是就采用路由的方式发送（向不同广播域/子网分发数据包），MAC地址是无法区分的，网络层就是用来解决这一问题的。

3.1 IP

规定网络地址的协议叫作IP（Internet Protocol，网际互联协议），它定义的地址称为IP地址。广泛采用v4版本即IPv4，规定网络地址由32位二进制数表示。一个IP地址通常写成四段十进制数，例如172.16.10.1，其取值范围为：0.0.0.0～255.255.255.255

3.2．子网掩码

所谓“子网掩码”，就是表示子网络特征的一个参数。

根据“子网掩码”就能判断任意两个IP地址是否处于同一个子网络。方法是将两个IP地址与子网掩码分别进行&运算（两个数位都为1，运算结果为1，否则为0），然后比较结果是否相同，如果相同，就表明它们在同一个子网络中，否则就不在

3.3．IP数据包

IP数据包也分为Head和Data两部分，无须为IP数据包定义单独的栏位，直接放入以太网包的Data部分即可。Head（IP头部）：长度为20～60字节。Data（IP数据）：最长为65515字节。而以太网数据包的Data部分，最长只有1500字节。因此，如果IP数据包超过1500字节，它就需要分割成几个以太网数据包，分开发送。

3.4．ARP

计算机通信方式是广播的方式。所有上层的数据包到最后都要封装到以太网头，然后通过以太网协议发送。在谈及以太网协议的时候，我们已经了解到，通信基于MAC地址的广播方式实现的，计算机在发送数据包时，获取自身的MAC地址是容易的，获取目标主机的MAC地址，需要通过

ARP（Address Resolution Protocol，地址解析协议）来实现。

11000000 10101000 10011011 10010000
192.168.155.144

11111111 11111111 11111100 00000000
255.255.252.0

11000000 10101000 10011000 0000000

11000000 10101000 10011011 10010001
192.168.155.145

11111111 11111111 11111100 00000000
255.255.252.0

11000000 10101000 10011000 0000000

11000000 10101000 10011011 10010010
192.168.155.146

11111111 11111111 11111100 00000000
255.255.252.0

11000000 10101000 10011000 00000000

11000000 10101000 10011010 10010000
192.168.154.144

11111111 11111111 11111100 00000000
255.255.252.0

11000000 10101000 10011000 00000000

255.255.252.0

4 传输层

网络层的IP地址帮我们区分子网，以太网层的MAC地址帮我们找到主机

那么我们通过IP地址和MAC地址找到了一台特定的主机，如何标识这台主机上的应用程序?

端口就是应用程序与网卡关联的编号。那么传输层就是用来建立端口到端口的通信机制的

4.1 TCP

TCP（Transmission Control Protocol，传输控制协议）是一种可靠传输协议，TCP数据包没有长度限制，理论上可以无限长，但是为了保证网络的效率，通常TCP数据包的长度不会超过IP数据包的长度，以确保单个TCP数据包不必再分割

|以太网头部|IP头部|TCP头部|数据|

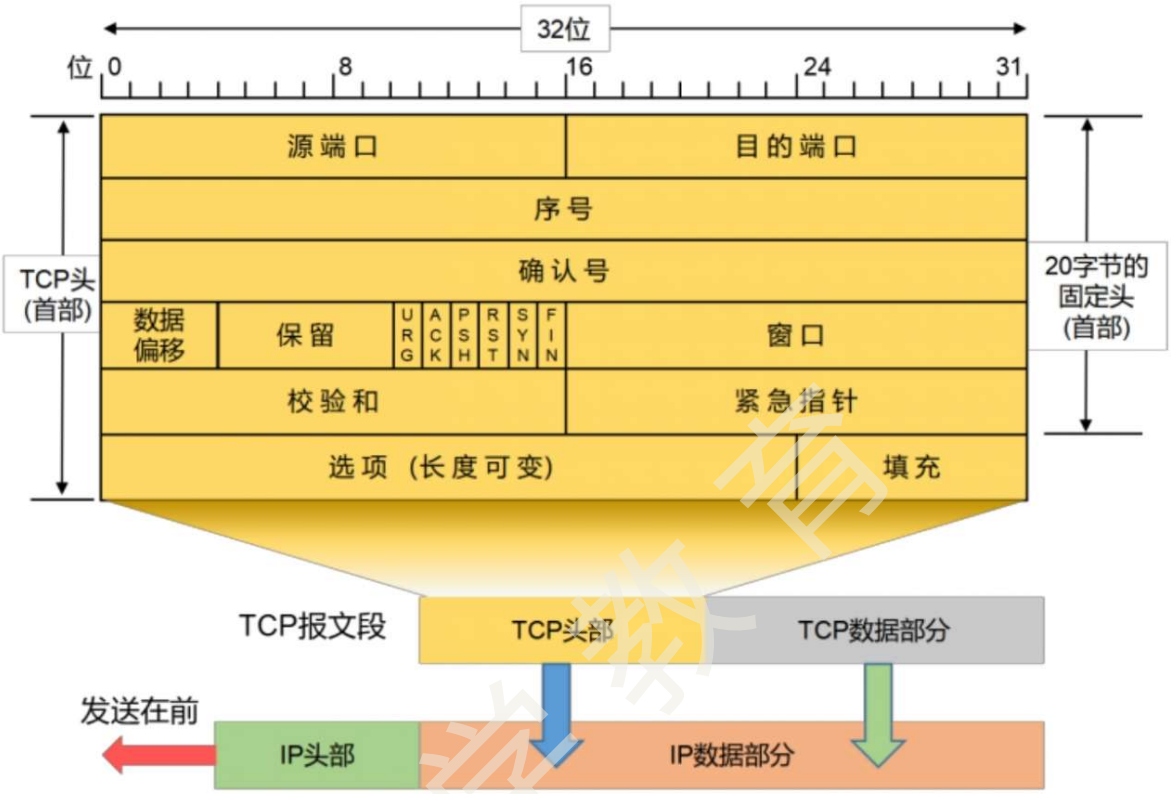
4.2 UDP

UDP（User Datagram Protocol，用户数据报协议）是一种不可靠传输协议，“报头”部分总共有8字节，总长度不超过65535字节，正好放进一个IP数据包

|以太网头部|IP头部|UTP头部|数据|

4.3 TCP

报文结构TCP报文是TCP层传输的数据单元，也叫作报文段



下面对报文内容做详细介绍。

（1）端口号：用来标识同一台计算机的不同应用进程。

● 源端口：源端口和IP地址的作用是标识报文的返回地址。

● 目的端口：目的端口指明接收方计算机上的应用程序接口。TCP报头中的源端口号和目的端口号同IP数据包中的源IP地址和目的IP地址唯一确定一条TCP连接。

> 端口从0-65535
>
> 常见的端口有

> HTTP（HyperText Transfer Protocol）超文本传送协议：80
>
> FTP（File Transfer Protocol）文件传输协议： 21
>
> SMTP（Simple Mail Transfer Protocol）简单邮件传送协议：25
>
> DNS（Domain Name System）域名系统：53

（2）序号和确认号：TCP可靠传输的关键部分。序号是本报文段发送的数据组的第一个字节的序号。在TCP传送的流中，每一个字节都有一个序号。例如：一个报文段的序号为300，此报文段数据部分共有100字节，则下一个报文段的序号为400。所以序号确保了TCP传输的有序性。

> 因为一个请求会被拆分成多个数据包，所以TCP需要序号来分段传输
>
> 如果当前序号为300，包含100个字节，那么下一个报文的序号就是400，以此保证TCP传输的有序性

==确认号，即ACK，指明下一个期待收到的字节序号，表明该序号之前的所有数据已经正确无误地收到。确认号只有当ACK标志为1时才有效。==比如建立连接时，SYN报文的ACK标志为0。

>

（3）数据偏移/头部长度：4位。由于头部可能含有可选项内容，TCP报头的长度是不确定的，报头不包含任何任选属性则长度为20字节，4位头部长度属性所能表示的最大值为1111，转化成十进制为15，15×32/8=60，故报头最大长度为

