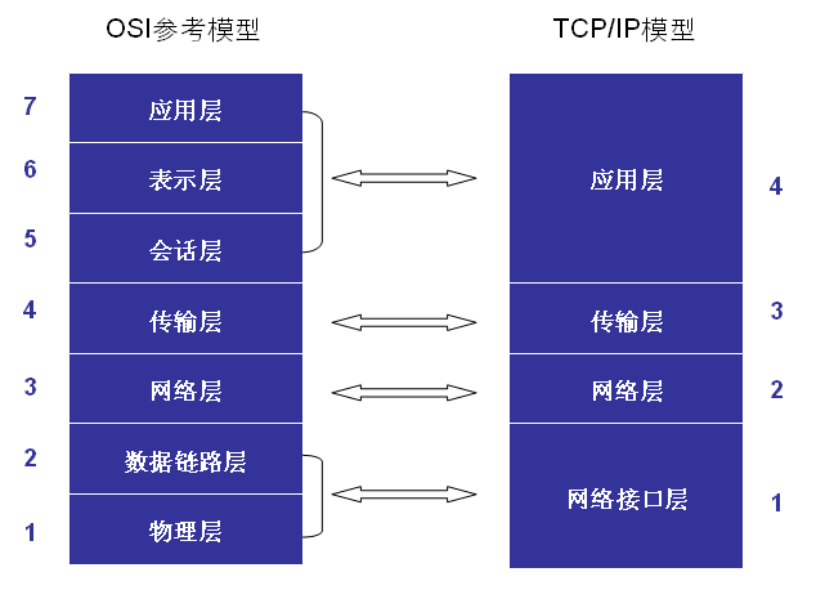
网络基础

分层模型

OSI 七层模型



OSI 模型

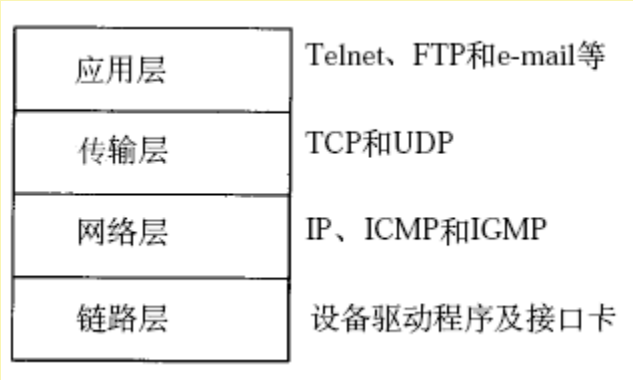
1. **物理层**：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由 1、0 转化为电流强弱来进行传输，到达目的地后再转化为 1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。
2. **数据链路层**：定义了如何让格式化数据以帧为单位进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。如：串口通信中使用到的 115200、8、N、1
3. **网络层**：在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。 Internet 的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。
4. **传输层**：定义了一些传输数据的协议和端口号（WWW 端口 80 等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与 TCP 特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如 QQ 聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。
5. **会话层**：通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的系统之间发起会话或者

接受会话请求（设备之间需要互相认识可以是 IP 也可以是 MAC 或者是主机名）。

6. **表示层**：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC 程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码 (EBCDIC)，而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。
7. **应用层**：是最靠近用户的 OSI 层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。

TCP/IP 四层模型

TCP/IP 网络协议栈分为应用层（Application）、传输层（Transport）、网络层（Network）和链路层（Link）四层。如下图所示：



TCP/IP 模型

一般在应用开发过程中，讨论最多的是 TCP/IP 模型。

协议的概念

什么是协议

从应用的角度出发，协议可理解为“规则”，是数据传输和数据的解释的规则。

假设，A、B 双方欲传输文件。规定：

第一次，传输文件名，接收方接收到文件名，应答 OK 给传输方；
第二次，发送文件的尺寸，接收方接收到该数据再次应答一个 OK；
第三次，传输文件内容。同样，接收方接收数据完成后应答 OK 表示文件内容接收成功。

由此，无论 A、B 之间传递何种文件，都是通过三次数据传输来完成。A、B 之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B 之间达成的这个相互遵守的规则即为协议。

这种仅在 A、B 之间被遵守的协议称之为**原始协议**。当此协议被更多的人采用，不断的增加、改进、维护、完善。最终形成一个稳定的、完整的文件传输协议，被广泛应用于各种文件传输过程中。该协议就成为一个**标准协议**。最早

的 ftp 协议就是由此衍生而来。

TCP 协议注重数据的传输。http 协议着重于数据的解释。

典型协议

传输层 常见协议有 TCP/UDP 协议。

应用层 常见的协议有 HTTP 协议，FTP 协议。

网络层 常见协议有 IP 协议、ICMP 协议、IGMP 协议。

网络接口层 常见协议有 ARP 协议、RARP 协议。

TCP 传输控制协议（Transmission Control Protocol）是一种面向连接的、可靠的、基于字节流的传输层通信协议。

UDP 用户数据报协议（User Datagram Protocol）是 OSI 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

HTTP 超文本传输协议（Hyper Text Transfer Protocol）是互联网上应用最为广泛的一种网络协议。

FTP 文件传输协议（File Transfer Protocol）

IP 协议是因特网互联协议（Internet Protocol）

ICMP 协议是 Internet 控制报文协议（Internet Control Message Protocol）它是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息。

IGMP 协议是 Internet 组管理协议（Internet Group Management Protocol），是因特网协议家族中的一个组播协议。该协议运行在主机和组播路由器之间。

ARP 协议是正向地址解析协议（Address Resolution Protocol），通过已知的 IP，寻找对应主机的 MAC 地址。

RARP 是反向地址转换协议，通过 MAC 地址确定 IP 地址。

网络应用程序设计模式

C/S 模式

传统的网络应用设计模式，客户机(client)/服务器(server)模式。需要在通讯两端各自部署客户机和服务器来完成数据通信。

B/S 模式

浏览器()/服务器(server)模式。只需在一端部署服务器，而另外一端使用每台 PC 都默认配置的浏览器即可完

成数据的传输。

优缺点

对于 C/S 模式来说，其优点明显。客户端位于目标主机上可以保证性能，将数据缓存至客户端本地，从而**提高数据传输效率**。且，一般来说客户端和服务器程序由一个开发团队创作，所以他们之间**所采用的协议相对灵活**。可以在标准协议的基础上根据需求裁剪及定制。例如，腾讯公司所采用的通信协议，即为 ftp 协议的修改剪裁版。

因此，传统的网络应用程序及较大型的网络应用程序都首选 C/S 模式进行开发。如，知名的网络游戏魔兽世界。3D 画面，数据量庞大，使用 C/S 模式可以提前在本地进行大量数据的缓存处理，从而提高观感。

C/S 模式的缺点也较突出。由于客户端和服务器都需要有一个开发团队来完成开发。**工作量**将成倍提升，开发周期较长。另外，从用户角度出发，需要将客户端安插至用户主机上，对用户主机的**安全性构成威胁**。这也是很多用户不愿使用 C/S 模式应用程序的重要原因。

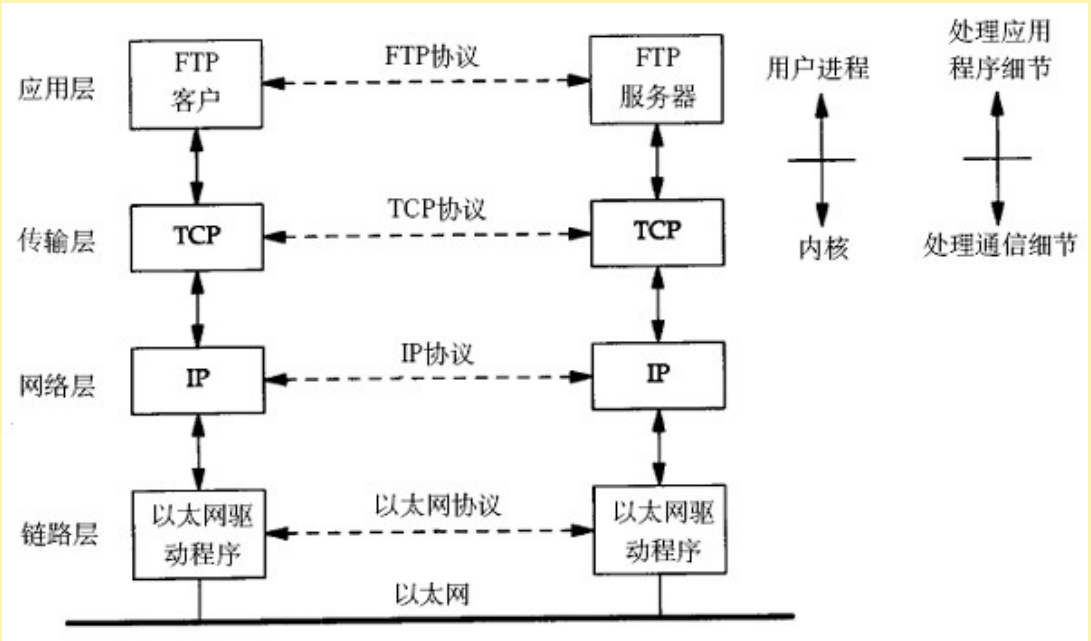
B/S 模式相比 C/S 模式而言，由于它没有独立的客户端，使用标准浏览器作为客户端，其工作**开发量较小**。只需开发服务器端即可。另外由于其采用浏览器显示数据，因此移植性非常好，**不受平台限制**。如早期的偷菜游戏，在各个平台上都可以完美运行。