60字节。头部长度也叫数据偏移，是因为头部长度实际上指示了数据区在报文段中的起始偏移值。

（4）保留：为将来定义新的用途保留，现在一般设置为0。

（5）标志位：URG、ACK、PSH、RST、SYN、FIN，共6个，每一个标志位都表示一个控制功能，具体含义如下表所示。

字　段	中文名称	含　义
URG	紧急指针标志	为1表示紧急指针有效，为0则忽略紧急指针
ACK	确认序号标志	为1表示确认号有效，为0表示报文中不含确认信息，忽略确认号属性
PSH	接收信号标志	为1表示带有Push标志的数据，指示接收方在接收到该报文段以后，应尽快将这个报文段交给应用程序，而不是在缓冲区排队
RST	重置连接标志	用于重置由于主机崩溃或其他原因而出现错误的连接，或者用于拒绝非法的报文段和拒绝连接请求
SYN	同步序号标志	用于建立连接过程，在连接请求中，SYN=1和ACK=0表示该数据段没有使用捎带的确认域，而连接应答捎带一个确认，即SYN=1和ACK=1
FIN	完成标志	用于释放连接，为1表示发送方已经没有数据发送了，即关闭本方数据流

（6）窗口：滑动窗口大小，用来告知发送端接收端的缓存大小，以此控制发送端发送数据的速率，从而达到流量控制。窗口大小是一个16位属性，因而窗口大小最大为65535。

（7）校验和：奇偶校验，此校验和针对整个TCP报文段，包括TCP头部和TCP数据，以16位属性进行计算所得。由发送端计算和存储，并由接收端进行验证。

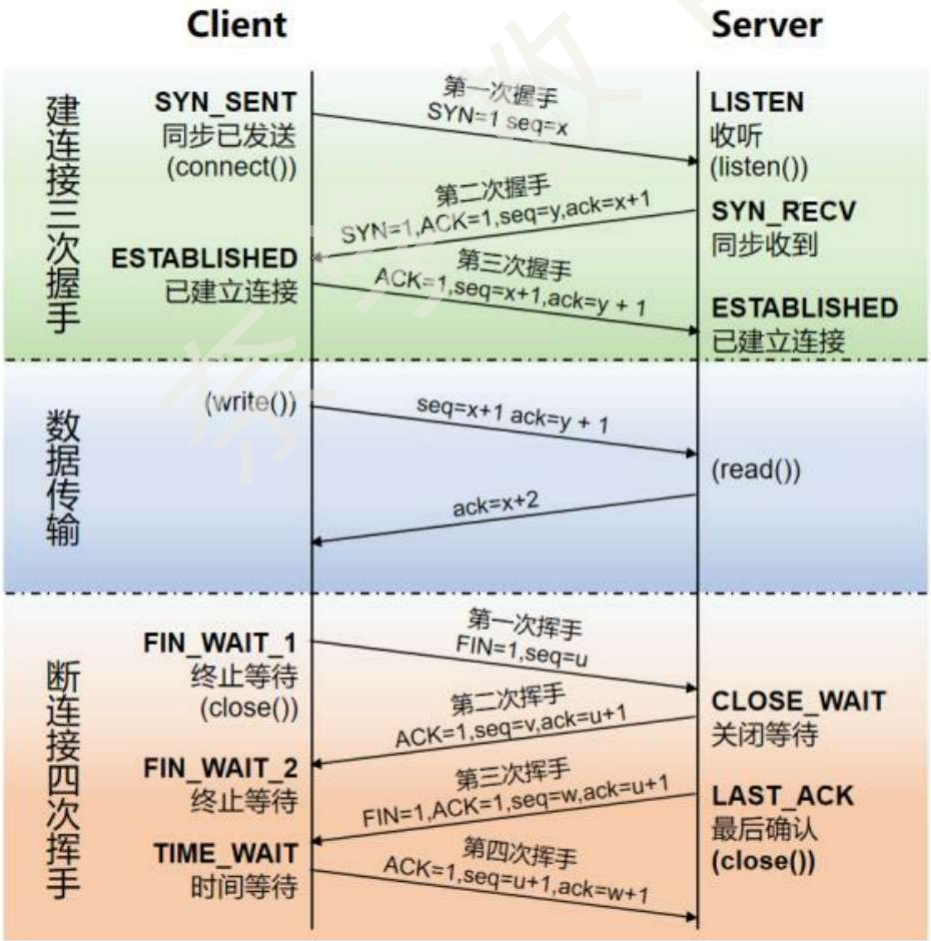
（8）紧急指针：只有当URG标志为1时紧急指针才有效。紧急指针是一个正的偏移量，和顺序号属性中的值相加表示紧急数据最后一个字节的序号。TCP的紧急方式是发送端向另一端发送紧急数据的一种方式。

（9）选项和填充：最常见的可选属性是最长报文大小，又称为MSS（Maximum Segment Size），每个连接方通常都在通信的第一个报文段（为建立连接而设置SYN标志为1的那个段）中指明这个选项，它表示本端所能接收的最大报文段的

长度。选项长度不一定是32位的整数倍，所以要加填充位，即在这个属性中加入额外的零，以保证TCP头部长度是32位的整数倍。

（10）数据部分：TCP报文段中的数据部分是可选的。在一个连接建立和一个连接终止时，双方交换的报文段仅有TCP头部。如果一方没有数据要发送，也使用没有任何数据的头部来确认收到的数据。在处理超时的许多情况中，也会发送不带任何数据的报文段。

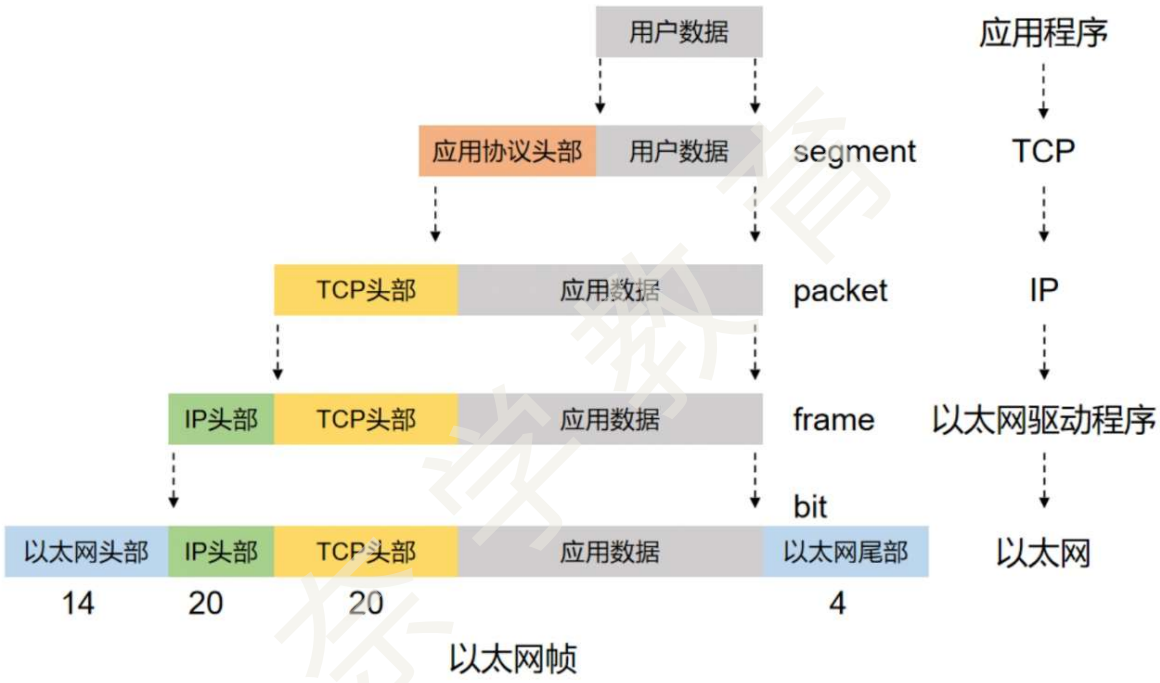
4.4 TCP交互流程



A： 你好，我是A 请求

B： 你好A,我是B 应答

A：你好B， 应答之应答



5 应用层

电脑 1走的tcp协议发送一段东西过来，发送到电脑2023端口

Get http://localhost:8080 http/1.1

key:value1

key:value2

http: smtp mq

5.1 DNS协议

DNS是英文Domain Name System（域名系统）的缩写，用来把便于人们使用的机器名字转换为IP地址。现在顶级域名TLD（Totel Lead Domination）分为三大类：国家顶级域名nTLD、通用顶级域名gTLD和基础结构域名。域名服务器分为四种类型：根域名服务器、顶级域名服务器、本地域名服务器和权限域名服务器。DNS使用TCP和UDP端口53。当前，对于每一级域名长度的限制是63个字符，域名总长度则不能超过253个字符。

5.2 HTTP

HTTP（HyperText Transfer Protocol，超文本传输协议）是面向事务的应用层协议。它是互联网上能够可靠地交换信息的重要基础。HTTP使用面向连接的TCP作为运输层协议，保证了数据的可靠传输。

5.3 FTP

FTP（File Transfer Protocol，文件传输协议）是互联网上使用最广泛的文件传送协议。FTP提供交互式的访问，允许客户指明文件类型与格式，并允许文件具有存取权限。FTP基于TCP工作。

5.4 SMTP

SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）规定了在两个相互通信的SMTP进程之间应如何交换信息。SMTP通信的三个阶段：建立连接、邮件传送和连接释放。

5.5 POP3

POP3（Post Office Protocol 3，邮件读取协议）通常被用来接收电子邮件。

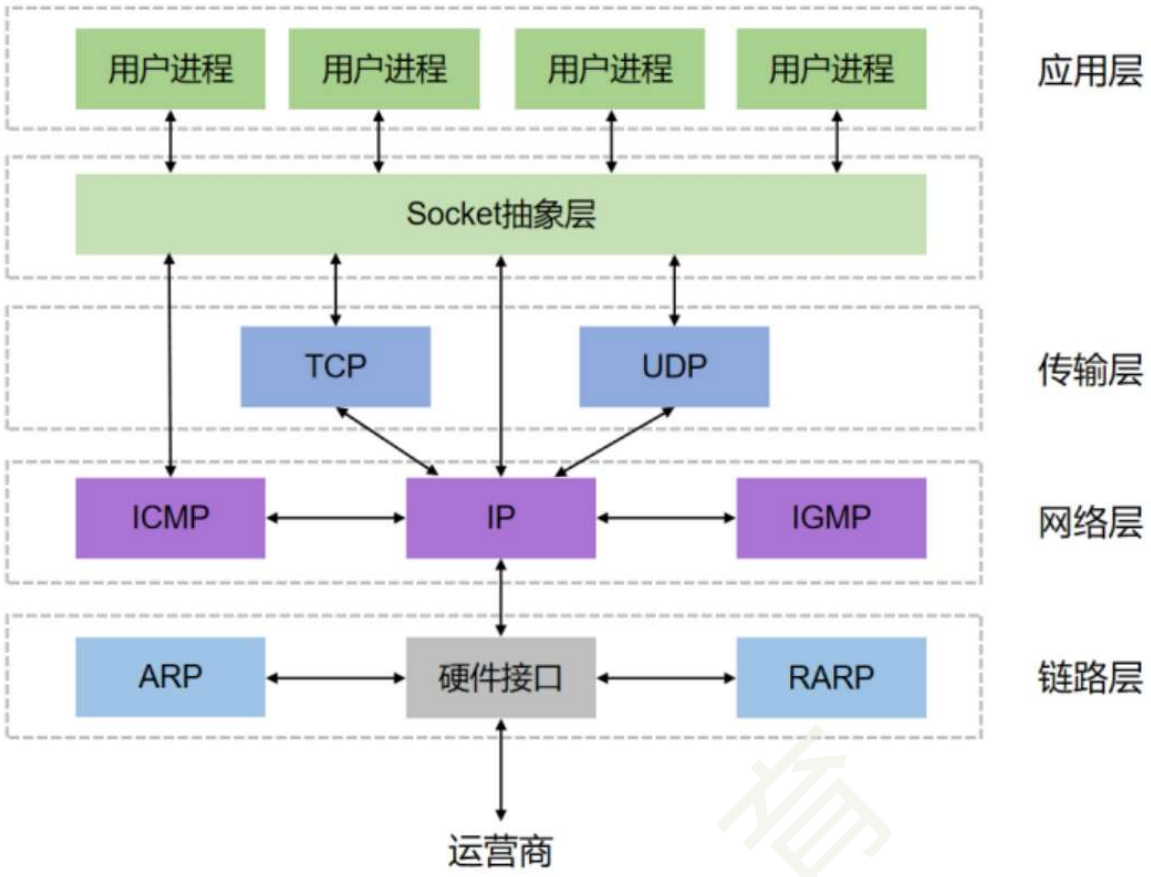
5.6 Telnet协议

Telnet协议是一个简单的远程终端协议，也是互联网的正式标准，又称为终端仿真协议。

网络通信之Socket

Socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用。

我们知道IP层的IP地址可以唯一标识主机，而TCP层的协议和端口号可以唯一标识主机的一个进程，可以用IP地址+协议+端口号唯一标识网络中的一个进程。



Socket是一种从打开，到完成读、写操作，最后关闭的模式，服务器和客户端各自维护一个“文件”，在建立连接打开文件后，可以向自己的文件写入内容供对方读取或者读取对方的内容，通信结束时关闭文件。

Java和IO网络模型

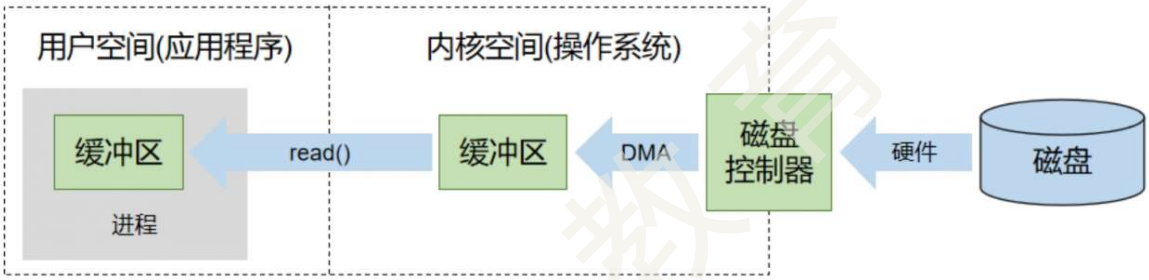
什么是I/O?

在信息交换的过程中，计算机都是对这些流进行数据的收发操作，简称为I/O操作（Input and Output）

I/O交互流程

操作系统的核心是内核，它独立于普通的应用程序，可以访问受保护的内核空间，也有访问底层硬件设备的所有权限。为了保护内核的安全，现在操作系统一般都强制用户进程不能直接操作内核，所以操作系统把内存空间划分成了两个部分：内核空间和用户空间。

通常用户进程中的一次完整I/O交互流程分为两阶段，首先是经过内核空间，也就是由操作系统处理；紧接着就是到用户空间，也就是交由应用程序。



内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。

操作系统和驱动程序运行在内核空间，应用程序运行在用户空间，两者不能简单地使用指针传递数据。因为Linux使用的虚拟内存机制，必须通过系统调用请求Kernel来协助完成I/O操作，内核会为每个I/O设备维护一个缓冲区，用户空间的数据可能被换出，所以当内核空间使用用户空间的指针时，对应的数据可能不在内存中。