B/S 模式的缺点也较明显。由于使用第三方浏览器，因此**网络应用支持受限**。另外，没有客户端放到对方主机上，**缓存数据不尽如人意**，从而传输数据量受到限制。应用的观感大打折扣。第三，必须与浏览器一样，采用标准 http 协议进行通信，**协议选择不灵活**。

因此在开发过程中，模式的选择由上述各自的特点决定。根据实际需求选择应用程序设计模式。

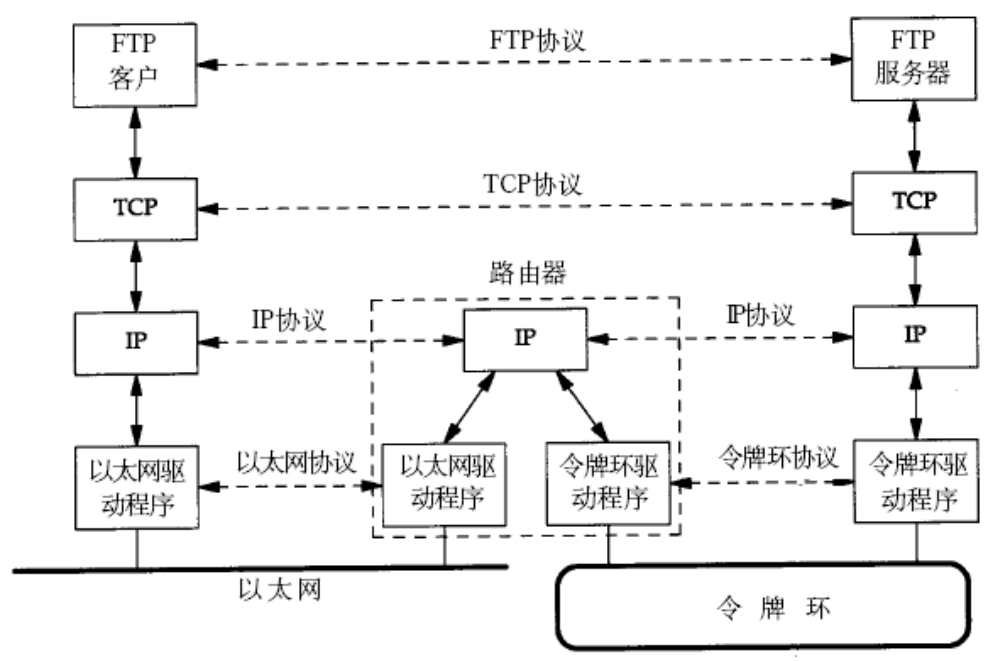
通信过程

两台计算机通过 TCP/IP 协议通讯的过程如下所示：



TCP/IP 通信过程

上图对应两台计算机在同一网段中的情况，如果两台计算机在不同的网段中，那么数据从一台计算机到另一台计算机传输过程中要经过一个或多个路由器，如下图所示：



跨路由通信

链路层有以太网、令牌环网等标准，链路层负责网卡设备的驱动、帧同步（即从网线上检测到什么信号算作新帧的开始）、冲突检测（如果检测到冲突就自动重发）、数据差错校验等工作。交换机是工作在链路层的网络设备，可以在不同的链路层网络之间转发数据帧（比如十兆以太网和百兆以太网之间、以太网和令牌环网之间），由于不同链路层的帧格式不同，交换机要将进来的数据包拆掉链路层首部重新封装之后再转发。

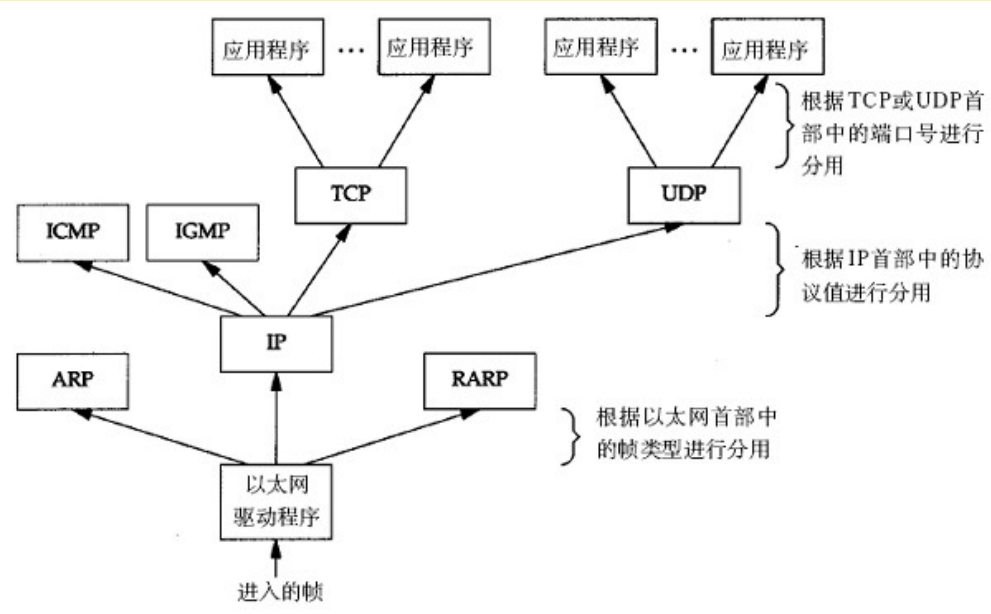
网络层的 IP 协议是构成 Internet 的基础。Internet 上的主机通过 IP 地址来标识，Inter-net 上有大量路由器负责根据 IP 地址选择合适的路径转发数据包，数据包从 Internet 上的源主机到目的主机往往要经过十多个路由器。路由器是工作在第三层的网络设备，同时兼有交换机的功能，可以在不同的链路层接口之间转发数据包，因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。IP 协议不保证传输的可靠性，数据包在传输过程中可能丢失，可靠性可以在上层协议或应用程序中提供支持。

网络层负责点到点（ptop，point-to-point）的传输（这里的“点”指主机或路由器），而传输层负责端到端（etoe，end-to-end）的传输（这里的“端”指源主机和目的主机）。传输层可选择 TCP 或 UDP 协议。

TCP 是一种面向连接的、可靠的协议，有点像打电话，双方拿起电话互通身份之后就建立了连接，然后说话就行了，这边说的话那边保证听得到，并且是按说话的顺序听到的，说完话挂机断开连接。也就是说 TCP 传输的双方需要首先建立连接，之后由 TCP 协议保证数据收发的可靠性，丢失的数据包自动重发，上层应用程序收到的总是可靠的数据流，通讯之后关闭连接。

UDP 是无连接的传输协议，不保证可靠性，有点像寄信，信写好放到邮筒里，既不能保证信件在邮递过程中不会丢失，也不能保证信件寄送顺序。使用 UDP 协议的应用程序需要自己完成丢包重发、消息排序等工作。

目的主机收到数据包后，如何经过各层协议栈最后到达应用程序呢？其过程如下图所示：



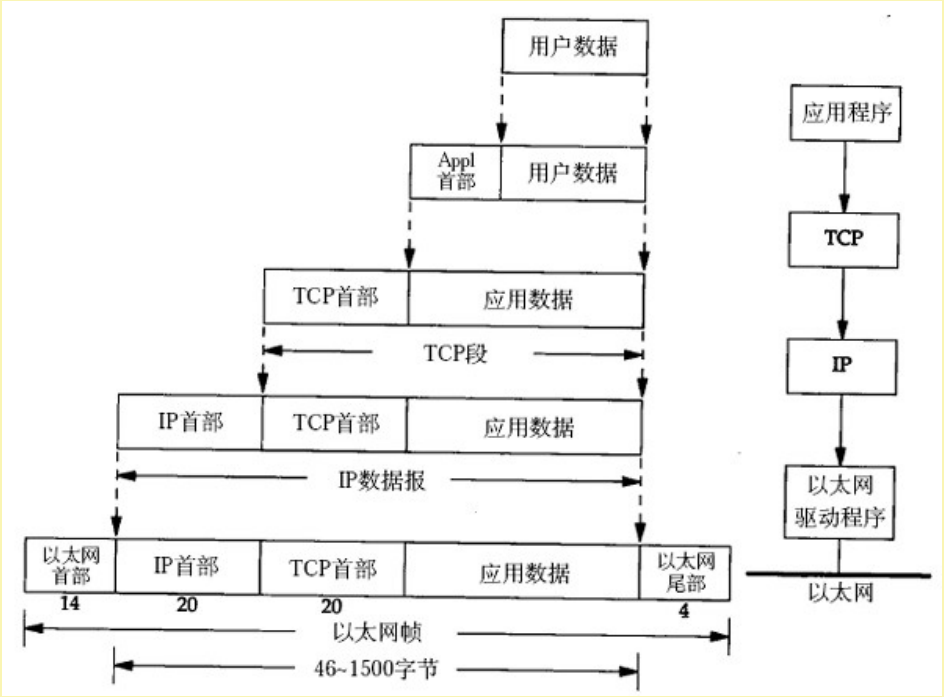
以太网驱动程序首先根据以太网首部中的“上层协议”字段确定该数据帧的有效载荷（payload，指除去协议首部之外实际传输的数据）是 IP、ARP 还是 RARP 协议的数据报，然后交给相应的协议处理。假如是 IP 数据报，IP 协议再根据 IP 首部中的“上层协议”字段确定该数据报的有效载荷是 TCP、UDP、ICMP 还是 IGMP，然后交给相应的协议处理。假如是 TCP 段或 UDP 段，TCP 或 UDP 协议再根据 TCP 首部或 UDP 首部的“端口号”字段确定应该将应用层数据交给哪个用户进程。IP 地址是标识网络中不同主机的地址，而端口号就是同一台主机上标识不同进程的地址，IP 地址和端口号合起来标识网络中唯一的进程。