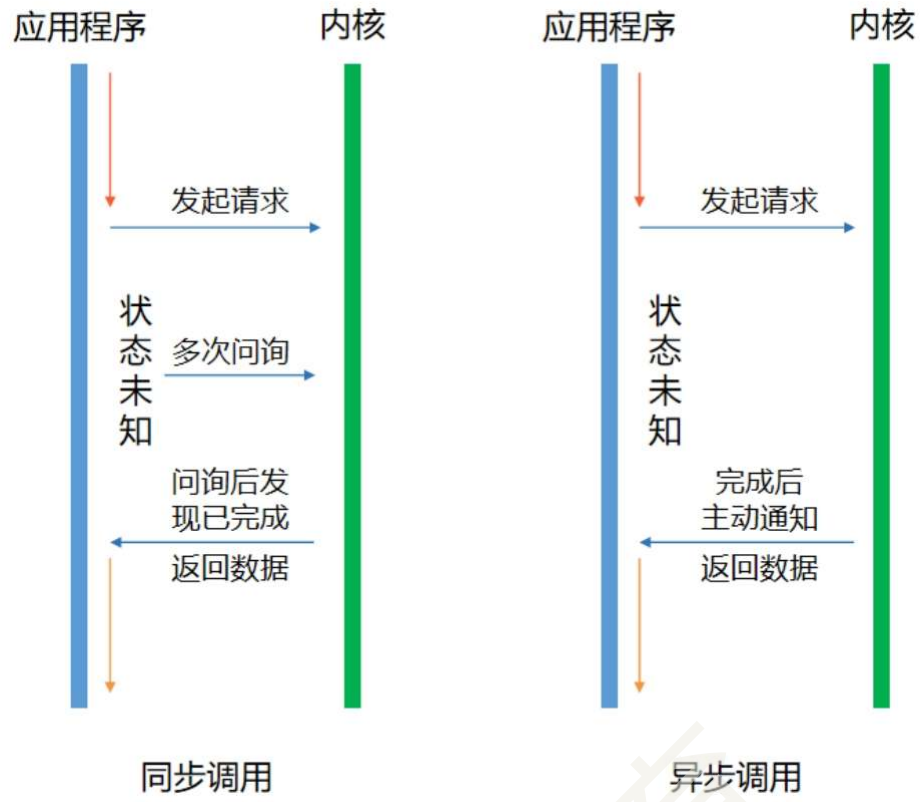
一个网络输入操作通常包括两个不同阶段。

（1）等待网络数据到达网卡，然后将数据读取到内核缓冲区。

（2）从内核缓冲区复制数据，然后拷贝到用户空间。I/O有内存I/O、网络I/O和磁盘I/O三种，通常我们说的I/O指的是后两者。如下图所示是I/O通信过程的调度示意。

同步和异步

同步和异步其实是指CPU时间片的利用，主要看请求发起方对消息结果的获取是主动发起的，还是被动通知的，如下图所示。如果是请求方主动发起的，一直在等待应答结果（同步阻塞），或者可以先去处理其他事情，但要不断轮询查看发起的请求是否有应答结果（同步非阻塞），因为不管如何都要发起方主动获取消息结果，所以形式上还是同步操作。如果是由服务方通知的，也就是请求方发出请求后，要么一直等待通知（异步阻塞），要么先去干自己的事（异步非阻塞）。当事情处理完成后，服务方会主动通知请求方，它的请求已经完成，这就是异步。异步通知的方式一般通过状态改变、消息通知或者回调函数来完成，大多数时候采用的都是回调函数。

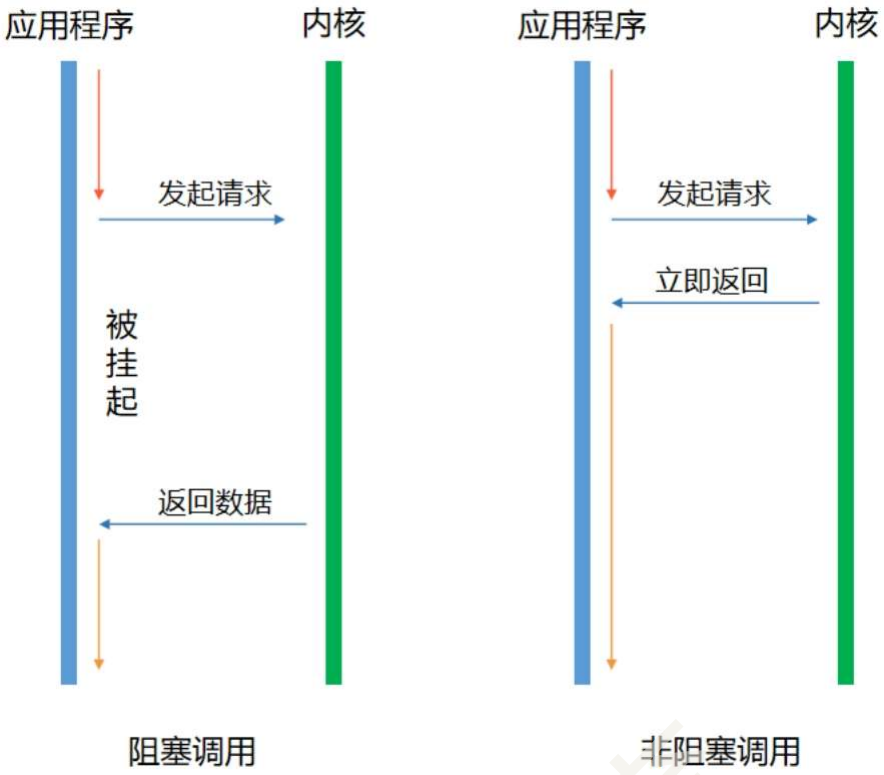


同步：调用者会被阻塞直到 IO 操作完成，调用的结果随着请求的结束而返回。

异步：调用者不会被阻塞，调用的结果不随着请求的结束而返回，而是通过通知或回调函数的形式返回

阻塞和非阻塞

阻塞和非阻塞在计算机的世界里，通常指针对I/O的操作，如网络I/O和磁盘I/O等。那么什么是阻塞和非阻塞呢？简单地说，就是我们调用了一个函数后，在等待这个函数返回结果之前，当前的线程是处于挂起状态还是运行状态。如果是挂起状态，就意味着当前线程什么都不能干，就等着获取结果，这就是同步阻塞；如果仍然是运行状态，就意味着当前线程是可以继续处理其他任务的，但要时不时地看一下是否有结果了，这就是同步非阻塞



老张爱喝茶，废话不说，煮开水。

出场人物：老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。

1 老张把水壶放到火上，立等水开。 **（同步阻塞）**

老张觉得自己有点傻

2 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。 **（同步非阻塞）**

老张还是觉得自己有点傻，于是变高端了，买了把会响笛的那种水壶。水开之后，能大声发出嘀~~~~的噪音。

3 老张把响水壶放到火上，立等水开。 **（异步阻塞）**

老张觉得这样傻等意义不大

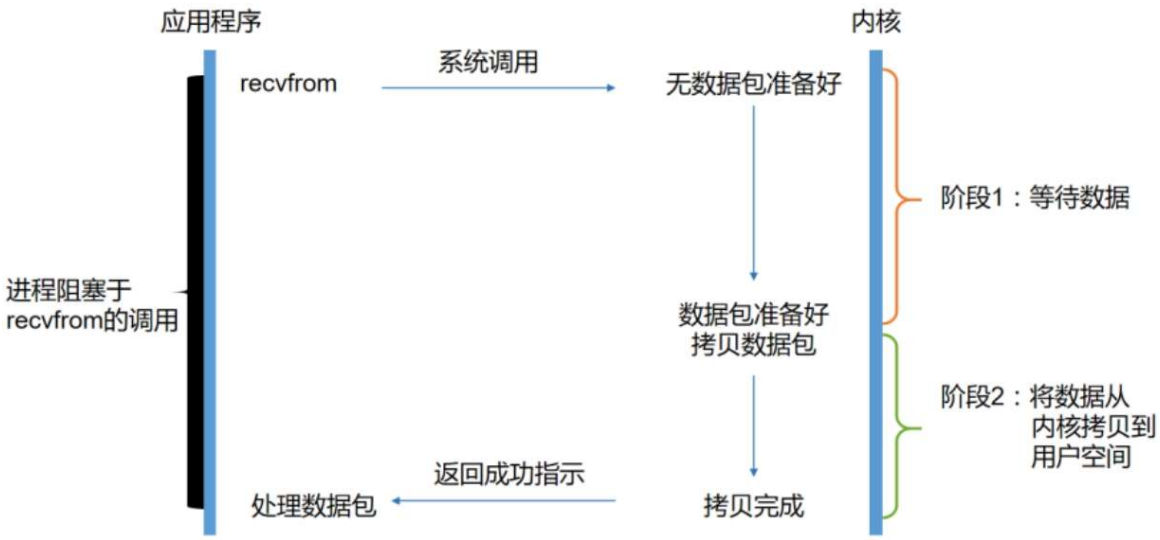
> 4 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。**（异步非阻塞）**
>
> 老张觉得自己聪明了。
>
> 所谓同步异步，只是对于水壶而言。
>
> 普通水壶，同步；响水壶，异步。
>
> 虽然都能干活，但响水壶可以在自己完工之后，提示老张水开了。这是普通水壶所不能及的。
>
> 同步只能让调用者去轮询自己（情况2中），造成老张效率的低下。
>
> 所谓阻塞非阻塞，仅仅对于老张而言。
>
> 立等的老张，阻塞；看电视的老张，非阻塞。

五种I/O通信模型

五种I/O模型，分别是：

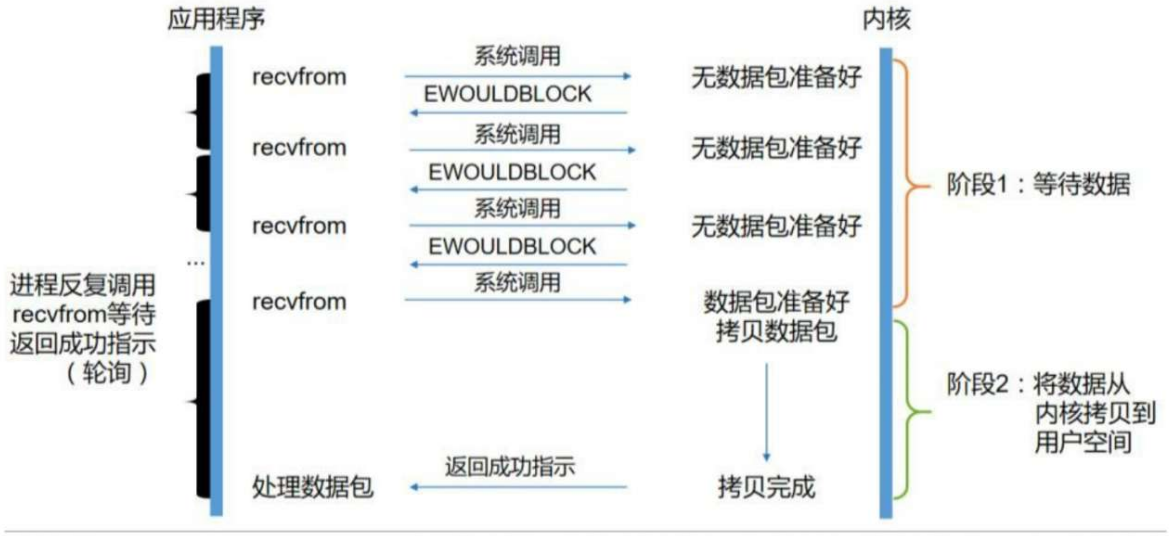
- 阻塞I/O模型
- 非阻塞I/O模型
- 多路复用I/O模型
- 信号驱动I/O模型
- 异步I/O模型

阻塞I/O模型



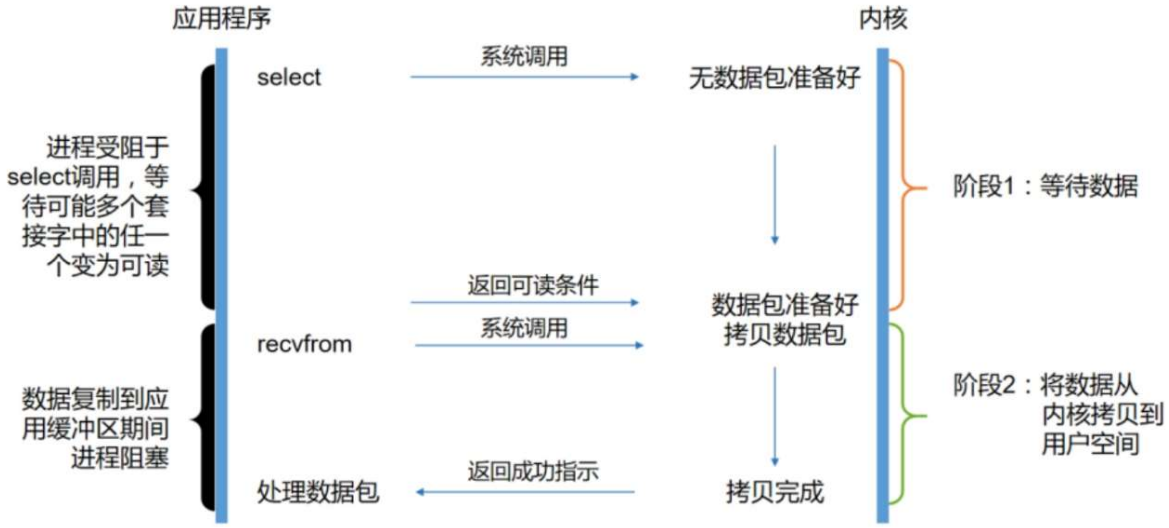
当用户进程调用了recvfrom这个系统调用，内核就开始了I/O的第一个阶段：准备数据。对于网络I/O来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的UDP包），这个时候内核就要等待足够的数据到来。而在用户进程这边，整个进程会被阻塞，当数据准备好时，它就会将数据从内核拷贝到用户内存，然后返回结果，用户进程才解除阻塞的状态，重新运行起来。几乎所有的开发者第一次接触到的网络编程都是从listen()、send()、recv()等接口开始的，这些接口都是阻塞型的

非阻塞I/O模型



当用户进程发出read操作时，如果内核中的数据还没有准备好，那么它并不会阻塞用户进程，而是立刻返回一个error。从用户进程角度讲，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果，用户进程判断结果是一个error时，它就知道数据还没有准备好。于是它可以再次发送read操作，一旦内核中的数据准备好了，并且再次收到了用户进程的系统调用，那么它会马上将数据拷贝到用户内存，然后返回，非阻塞型接口相比于阻塞型接口的显著差异在于，在被调用之后立即返回

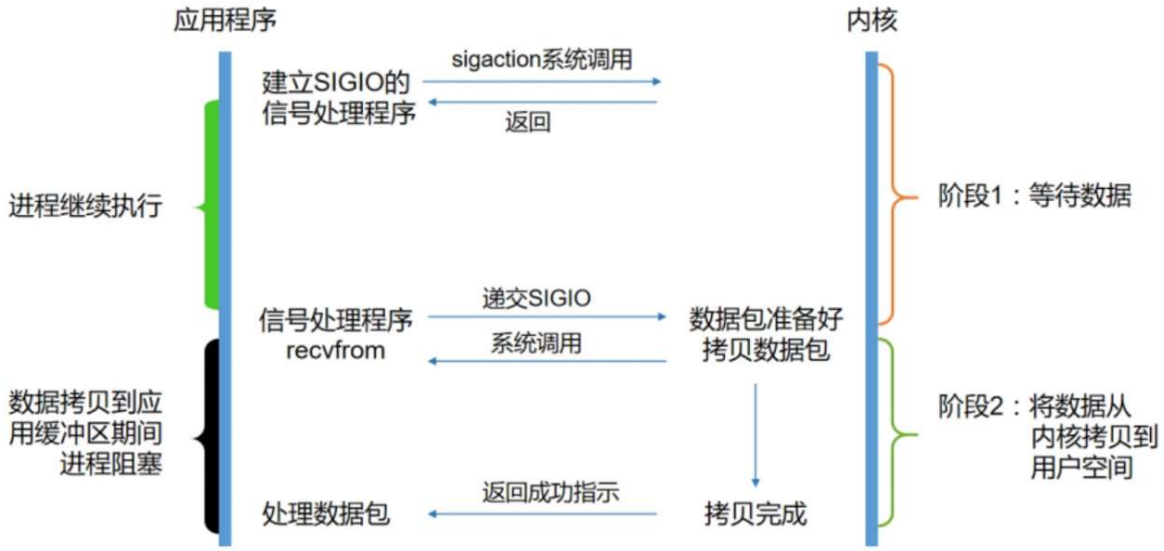
多路复用I/O模型



多个进程的I/O可以注册到一个复用器（Selector）上，当用户进程调用该Selector，Selector会监听注册进来的所有I/O，如果Selector监听的所有I/O在内核缓冲区都没有可读数据，select调用进程会被阻塞，而当任一I/O在内核缓冲区中有可读数据时，select调用就会返回，而后select调用进程可以自己或通知另外的进程（注册进程）再次发起读取I/O，读取内核中准备好的数据，多个进程注册I/O后，只有一个select调用进程被阻塞。