虽然 IP、ARP 和 RARP 数据报都需要以太网驱动程序来封装成帧，但是从功能上划分，ARP 和 RARP 属于链路层，IP 属于网络层。虽然 ICMP、IGMP、TCP、UDP 的数据都需要 IP 协议来封装成数据报，但是从功能上划分，ICMP、IGMP 与 IP 同属于网络层，TCP 和 UDP 属于传输层。

协议格式

数据包封装

传输层及其以下的机制由内核提供，应用层由用户进程提供（后面将介绍如何使用 socket API 编写应用程序），应用程序对通讯数据的含义进行解释，而传输层及其以下处理通讯的细节，将数据从一台计算机通过一定的路径发送到另一台计算机。应用层数据通过协议栈发到网络上时，每层协议都要加上一个数据首部（header），称为封装（Encapsulation），如下图所示：

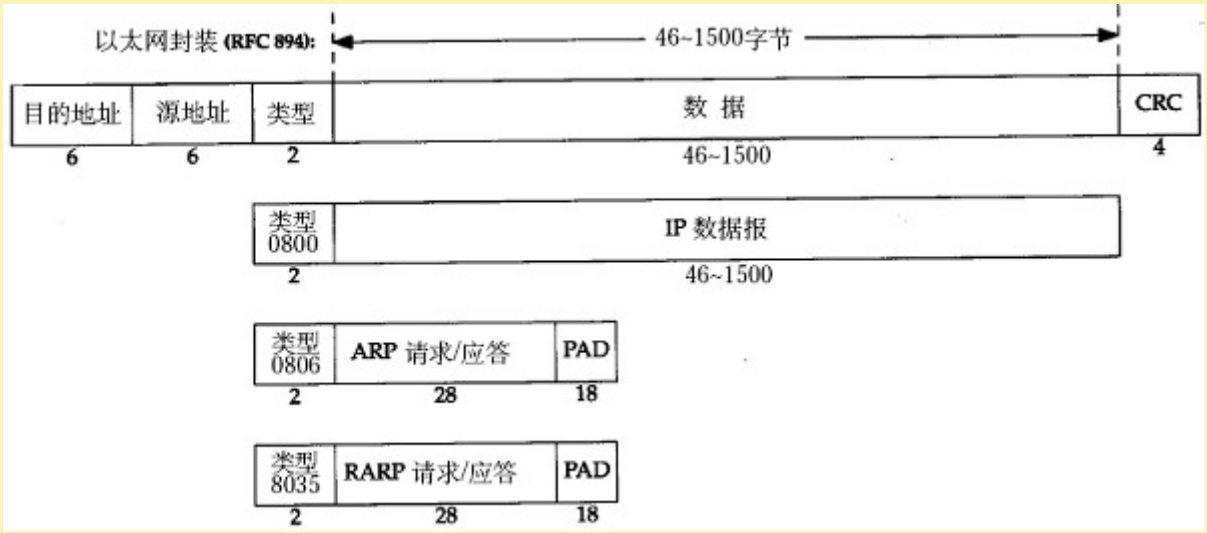


TCP/TP 数据包封装

不同的协议层对数据包有不同的称谓，在传输层叫做段（segment），在网络层叫做数据报（datagram），在链路层叫做帧（frame）。数据封装成帧后发到传输介质上，到达目的主机后每层协议再剥掉相应的首部，最后将应用层数据交给应用程序处理。

以太网帧格式

以太网的帧格式如下所示：



以太网帧格式

其中的源地址和目的地址是指网卡的硬件地址（也叫 MAC 地址），长度是 48 位，是在网卡出厂时固化的。可在 shell 中使用 ifconfig 命令查看，“HWaddr 00:15:F2:14:9E:3F”部分就是硬件地址。协议字段有三种值，分别对应 IP、ARP、RARP。帧尾是 CRC 校验码。

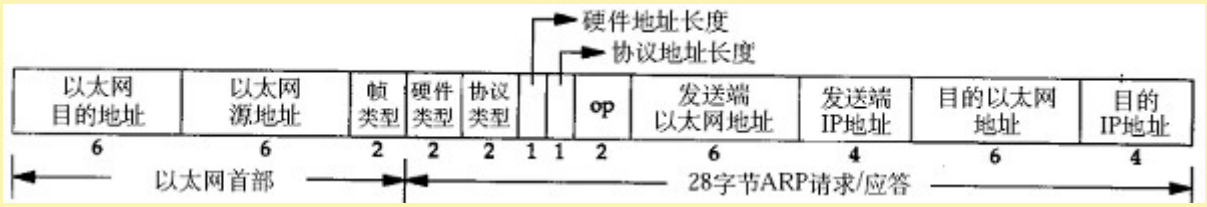
以太网帧中的数据长度规定最小 46 字节，最大 1500 字节，ARP 和 RARP 数据包的长度不够 46 字节，要在后面补填充位。**最大值 1500 称为以太网的最大传输单元（MTU）**，不同的网络类型有不同的 MTU，如果一个数据包从以太网路由到拨号链路上，数据包长度大于拨号链路的 MTU，则需要对数据包进行分片（fragmentation）。ifconfig 命令输出中也有“MTU:1500”。注意，MTU 这个概念指数据帧中有效载荷的最大长度，不包括帧头长度。

ARP 数据报格式

在网络通讯时，源主机的应用程序知道目的主机的 IP 地址和端口号，却不知道目的主机的硬件地址，而数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃。因此在通讯前必须获得目的主机的硬件地址。ARP 协议就起到这个作用。源主机发出 ARP 请求，询问“IP 地址是 192.168.0.1 的主机的硬件地址是多少”，并将这个请求广播到本地网段（以太网帧首部的硬件地址填 FF:FF:FF:FF:FF:FF 表示广播），目的主机接收到广播的 ARP 请求，发现其中的 IP 地址与本机相符，则发送一个 ARP 应答数据包给源主机，将自己的硬件地址填写在应答包中。

每台主机都维护一个 ARP 缓存表，可以用 arp -a 命令查看。缓存表中的表项有过期时间（一般为 20 分钟），如果 20 分钟内没有再次使用某个表项，则该表项失效，下次还要发 ARP 请求来获得目的主机的硬件地址。想一想，为什么表项要有过期时间而不是一直有效？

ARP 数据报的格式如下所示：



ARP 数据报格式

源 MAC 地址、目的 MAC 地址在以太网首部和 ARP 请求中各出现一次，对于链路层为以太网的情况是多余的，但如果链路层是其它类型的网络则有可能是必要的。硬件类型指链路层网络类型，1 为以太网，协议类型指要转换的地址类型，0x0800 为 IP 地址，后面两个地址长度对于以太网地址和 IP 地址分别为 6 和 4（字节），op 字段为 1 表示 ARP 请求，op 字段为 2 表示 ARP 应答。

看一个具体的例子。

请求帧如下（为了清晰在每行的前面加了字节计数，每行 16 个字节）：
以太网首部（14 字节）
0000: ff ff ff ff ff ff 00 05 5d 61 58 a8 08 06
ARP 帧（28 字节）
0000: 00 01
0010: 08 00 06 04 00 01 00 05 5d 61 58 a8 c0 a8 00 37
0020: 00 00 00 00 00 00 c0 a8 00 02
填充位（18 字节）
0020: 00 77 31 d2 50 10
0030: fd 78 41 d3 00 00 00 00 00 00 00 00

以太网首部：目的主机采用广播地址，源主机的 MAC 地址是 00:05:5d:61:58:a8，上层协议类型 0x0806 表示 ARP。

ARP 帧：硬件类型 0x0001 表示以太网，协议类型 0x0800 表示 IP 协议，硬件地址（MAC 地址）长度为 6，协议地址（IP 地址）长度为 4，op 为 0x0001 表示请求目的主机的 MAC 地址，源主机 MAC 地址为 00:05:5d:61:58:a8，源主机 IP 地址为 c0 a8 00 37（192.168.0.55），目的主机 MAC 地址全 0 待填写，目的主机 IP 地址为 c0 a8 00 02（192.168.0.2）。