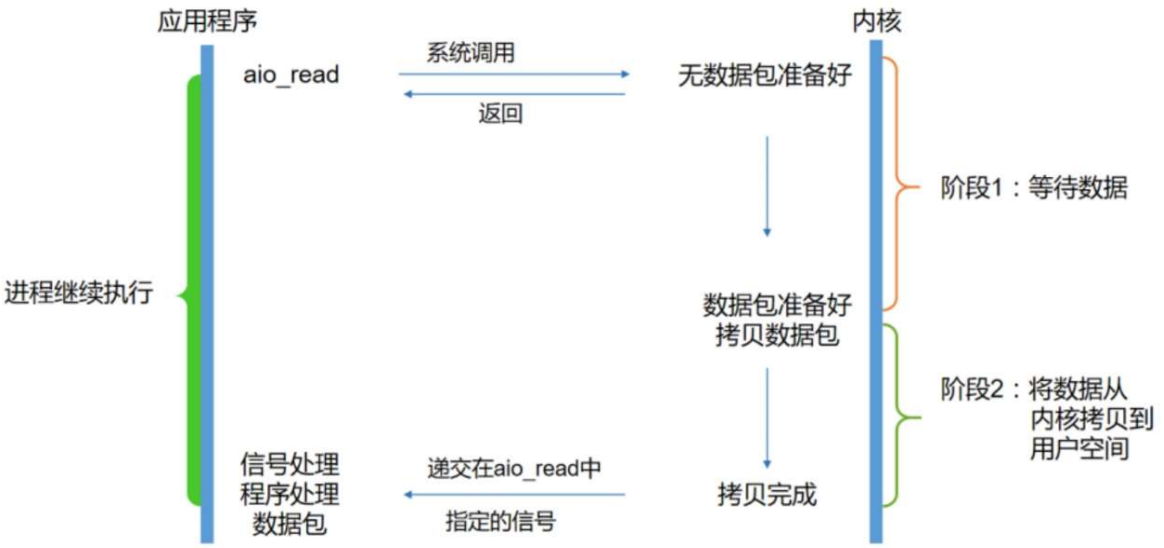
多路复用I/O相对阻塞和非阻塞更难简单说明，所以额外解释一段，其实多路复用I/O模型和阻塞I/O模型并没有太大的不同，**事实上，还更差一些，因为这里需要使用两个系统调用（select和recvfrom），而阻塞I/O模型只有一次系统调用（recvfrom）。但是，用Selector的优势在于它可以同时处理多个连接，所以如果处理的连接数不是很多**，使用select/epoll的Web Server不一定比使用多线程加阻塞I/O的Web Server性能更好，可能延迟还更大，select/epoll的优势并不是对于单个连接能处理得更快，而是能处理更多的连接

信号驱动I/O模型



信号驱动I/O是指进程预先告知内核，向内核注册一个信号处理函数，然后用户进程返回不阻塞，当内核数据就绪时会发送一个信号给进程，用户进程便在信号处理函数中调用I/O读取数据。从上图可以看出，实际上I/O内核拷贝到用户进程的过程还是阻塞的，信号驱动I/O并没有实现真正的异步，因为通知到进程之后，依然由进程来完成I/O操作。这和后面的异步I/O模型很容易混淆，需要理解I/O交互并结合五种I/O模型进行比较阅读

异步I/O模型

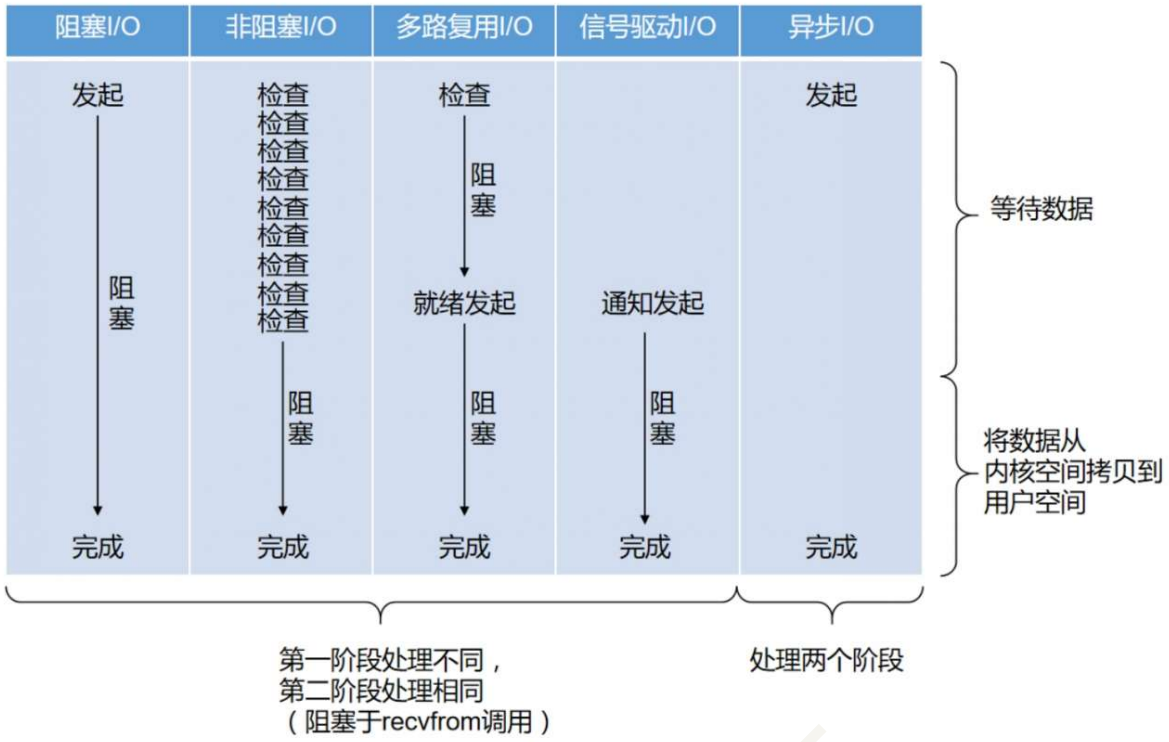


用户进程发起aio_read操作后，给内核传递与read相同的描述符、缓冲区指针、缓冲区大小三个参数及文件偏移，告诉内核当整个操作完成时，如何通知我们立刻就可以开始去做其他的事；而另一方面，从内核的角度，当它收到一个aio_read之后，首先它会立刻返回，所以不会对用户进程产生任何阻塞，内核会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，内核会给用户进程发送一个信号，告诉它aio_read操作完成。

异步I/O的工作机制是：告知内核启动某个操作，并让内核在整个操作完成后通知我们，这种模型与信号驱动I/O模型的区别在于，信号驱动I/O模型是由内核通知我们何时可以启动一个I/O操作，这个I/O操作由用户自定义的信号函数来实现，而异步I/O模型由内核告知我们I/O操作何时完成。

各I/O模型的对比与总结

其实前四种I/O模型都是同步I/O操作，它们的区别在于第一阶段，而第二阶段是一样的：在数据从内核拷贝到应用缓冲区期间（用户空间），进程阻塞于recvfrom调用。有人可能会说，NIO（Non-Blocking I/O）并没有被阻塞。这里有个非常“狡猾”的地方，定义中所指的“I/O Operation”是指真实的I/O操作。NIO在执行recvfrom的时候，如果内核（Kernel）的数据没有准备好，这时候不会阻塞进程。但是，当内核（Kernel）中数据准备好的时候，recvfrom会将数据从内核（Kernel）拷贝到用户内存中，这个时候进程就被阻塞了。在这段时间内，进程是被阻塞的