由于以太网规定最小数据长度为 46 字节，ARP 帧长度只有 28 字节，因此有 18 字节填充位，填充位的内容没有定义，与具体实现相关。

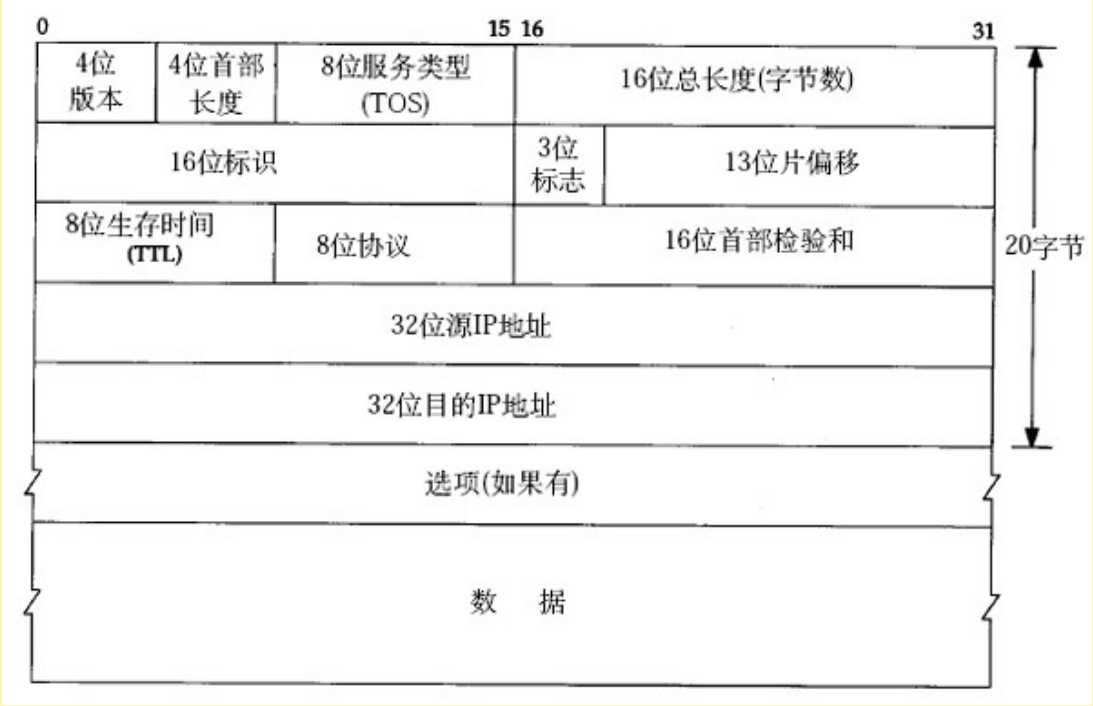
应答帧如下：

以太网首部
0000: 00 05 5d 61 58 a8 00 05 5d a1 b8 40 08 06
ARP 帧
0000: 00 01
0010: 08 00 06 04 00 02 00 05 5d a1 b8 40 c0 a8 00 02
0020: 00 05 5d 61 58 a8 c0 a8 00 37
填充位
0020: 00 77 31 d2 50 10
0030: fd 78 41 d3 00 00 00 00 00 00 00 00
以太网首部：目的主机的 MAC 地址是 00:05:5d:61:58:a8，源主机的 MAC 地址是 00:05:5d:a1:b8:40，上层协议类型 0x0806 表示 ARP。

ARP 帧：硬件类型 0x0001 表示以太网，协议类型 0x0800 表示 IP 协议，硬件地址（MAC 地址）长度为 6，协议地址（IP 地址）长度为 4，op 为 0x0002 表示应答，源主机 MAC 地址为 00:05:5d:a1:b8:40，源主机 IP 地址为 c0 a8 00 02（192.168.0.2），目的主机 MAC 地址为 00:05:5d:61:58:a8，目的主机 IP 地址为 c0 a8 00 37（192.168.0.55）。

思考题：如果源主机和目的主机不在同一网段，ARP 请求的广播帧无法穿过路由器，源主机如何与目的主机通信?

IP 段格式

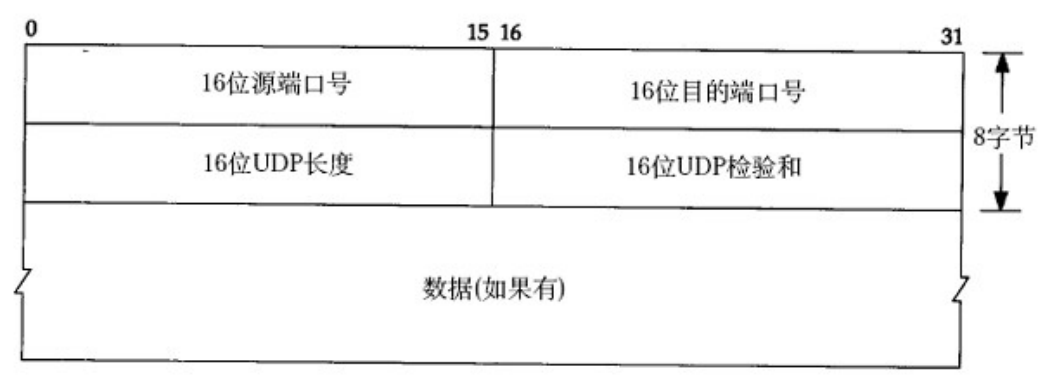


IP 数据报格式

IP 数据报的首部长度和数据长度都是可变长的，**但总是 4 字节的整数倍**。对于 IPv4，4 位版本字段是 4。4 位首部长度的数值是以 4 字节为单位的，**最小值为 5**，也就是说首部长度最小是 4x5=20 字节，也就是不带任何选项的 IP 首部，4 位能表示的最大值是 15，也就是说**首部长度最大是 60 字节**。8 位 TOS 字段有 3 个位用来指定 IP 数据报的优先级（目前已经废弃不用），还有 4 个位表示可选的服务类型（最小延迟、最大吞吐量、最大可靠性、最小成本），还有一个位总是 0。总长度是整个数据报（包括 IP 首部和 IP 层 payload）的字节数。每传一个 IP 数据报，16 位的标识加 1，可用于分片和重新组装数据报。3 位标志和 13 位片偏移用于分片。TTL（Time to live)是这样用的：源主机为数据包设定一个生存时间，比如 64，每过一个路由器就把该值减 1，如果减到 0 就表示路由已经太长了仍然找不到目的主机的网络，就丢弃该包，因此这个生存时间的单位不是秒，而是跳（hop）。协议字段指示上层协议是 TCP、UDP、ICMP 还是 IGMP。然后是校验和，只校验 IP 首部，数据的校验由更高层协议负责。IPv4 的 IP 地址长度为 32 位。

想一想，前面讲了以太网帧中的最小数据长度为 46 字节，不足 46 字节的要用填充字节补上，那么如何界定这 46 字节里前多少个字节是 IP、ARP 或 RARP 数据报而后面是填充字节？

UDP 数据报格式



UDP 数据段

下面分析一帧基于 UDP 的 TFTP 协议帧。

以太网首部
0000: 00 05 5d 67 d0 b1 00 05 5d 61 58 a8 08 00
IP 首部
0000: 45 00
0010: 00 53 93 25 00 00 80 11 25 ec c0 a8 00 37 c0 a8
0020: 00 01
UDP 首部
0020： 05 d4 00 45 00 3f ac 40
TFTP 协议
0020: 00 01 'c'':''\''q'
0030: 'w''e''r''q''.''q''w''e'00 'n''e''t''a''s''c''i'
0040: 'i'00 'b''l''k''s''i''z''e'00 '5''1''2'00 't''i'
0050: 'm''e''o''u''t'00 '1''0'00 't''s''i''z''e'00 '0'
0060: 00 以太网首部：源 MAC 地址是 00:05:5d:61:58:a8，目的 MAC 地址是 00:05:5d:67:d0:b1，上层协议类型 0x0800 表示 IP。

IP 首部：每一个字节 0x45 包含 4 位版本号和 4 位首部长度，版本号为 4，即 IPv4，首部长度为 5，说明 IP 首部不带有选项字段。服务类型为 0，没有使用服务。16 位总长度字段（包括 IP 首部和 IP 层 payload 的长度）为 0x0053，即 83 字节，加上以太网首部 14 字节可知整个帧长度是 97 字节。IP 报标识是 0x9325，标志字段和片偏移字段设置为 0x0000，就是 DF=0 允许分片，MF=0 此数据报没有更多分片，没有分片偏移。TTL 是 0x80，也就是 128。上层协议 0x11 表示 UDP 协议。IP 首部校验和为 0x25ec，源主机 IP 是 c0 a8 00 37（192.168.0.55），目的主机 IP 是 c0 a8 00 01（192.168.0.1）。

UDP 首部：源端口号 0x05d4（1492）是客户端的端口号，目的端口号 0x0045（69）是 TFTP 服务的 well-known 端口号。UDP 报长度为 0x003f，即 63 字节，包括 UDP 首部和 UDP 层 pay-load 的长度。UDP 首部和 UDP 层 payload 的校验和为 0xac40。

TFTP 是基于文本的协议，各字段之间用字节 0 分隔，开头的 00 01 表示请求读取一个文件，接下来的各字段是：

c:\qwerq.qwe
netascii

blksize 512
timeout 10
tsize 0

一般的网络通信都是像 TFTP 协议这样，通信的双方分别是客户端和服务器，客户端主动发起请求（上面的例子就是客户端发起的请求帧），而服务器被动地等待、接收和应答请求。客户端的 IP 地址和端口号唯一标识了该主机上的 TFTP 客户端进程，服务器的 IP 地址和端口号唯一标识了该主机上的 TFTP 服务进程，由于客户端是主动发起请求的一方，它必须知道服务器的 IP 地址和 TFTP 服务进程的端口号，所以，一些常见的网络协议有默认的服务器端口，例如 HTTP 服务默认 TCP 协议的 80 端口，FTP 服务默认 TCP 协议的 21 端口，TFTP 服务默认 UDP 协议的 69 端口（如上例所示）。在使用客户端程序时，必须指定服务器的主机名或 IP 地址，如果不明确指定端口号则采用默认端口，请读者查阅 ftp、tftp 等程序的 man page 了解如何指定端口号。/etc/services 中列出了所有 well-known 的服务端口和对应的传输层协议，这是由 IANA（Internet Assigned Numbers Authority）规定的，其中有些服务既可以用 TCP 也可以用 UDP，为了清晰，IANA 规定这样的服务采用相同的 TCP 或 UDP 默认端口号，而另外一些 TCP 和 UDP 的相同端口号却对应不同的服务。

很多服务有 well-known 的端口号，然而客户端程序的端口号却不必是 well-known 的，往往是每次运行客户端程序时由系统自动分配一个空闲的端口号，用完就释放掉，称为 ephemeral 的端口号，想想这是为什么？

前面提过，UDP 协议不面向连接，也不保证传输的可靠性，例如：

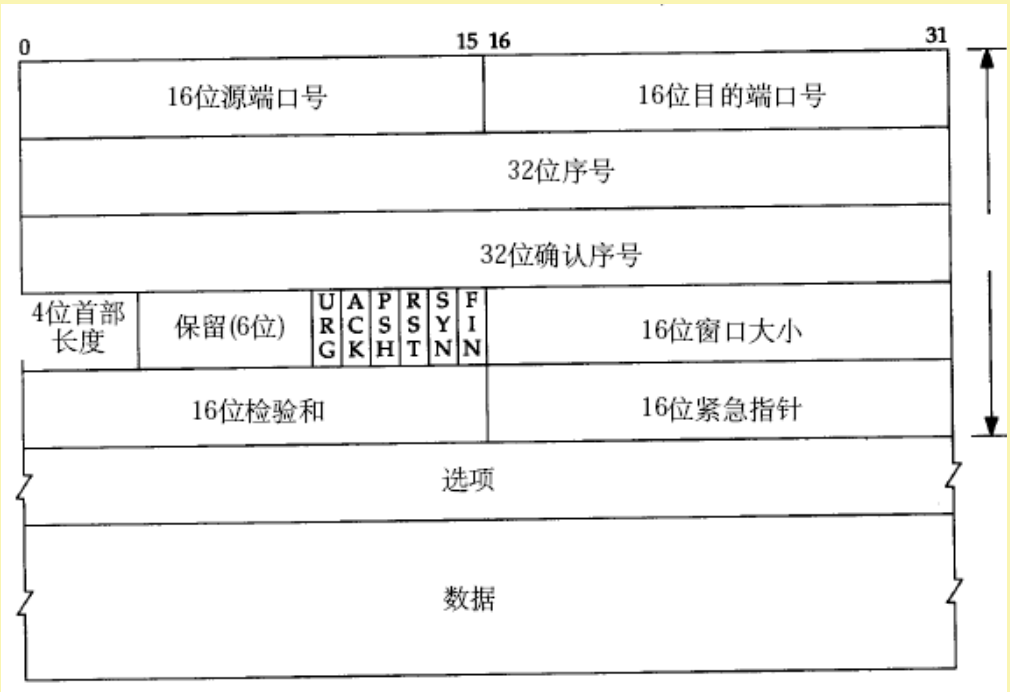
发送端的 UDP 协议层只管把应用层传来的数据封装成段交给 IP 协议层就算完成任务了，如果因为网络故障该段无法发到对方，UDP 协议层也不会给应用层返回任何错误信息。

接收端的 UDP 协议层只管把收到的数据根据端口号交给相应的应用程序就算完成任务了，如果发送端发来多个数据包并且在网络上经过不同的路由，到达接收端时顺序已经错乱了，UDP 协议层也不保证按发送时的顺序交给应用层。

通常接收端的 UDP 协议层将收到的数据放在一个固定大小的缓冲区中等待应用程序来提取和处理，如果应用程序提取和处理的速度很慢，而发送端发送的速度很快，就会丢失数据包，UDP 协议层并不报告这种错误。

因此，使用 UDP 协议的应用程序必须考虑到这些可能的问题并实现适当的解决方案，例如等待应答、超时重发、为数据包编号、流量控制等。一般使用 UDP 协议的应用程序实现都比较简单，只是发送一些对可靠性要求不高的消息，而不发送大量的数据。例如，基于 UDP 的 TFTP 协议一般只用于传送小文件（所以才叫 trivial 的 ftp），而基于 TCP 的 FTP 协议适用于 各种文件的传输。TCP 协议又是如何用面向连接的服务来代替应用程序解决传输的可靠性问题呢。

TCP 数据报格式



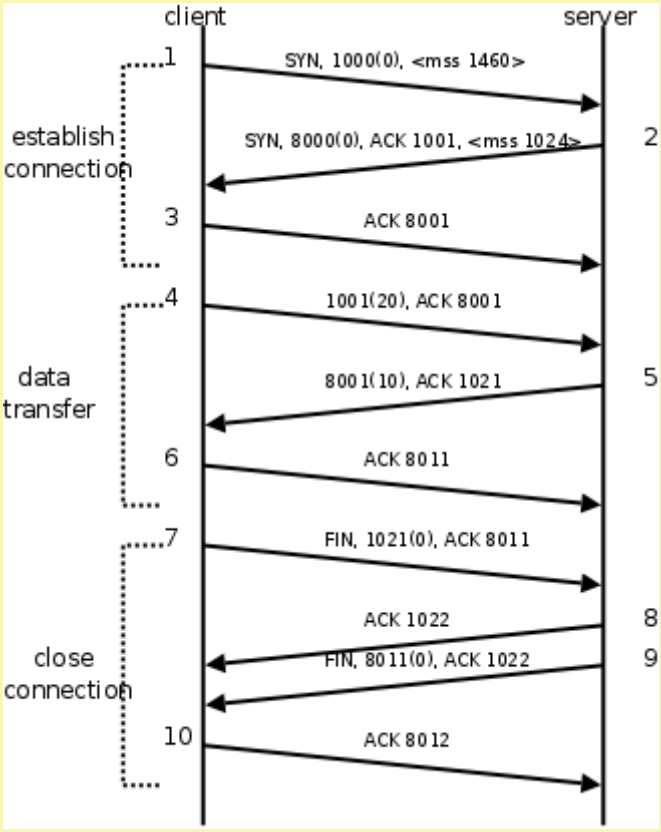
TCP 数据段

与 UDP 协议一样也有源端口号和目的端口号，通讯的双方由 IP 地址和端口号标识。32 位序号、32 位确认序号、窗口大小稍后详细解释。4 位首部长度和 IP 协议头类似，表示 TCP 协议头的长度，以 4 字节为单位，因此 TCP 协议头最长可以是 4x15=60 字节，如果没有选项字段，TCP 协议头最短 20 字节。URG、ACK、PSH、RST、SYN、FIN 是六个控制位，本节稍后将解释 SYN、ACK、FIN、RST 四个位，其它位的解释从略。16 位检验和将 TCP 协议头和数据都计算在内。紧急指针和各种选项的解释从略。

TCP 协议

TCP 通信时序

下图是一次 TCP 通讯的时序图。TCP 连接建立断开。包含大家熟知的**三次握手**和四次握手。



TCP 通讯时序

在这个例子中，首先客户端主动发起连接、发送请求，然后服务器端响应请求，然后客户端主动关闭连接。两条竖线表示通讯的两端，从上到下表示时间的先后顺序，注意，数据从一端传到网络的另一端也需要时间，所以图中的箭头都是斜的。双方发送的段按时间顺序编号为 1-10，各段中的主要信息在箭头上标出，例如段 2 的箭头上标着 SYN, 8000(0), ACK1001, ，表示该段中的 SYN 位置 1，32 位序号是 8000，该段不携带有效载荷（数据字节数为 0），ACK 位置 1，32 位确认序号是 1001，带有一个 **mss**（Maximum Segment Size，**最大报文长度**）选项值为 1024。

建立连接（三次握手）的过程：

1. 客户端发送一个带 SYN 标志的 TCP 报文到服务器。这是三次握手过程中的段 1。

   客户端发出段 1，SYN 位表示连接请求。序号是 1000，这个序号在网络通讯中用作临时的地址，每发一个数据字节，这个序号要加 1，这样在接收端可以根据序号排出数据包的正确顺序，也可以发现丢包的情况，另外，规定 SYN 位和 FIN 位也要占一个序号，这次虽然没发数据，但是由于发了 SYN 位，因此下次再发送应该用序号 1001。mss 表示最大段尺寸，如果一个段太大，封装成帧后超过了链路层的最大帧长度，就必须在 IP 层分片，为了避免这种情况，客户端声明自己的最大段尺寸，建议服务器端发来的段不要超过这个长度。

2. 服务器端回应客户端，是三次握手中的第 2 个报文段，同时带 ACK 标志和 SYN 标志。它表示对刚才客户端 SYN 的回应；同时又发送 SYN 给客户端，询问客户端是否准备好进行数据通讯。

   服务器发出段 2，也带有 SYN 位，同时置 ACK 位表示确认，确认序号是 1001，表示“我接收到序号 1000 及其以前所有的段，请你下次发送序号为 1001 的段”，也就是应答了客户端的连接请求，同时也给客户端发出一个连接请求，同时声明最大尺寸为 1024。

3. 客户必须再次回应服务器端一个 ACK 报文，这是报文段 3。

客户端发出段 3，对服务器的连接请求进行应答，确认序号是 8001。在这个过程中，客户端和服务器分别给对方发了连接请求，也应答了对方的连接请求，其中服务器的请求和应答在一个段中发出，因此一共有三个段用于建立连接，称为“三方握手（three-way-handshake）”。在建立连接的同时，双方协商了一些信息，例如双方发送序号的初始值、最大段尺寸等。

在 TCP 通讯中，如果一方收到另一方发来的段，读出其中的目的端口号，发现本机并没有任何进程使用这个端口，就会应答一个包含 RST 位的段给另一方。例如，服务器并没有任何进程使用 8080 端口，我们却用 telnet 客户端去连接它，服务器收到客户端发来的 SYN 段就会应答一个 RST 段，客户端的 telnet 程序收到 RST 段后报告错误 Connection refused：

```
$ telnet 192.168.0.200 8080
Trying 192.168.0.200...
telnet: Unable to connect to remote host: Connection refused
```

数据传输的过程：

1. 客户端发出段 4，包含从序号 1001 开始的 20 个字节数据。
2. 服务器发出段 5，确认序号为 1021，对序号为 1001-1020 的数据表示确认收到，同时请求发送序号 1021 开始的数据，服务器在应答的同时也向客户端发送从序号 8001 开始的 10 个字节数据，这称为 piggyback。
3. 客户端发出段 6，对服务器发来的序号为 8001-8010 的数据表示确认收到，请求发送序号 8011 开始的数据。