从上图可以看出，阻塞程度：阻塞I/O>非阻塞I/O>多路复用I/O>信号驱动I/O>异步I/O，效率是由低到高的。最后，再看一下下表，从多维度总结了各I/O模型之间的差异，可以加深理解。

属　　性	同步阻塞I/O	伪异步I/O	非阻塞I/O（NIO）	异步I/O
客户端数:I/O线程数	1:1	M:N（M>=N）	M:1	M:0
阻塞类型	阻塞	阻塞	非阻塞	非阻塞
同步	同步	同步	同步（多路复用）	异步
API使用难度	简单	简单	复杂	一般
调试难度	简单	简单	复杂	复杂
可靠性	非常差	差	高	高
吞吐量	低	中	高	高

在计算机的早期，所有的网络通信使用的都是阻塞型 IO，即 BIO，随着计算机技术的不断发展，才衍生出了其它四种模型。而且，在当前这个阶段，linux 系统上 AIO 还不成熟，因此，现在 NIO 才是最流行的。

Java中BIO、NIO和AIO

BIO

定义：同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务
器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开 销，当然可以通过线程池机制改善。

场景：用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
问题：当并发数较大时，需要创建大量线程来处理连接，系统资源占用较大；连接建立 后，如果当前线程暂时没有数据可读，则线程就阻塞在Read操作上，造成线程资源浪费

NIO

定义：同步非阻塞，服务器实现模式为多个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

场景：适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持

AIO

定义： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理

场景：用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持

所以，从效率上来看，AIO 无疑是最高的，然而，遗憾地是，目前作为广大服务器使用的系统 linux 对 AIO 的支持还不完善，导致我们还不能放心地使用 AIO 这项技术，不过，我相信有一天 AIO 会成为那颗闪亮的星的，现阶段，还是以学好 NIO 为主。

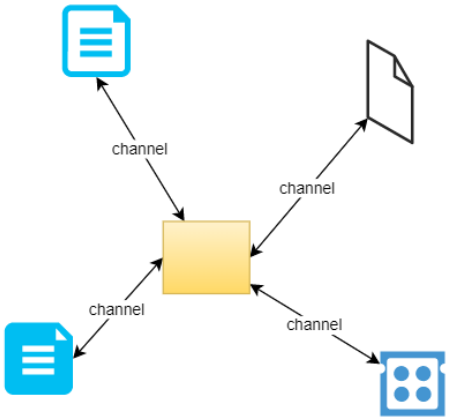
Java NIO三件套

在NIO中有三个核心对象需要掌握：缓冲区（Buffer）、选择器（Selector）和通道（Channel）

Channel

Channel 是一种 IO 操作的连接（nexus，连接的意思），它代表的是到实体的开放连接，这个实体可以是硬件设备、文件、网络套接字或者可执行 IO 操作（比如读、写）的程序组件。

在 linux 系统中，一切皆可看作是文件，所以，简单点讲，Channel 就是到文件的连接，并可以通过 IO 操作这些文件。



因此，针对不同的文件类型又衍生出了**不同类型的 Channel**：

- FileChannel：操作普通文件
- DatagramChannel：用于 UDP 协议
- SocketChannel：用于 TCP 协议，客户端与服务端之间的 Channel
- ServerSocketChannel：用于 TCP 协议，仅用于服务端的 Channel

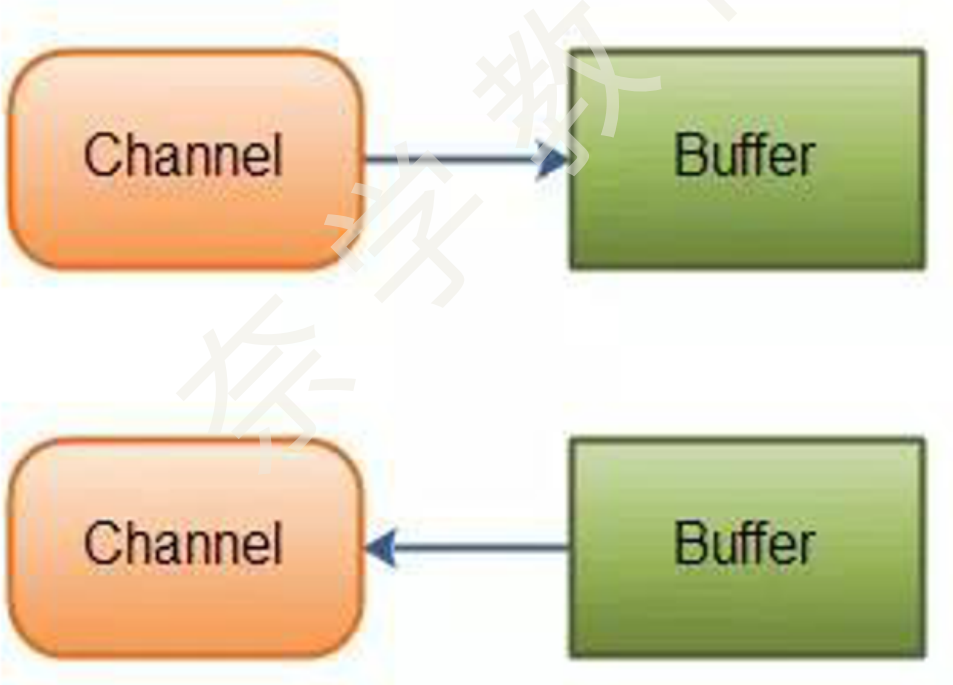
> ServerSocketChannel 和 SocketChannel 是专门用于 TCP 协议中的。
> ServerSocketChannel 是一种服务端的 Channel，只能用在服务端，可以看作是到网卡的一种 Channel，它监听着网卡的某个端口。
> SocketChannel 是一种客户端与服务端之间的 Channel，客户端连接到服务器的网卡之后，被服务端的 Channel 监听到，然后与客户端之间建立一个 Channel，这个 Channel 就是 SocketChannel。

那么，这些 **Channel 又该如何使用呢？** 我们以 FileChannel 为代表来写一个简单的示例。

```java
public class FileChannelTest {
    public static void main(String[] args) throws IOException {
        // 从文件获取一个FileChannel
        FileChannel fileChannel = new RandomAccessFile("D:\\object.txt", "rw").getChannel();
        // 声明一个Byte类型的Buffer
        ByteBuffer buffer = ByteBuffer.allocate(10);
        // 将FileChannel中的数据读出到buffer中，-1表示读取完毕
        // buffer默认为写模式
        // read()方法是相对channel而言的，相对buffer就是写
        while ((fileChannel.read(buffer)) != -1) {
            // buffer切换为读模式
            buffer.flip();
            // buffer中是否有未读数据
            while (buffer.hasRemaining()) {
                // 未读数据的长度
                int remain = buffer.remaining();
                // 声明一个字节数组
                byte[] bytes = new byte[remain];
                // 将buffer中数据读出到字节数组中
                buffer.get(bytes);
                // 打印出来
                System.out.println(new String(bytes, StandardCharsets.UTF_8));
```

```
23                  }
24                  // 清空buffer，为下一次写入数据做准备
25                  // clear()会将buffer再次切换为写模式
26                  buffer.clear();
27              }
28          }
29      }
```

通过上面的示例，我们可以发现，Channel 是和 Buffer 一起使用的，那么，什么是 Buffer 呢？为什么要和 Buffer 一起使用呢？必须吗？



缓冲区

Buffer 是一个容器，什么样的容器呢？存放数据的容器。存放什么样的数据呢？特定基本类型的数据。这个容器有哪些特点呢？它是线性的，有限的序列，元素是某种基本类型的数据。它又有哪些属性呢？主要有三个属性：capacity、limit、position。