在数据传输过程中，ACK 和确认序号是非常重要的，应用程序交给 TCP 协议发送的数据会暂存在 TCP 层的发送缓冲区中，发出数据包给对方之后，只有收到对方应答的 ACK 段才知道该数据包确实发到了对方，可以从发送缓冲区中释放掉了，如果因为网络故障丢失了数据包或者丢失了对方发回的 ACK 段，经过等待超时后 TCP 协议自动将发送缓冲区中的数据包重发。

关闭连接（四次握手）的过程：

由于 TCP 连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动，一个 TCP 连接在收到一个 FIN 后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

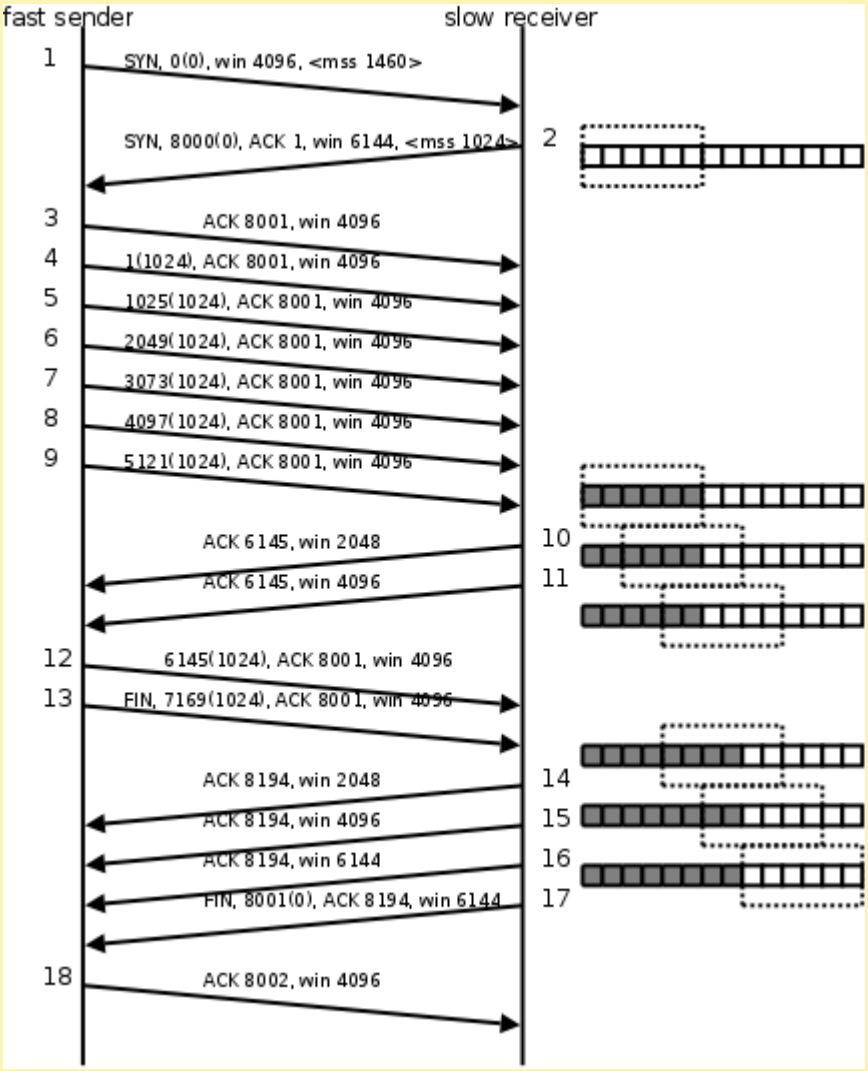
1. 客户端发出段 7，FIN 位表示关闭连接的请求。
2. 服务器发出段 8，应答客户端的关闭连接请求。
3. 服务器发出段 9，其中也包含 FIN 位，向客户端发送关闭连接请求。
4. 客户端发出段 10，应答服务器的关闭连接请求。

建立连接的过程是三方握手，而关闭连接通常需要 4 个段，服务器的应答和关闭连接请求通常不合并在一个段中，因为有连接半关闭的情况，这种情况下客户端关闭连接之后就不能再发送数据给服务器了，但是服务器还可以发送数据给客户端，直到服务器也关闭连接为止。

滑动窗口 (TCP 流量控制)

介绍 UDP 时我们描述了这样的问题：如果发送端发送的速度较快，接收端接收到数据后处理的速度较慢，而接收缓冲区的大小是固定的，就会丢失数据。TCP 协议通过“滑动窗口（Sliding Window）”机制解决这一问题。

看下图的通讯过程：



滑动窗口

1. 发送端发起连接，声明最大段尺寸是 1460，初始序号是 0，窗口大小是 4K，表示“我的接收缓冲区还有 4K 字节空闲，你发的数据不要超过 4K”。接收端应答连接请求，声明最大段尺寸是 1024，初始序号是 8000，窗口大小是 6K。发送端应答，三方握手结束。
2. 发送端发出段 4-9，每个段带 1K 的数据，发送端根据窗口大小知道接收端的缓冲区满了，因此停止发送数据。
3. 接收端的应用程序提走 2K 数据，接收缓冲区又有了 2K 空闲，接收端发出段 10，在应答已收到 6K 数据的同时声明窗口大小为 2K。
4. 接收端的应用程序又提走 2K 数据，接收缓冲区有 4K 空闲，接收端发出段 11，重新声明窗口大小为 4K。
5. 发送端发出段 12-13，每个段带 2K 数据，段 13 同时还包含 FIN 位。
6. 接收端应答接收到的 2K 数据（6145-8192），再加上 FIN 位占一个序号 8193，因此应答序号是 8194，连接处于半关闭状态，接收端同时声明窗口大小为 2K。
7. 接收端的应用程序提走 2K 数据，接收端重新声明窗口大小为 4K。
8. 接收端的应用程序提走剩下的 2K 数据，接收缓冲区全空，接收端重新声明窗口大小为 6K。
9. 接收端的应用程序在提走全部数据后，决定关闭连接，发出段 17 包含 FIN 位，发送端应答，连接完全关闭。

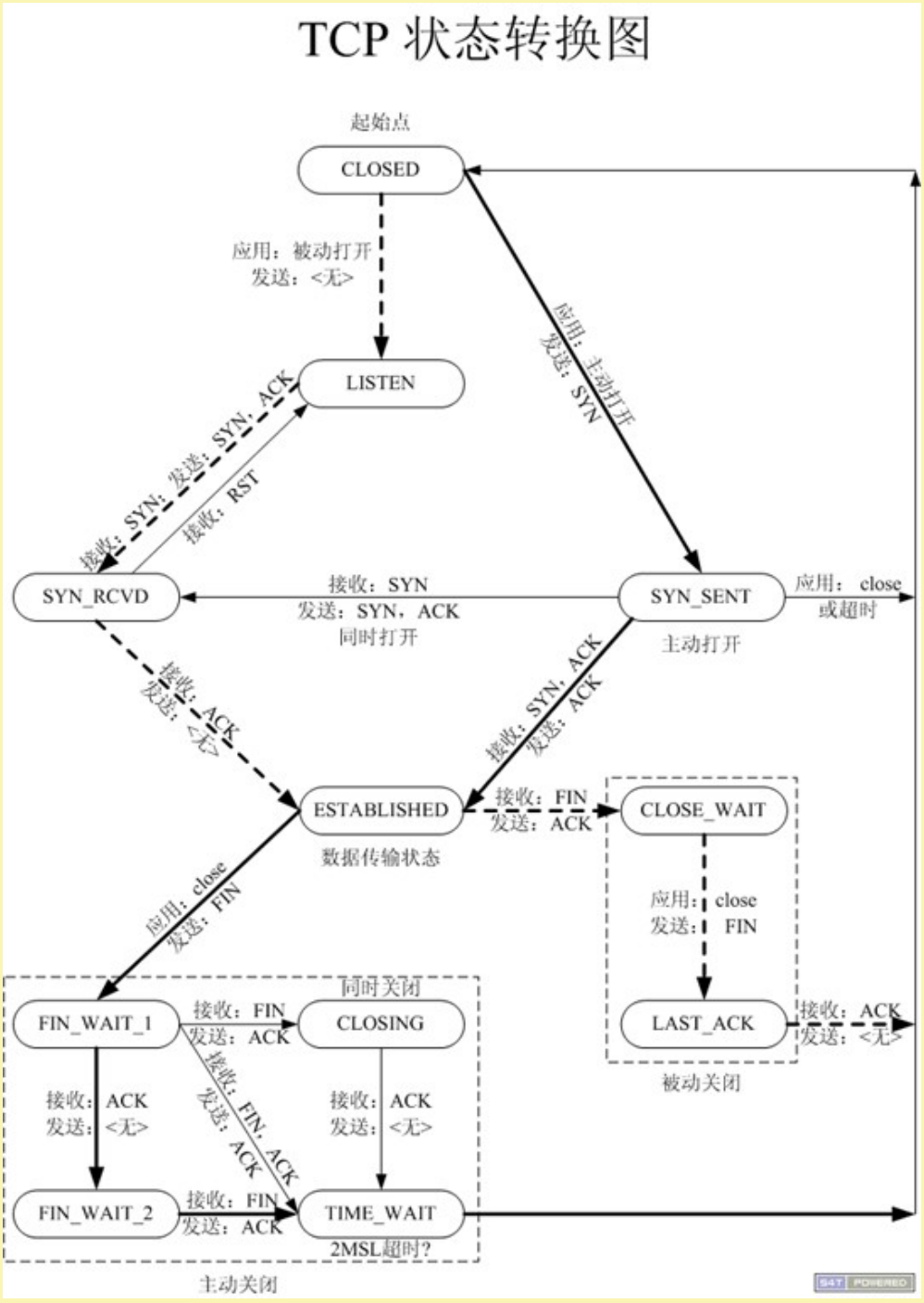
上图在接收端用小方块表示 1K 数据，实心的小方块表示已接收到的数据，虚线框表示接收缓冲区，因此套在