那么，**Buffer 为什么要和 Channel 一起使用呢？必须一起使用吗?**

Buffer操作基本API

缓冲区实际上是一个容器对象，更直接地说，其实就是一个数组，在NIO库中，所有数据都是用缓冲区处理的

在读取数据时，它是直接读到缓冲区中的；在写入数据时，它也是写入缓冲区的；任何时候访问NIO中的数据，都是将它放到缓冲区中。

在NIO中，所有的缓冲区类型都继承于抽象类Buffer，最常用的就是ByteBuffer

Buffer的基本原理

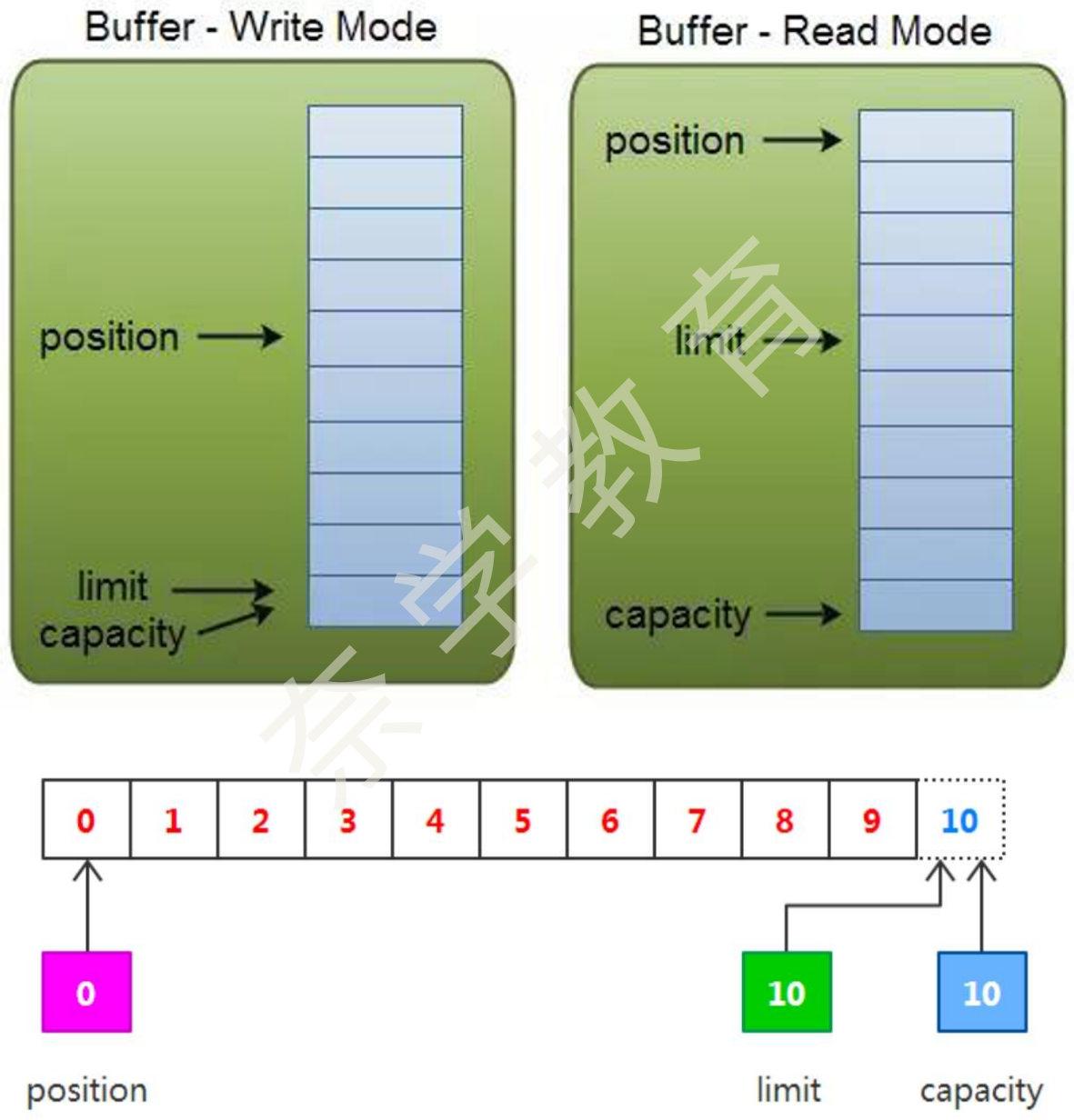
在缓冲区中，最重要的属性有下面三个，它们一起合作完成对缓冲区内部状态的变化跟踪

position：指定下一个将要被写入或者读取的元素索引，它的值由get()/put()方法自动更新，在新创建一个Buffer对象时，position被初始化为0。

limit：指定还有多少数据需要取出（在从缓冲区写入通道时），或者还有多少空间可以放入数据（在从通道读入缓冲区时）。

capacity：指定了可以存储在缓冲区中的最大数据容量

缓冲区示意图



缓冲区写入：写入时position右移

缓冲区读取：一是把limit设置为当前的position值。二是把position设置为0。通过position下标从0开始读取，终点为limit。

缓冲区清空：恢复初始状态。

选择器

select，poll和epoll其实都是操作系统中IO多路复用实现的方法。

select

select方法本质其实就是维护了一个文件描述符（fd）数组，以此为基础，实现IO多路复用的功能。这个fd数组有长度限制，在32位系统中，最大值为1024个，而在64位系统中，最大值为2048个，这个配置可以调用

select方法被调用，首先需要将fd_set从用户空间拷贝到内核空间，然后内核用poll机制（此poll机制非IO多路复用的那个poll方法，可参加附录）直到有一个fd活跃，或者超时了，方法返回。

- fd_set在用户空间和内核空间的频繁复制，效率低
- 单个进程可监控的fd数量有限制，无论是1024还是2048，对于很多情景来说都是不够用的。
- 基于轮询来实现，效率低

poll

poll本质上和select没有区别，依然需要进行数据结构的复制，依然是基于轮询来实现，但区别就是，select使用的是fd数组，而poll则是维护了一个链表，所以从理论上，poll方法中，单个进程能监听的fd不再有数量限制。但是轮询，复制等select存在的问题，poll依然存在

epoll

epoll就是对select和poll的改进了。它的核心思想是基于事件驱动来实现的，实现起来也并不难，就是给每个fd注册一个回调函数，当fd对应的设备发生IO事件时，就会调用这个回调函数，将该fd放到一个链表中，然后由客户端从该链表中取出一个个fd，以此达到O（1）的时间复杂度

epoll操作实际上对应着有三个函数：**epoll_create**，**epoll_ctr**，**epoll_wait**

epoll_create

epoll_create相当于在内核中创建一个存放fd的数据结构。在select和poll方法中，内核都没有为fd准备存放其的数据结构，只是简单粗暴地把数组或者链表复制进来；而epoll则不一样，epoll_create会在内核建立一颗专门用来存放fd结点的红黑树，后续如果有新增的fd结点，都会注册到这个epoll红黑树上。

epoll_ctr

另一点不一样的是，select和poll会一次性将监听的所有fd都复制到内核中，而epoll不一样，当需要添加一个新的fd时，会调用epoll_ctr，给这个fd注册一个回调函数，然后将该fd结点注册到内核中的红黑树中。当该fd对应的设备活跃时，会调

用该fd上的回调函数，将该结点存放在一个就绪链表中。这也解决了在内核空间和用户空间之间进行来回复制的问题。

epoll_wait

epoll_wait的做法也很简单，其实直接就是从就绪链表中取结点，这也解决了轮询的问题，时间复杂度变成O(1)

所以综合来说，epoll的优点有：

- 没有最大并发连接的限制，远远比1024或者2048要大。（江湖传言1G的内存上能监听10W个端口）
- 效率变高。epoll是基于事件驱动实现的，不会随着fd数量上升而效率下降
- 减少内存拷贝的次数

	select	poll	epoll
操作方式	遍历	遍历	回调
底层实现	数组	链表	哈希表
IO效率	每次调用都进行线性遍历，时间复杂度为O(n)	每次调用都进行线性遍历，时间复杂度为O(n)	事件通知方式，每当fd就绪，系统注册的回调函数就会被调用，将就绪fd放到readyList里面，时间复杂度O(1)
最大连接数	1024	无上限	无上限
fd拷贝	每次调用select，都需要把fd集合从用户态拷贝到内核态	每次调用poll，都需要把fd集合从用户态拷贝到内核态	调用epoll_ctl时拷贝进内核并保存，之后每次epoll_wait不拷贝