虚线框中的空心小方块表示窗口大小，从图中可以看出，随着应用程序提走数据，虚线框是向右滑动的，因此称为滑动窗口。

从这个例子还可以看出，发送端是一 K 一 K 地发送数据，而接收端的应用程序可以两 K 两 K 地提走数据，当然也有可能一次提走 3K 或 6K 数据，或者一次只提走几个字节的数据。也就是说，应用程序所看到的数据是一个整体，或说是一个流（stream），在底层通讯中这些数据可能被拆成很多数据包来发送，但是一个数据包有多少字节对应用程序是不可见的，因此 TCP 协议是面向流的协议。而 UDP 是面向消息的协议，每个 UDP 段都是一条消息，应用程序必须以消息为单位提取数据，不能一次提取任意字节的数据，这一点和 TCP 是很不同的。

TCP 状态转换

这个图 N 多人都知道，它排除和定位网络或系统故障时大有帮助，但是怎样牢牢地将这张图刻在脑中呢？那么你就一定要对这张图的每一个状态，及转换的过程有深刻的认识，不能只停留在一知半解之中。下面对这张图的 11 种状态详细解析一下，以便加强记忆！不过在这之前，先回顾一下 TCP 建立连接的三次握手过程，以及 关闭连接的四次握手过程。



TCP 状态转换图

CLOSED：表示初始状态。

LISTEN：该状态表示服务器端的某个 SOCKET 处于监听状态，可以接受连接。

SYN_SENT：这个状态与 SYN_RCVD 遥相呼应，当客户端 SOCKET 执行 CONNECT 连接时，它首先发送 SYN 报文，随即进入到了 SYN_SENT 状态，并等待服务端的发送三次握手中的第 2 个报文。SYN_SENT 状态表示客户端已发送 SYN 报文。

SYN_RCVD: 该状态表示接收到 SYN 报文，在正常情况下，这个状态是服务器端的 SOCKET 在建立 TCP 连接时的三次握手会话过程中的一个中间状态，很短暂。此种状态时，当收到客户端的 ACK 报文后，会进入到 ESTABLISHED 状态。

ESTABLISHED：表示连接已经建立。

FIN_WAIT_1: FIN_WAIT_1 和 FIN_WAIT_2 状态的真正含义都是表示等待对方的 FIN 报文。区别是：

FIN_WAIT_1 状态是当 socket 在 ESTABLISHED 状态时，想主动关闭连接，向对方发送了 FIN 报文，此时该 socket 进入到 FIN_WAIT_1 状态。

FIN_WAIT_2 状态是当对方回应 ACK 后，该 socket 进入到 FIN_WAIT_2 状态，正常情况下，对方应马上回应 ACK 报文，所以 FIN_WAIT_1 状态一般较难见到，而 FIN_WAIT_2 状态可用 netstat 看到。

FIN_WAIT_2：主动关闭链接的一方，发出 FIN 收到 ACK 以后进入该状态。称之为半连接或半关闭状态。该状态下的 socket 只能接收数据，不能发。

TIME_WAIT: 表示收到了对方的 FIN 报文，并发送出了 ACK 报文，等 2MSL 后即可回到 CLOSED 可用状态。如果 FIN_WAIT_1 状态下，收到对方同时带 FIN 标志和 ACK 标志的报文时，可以直接进入到 TIME_WAIT 状态，而无须经过 FIN_WAIT_2 状态。

CLOSING: 这种状态较特殊，属于一种较罕见的状态。正常情况下，当你发送 FIN 报文后，按理来说是应该先收到（或同时收到）对方的 ACK 报文，再收到对方的 FIN 报文。但是 CLOSING 状态表示你发送 FIN 报文后，并没有收到对方的 ACK 报文，反而却也收到了对方的 FIN 报文。什么情况下会出现此种情况呢？如果双方几乎在同时 close 一个 SOCKET 的话，那么就出现了双方同时发送 FIN 报文的情况，也即会出现 CLOSING 状态，表示双方都正在关闭 SOCKET 连接。

CLOSE_WAIT: 此种状态表示在等待关闭。当对方关闭一个 SOCKET 后发送 FIN 报文给自己，系统会回应一个 ACK 报文给对方，此时则进入到 CLOSE_WAIT 状态。接下来呢，察看是否还有数据发送给对方，如果没有可以 close 这个 SOCKET，发送 FIN 报文给对方，即关闭连接。所以在 CLOSE_WAIT 状态下，需要关闭连接。

LAST_ACK: 该状态是被动关闭一方在发送 FIN 报文后，最后等待对方的 ACK 报文。当收到 ACK 报文后，即可以进入到 CLOSED 可用状态。

半关闭

当 TCP 链接中 A 发送 FIN 请求关闭，B 端回应 ACK 后（A 端进入 FIN_WAIT_2 状态），B 没有立即发送 FIN 给 A 时，A 方处在半链接状态，此时 A 可以接收 B 发送的数据，但是 A 已不能再向 B 发送数据。

从程序的角度，可以使用 API 来控制实现半连接状态。

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
sockfd: 需要关闭的 socket 的描述符
how:    允许为 shutdown 操作选择以下几种方式:
    SHUT_RD(0):     关闭 sockfd 上的读功能，此选项将不允许 sockfd 进行读操作。
                    该套接字不再接收数据，任何当前在套接字接受缓冲区的数据将被无声的丢弃掉。
    SHUT_WR(1):     关闭 sockfd 的写功能，此选项将不允许 sockfd 进行写操作。进程不能在对此套接字发出写操作。
    SHUT_RDWR(2):   关闭 sockfd 的读写功能。相当于调用 shutdown 两次：首先是以 SHUT_RD,然后以 SHUT_WR。
```

使用 close 中止一个连接，但它只是减少描述符的引用计数，并不直接关闭连接，只有当描述符的引用计数为 0 时才关闭连接。

shutdown 不考虑描述符的引用计数，直接关闭描述符。也可选择中止一个方向的连接，只中止读或只中止写。

注意:

1. 如果有多个进程共享一个套接字，close 每被调用一次，计数减 1，直到计数为 0 时，也就是所用进程都

调用了 close，套接字将被释放。

2. 在多进程中如果一个进程调用了 shutdown(sfd, SHUT_RDWR)后，其它的进程将无法进行通信。但，如果一个进程 close(sfd)将不会影响到其它进程。

2MSL

2MSL (Maximum Segment Lifetime) TIME_WAIT 状态的存在有两个理由：

（1）**让 4 次握手关闭流程更加可靠**；4 次握手的最后一个 ACK 是是由主动关闭方发送出去的，若这个 ACK 丢失，被动关闭方会再次发一个 FIN 过来。若主动关闭方能够保持一个 2MSL 的 TIME_WAIT 状态，则有更大的机会让丢失的 ACK 被再次发送出去。

（2）防止 lost duplicate 对后续新建正常链接的传输造成破坏。lost duplicate 在实际的网络中非常常见，经常是由于路由器产生故障，路径无法收敛，导致一个 packet 在路由器 A，B，C 之间做类似死循环的跳转。IP 头部有个 TTL，限制了一个包在网络中的最大跳数，因此这个包有两种命运，要么最后 TTL 变为 0，在网络中消失；要么 TTL 在变为 0 之前路由器路径收敛，它凭借剩余的 TTL 跳数终于到达目的地。但非常可惜的是 TCP 通过超时重传机制在早些时候发送了一个跟它一模一样的包，并先于它达到了目的地，因此它的命运也就注定被 TCP 协议栈抛弃。

另外一个概念叫做 incarnation connection，指跟上次的 socket pair 一摸一样的新连接，叫做 incarnation of previous connection。lost uplicate 加上 incarnation connection，则会对我们的传输造成致命的错误。

TCP 是流式的，所有包到达的顺序是不一致的，依靠序列号由 TCP 协议栈做顺序的拼接；假设一个 incarnation connection 这时收到的 seq=1000, 来了一个 lost duplicate 为 seq=1000，len=1000, 则 TCP 认为这个 lost duplicate 合法，并存放入了 receive buffer，导致传输出现错误。通过一个 2MSL TIME_WAIT 状态，确保所有的 lost duplicate 都会消失掉，避免对新连接造成错误。

该状态为什么设计在**主动关闭这一方**：

（1）发最后 ACK 的是主动关闭一方。

（2）只要有一方保持 TIME_WAIT 状态，就能起到避免 incarnation connection 在 2MSL 内的重新建立，不需要两方都有。

如何正确对待 2MSL TIME_WAIT?

RFC 要求 socket pair 在处于 TIME_WAIT 时，不能再起一个 incarnation connection。但绝大部分 TCP 实现，强加了更为严格的限制。在 2MSL 等待期间，socket 中使用的本地端口在默认情况下不能再被使用。

若 A 10.234.5.5 : 1234 和 B 10.55.55.60 : 6666 建立了连接，A 主动关闭，那么在 A 端只要 port 为 1234，无论对方的 port 和 ip 是什么，都不允许再起服务。这甚至比 RFC 限制更为严格，RFC 仅仅是要求 socket pair 不一致，而实现当中只要这个 port 处于 TIME_WAIT，就不允许起连接。这个限制对主动打开方来说是无所谓的，因为一般用的是临时端口；但对于被动打开方，一般是 server，就悲剧了，因为 server 一般是熟知端口。比如 http，一般端口是 80，不可能允许这个服务在 2MSL 内不能起来。

解决方案是给服务器的 socket 设置 SO_REUSEADDR 选项，这样的话就算熟知端口处于 TIME_WAIT 状态，在这个端口上依旧可以将服务启动。当然，虽然有了 SO_REUSEADDR 选项，但 sockt pair 这个限制依旧存在。比如上面的例子，A 通过 SO_REUSEADDR 选项依旧在 1234 端口上起了监听，但这时我们若是从 B 通过 6666 端口去连它，TCP 协议会告诉我们连接失败，原因为 Address already in use.

RFC 793 中规定 MSL 为 2 分钟，实际应用中常用的是 30 秒，1 分钟和 2 分钟等。

RFC (Request For Comments)，是一系列以编号排定的文件。收集了有关因特网相关资讯，以及 UNIX 和因特网社群的软件文件。

程序设计中的问题

做一个测试，首先启动 server，然后启动 client，用 Ctrl-C 终止 server，马上再运行 server，运行结果：

```
itcast$ ./server
bind error: Address already in use
```

这是因为，虽然 server 的应用程序终止了，但 TCP 协议层的连接并没有完全断开，因此不能再次监听同样的 server 端口。我们用 netstat 命令查看一下：

```
itcast$ netstat -apn |grep 6666
tcp        1      0 192.168.1.11:38103      192.168.1.11:6666       CLOSE_WAIT  3525/client
tcp        0      0 192.168.1.11:6666       192.168.1.11:38103      FIN_WAIT2   -
```

server 终止时，socket 描述符会自动关闭并发 FIN 段给 client，client 收到 FIN 后处于 CLOSE_WAIT 状态，但是 client 并没有终止，也没有关闭 socket 描述符，因此不会发 FIN 给 server，因此 server 的 TCP 连接处于 FIN_WAIT2 状态。

现在用 Ctrl-C 把 client 也终止掉，再观察现象：

```
itcast$ netstat -apn |grep 6666
tcp        0      0 192.168.1.11:6666       192.168.1.11:38104      TIME_WAIT   -
itcast$ ./server
bind error: Address already in use
```

client 终止时自动关闭 socket 描述符，server 的 TCP 连接收到 client 发的 FIN 段后处于 TIME_WAIT 状态 。TCP 协议规定，**主动关闭连接的一方要处于 TIME_WAIT 状态**，等待两个 MSL（maximum segment lifetime）的时间后才能回到 CLOSED 状态，因为我们先 Ctrl-C 终止了 server，所以 server 是主动关闭连接的一方，在 TIME_WAIT 期间仍然不能再次监听同样的 server 端口。

MSL 在 RFC 1122 中规定为两分钟，但是各操作系统的实现不同，在 Linux 上一般经过半分钟后就可以再次启动 server 了。至于为什么要规定 TIME_WAIT 的时间，可参考 UNP 2.7 节。

端口复用

在 server 的 TCP 连接没有完全断开之前不允许重新监听是不合理的。因为，TCP 连接没有完全断开指的是 connfd（127.0.0.1:6666）没有完全断开，而我们重新监听的是 lis-tenfd（0.0.0.0:6666），虽然是占用同一个端口，但 IP 地址不同，connfd 对应的是与某个客户端通讯的一个具体的 IP 地址，而 listenfd 对应的是 wildcard address。解决这个问题的方法是使用 setsockopt()设置 socket 描述符的选项 SO_REUSEADDR 为 1，表示允许创建端口号相同但 IP 地址不同的多个 socket 描述符。

在 server 代码的 socket()和 bind()调用之间插入如下代码：

```
    int opt = 1;
```

```
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

有关 setsockopt 可以设置的其它选项请参考 UNP 第 7 章。

TCP 异常断开

心跳检测机制

在 TCP 网络通信中，经常会出现客户端和服务器之间的非正常断开，需要实时检测查询链接状态。常用的解决方法就是在程序中加入心跳机制。

Heart-Beat 线程

这个是最常用的简单方法。在接收和发送数据时个人设计一个守护进程(线程)，定时发送 Heart-Beat 包，客户端/服务器收到该小包后，立刻返回相应的包即可检测对方是否实时在线。

该方法的好处是通用，但缺点就是会改变现有的通讯协议！大家一般都是使用业务层心跳来处理，主要是灵活可控。

UNIX 网络编程不推荐使用 SO_KEEPALIVE 来做心跳检测，还是在业务层以心跳包做检测比较好，也方便控制。

设置 TCP 属性

SO_KEEPALIVE 保持连接检测对方主机是否崩溃，避免（服务器）永远阻塞于 TCP 连接的输入。设置该选项后，如果 2 小时内在此套接口的任一方向都没有数据交换，TCP 就自动给对方发一个保持存活探测分节(keepalive probe)。这是一个对方必须响应的 TCP 分节.它会导致以下三种情况：对方接收一切正常：以期望的 ACK 响应。2 小时后，TCP 将发出另一个探测分节。对方已崩溃且已重新启动：以 RST 响应。套接口的待处理错误被置为 ECONNRESET，套接 口本身则被关闭。对方无任何响应：源自 berkeley 的 TCP 发送另外 8 个探测分节，相隔 75 秒一个，试图得到一个响应。在发出第一个探测分节 11 分钟 15 秒后若仍无响应就放弃。套接口的待处理错误被置为 ETIMEOUT，套接口本身则被关闭。如 ICMP 错误是“host unreachable(主机不可达)”，说明对方主机并没有崩溃，但是不可达，这种情况下待处理错误被置为 EHOSTUNREACH。

根据上面的介绍我们可以知道对端以一种非优雅的方式断开连接的时候，我们可以设置 SO_KEEPALIVE 属性使得我们在 2 小时以后发现对方的 TCP 连接是否依然存在。

```
keepAlive = 1;
setsockopt(listenfd, SOL_SOCKET, SO_KEEPALIVE, (void*)&keepAlive, sizeof(keepAlive));
```

如果我们不能接受如此之长的等待时间，从 TCP-Keepalive-HOWTO 上可以知道一共有两种方式可以设置，一种是修改内核关于网络方面的 配置参数，另外一种就是 SOL_TCP 字段的 TCP_KEEPIDLE，TCP_KEEPINTVL， TCP_KEEPCNT 三个选项。

1. The tcp_keepidle parameter specifies the interval of inactivity that causes TCP to generate a KEEPALIVE transmission for an application that requests them. tcp_keepidle defaults to 14400 (two hours).

/*开始首次 KeepAlive 探测前的 TCP 空闭时间 */

2. The tcp_keepintvl parameter specifies the interval between the nine retriesthat are attempted if a KEEPALIVE transmission is not acknowledged. tcp_keep ntvldefaults to 150 (75 seconds).

   /* 两次 KeepAlive 探测间的时间间隔 */

3. The tcp_keepcnt option specifies the maximum number of keepalive probes tobe sent. The value of TCP_KEEPCNT is an integer value between 1 and n, where n s the value of the systemwide tcp_keepcnt parameter.

   /* 判定断开前的 KeepAlive 探测次数*/

```
int keepIdle = 1000;
int keepInterval = 10;
int keepCount = 10;

Setsockopt(listenfd, SOL_TCP, TCP_KEEPIDLE, (void *)&keepIdle, sizeof(keepIdle));
Setsockopt(listenfd, SOL_TCP,TCP_KEEPINTVL, (void *)&keepInterval, sizeof(keepInterval));
Setsockopt(listenfd,SOL_TCP, TCP_KEEPCNT, (void *)&keepCount, sizeof(keepCount));
```

SO_KEEPALIVE 设置空闲 2 小时才发送一个“保持存活探测分节”，不能保证实时检测。对于判断网络断开时间太长，对于需要及时响应的程序不太适应。

当然也可以修改时间间隔参数，但是会影响到所有打开此选项的套接口！关联了完成端口的 socket 可能会忽略掉该套接字选项。

网络名词术语解析(自行阅读扫盲)

路由(route)

路由（名词）

数据包从源地址到目的地址所经过的路径，由一系列路由节点组成。

路由（动词）

某个路由节点为数据包选择投递方向的选路过程。

路由器工作原理

路由器（Router）是连接因特网中各局域网、广域网的设备，它会根据信道的情况自动选择和设定路由，以最佳路径，按前后顺序发送信号的设备。

传统地，路由器工作于 OSI 七层协议中的第三层，其主要任务是接收来自一个网络接口的数据包，根据其中所含的目的地址，决定转发到下一个目的地址。因此，路由器首先得在转发路由表中查找它的目的地址，若找到了目的地址，就在数据包的帧格前添加下一个 MAC 地址，同时 IP 数据包头的 TTL（Time To Live）域也开始减数，

并重新计算校验和。当数据包被送到输出端口时，它需要按顺序等待，以便被传送到输出链路上。

路由器在工作时能够按照某种路由通信协议查找设备中的路由表。如果到某一特定节点有一条以上的路径，则基本预先确定的路由准则是选择最优（或最经济）的传输路径。由于各种网络段和其相互连接情况可能会因环境变化而变化，因此路由情况的信息一般也按所使用的路由信息协议的规定而定时更新。

网络中，每个路由器的基本功能都是按照一定的规则来动态地更新它所保持的路由表，以便保持路由信息的有效性。为了便于在网络间传送报文，路由器总是先按照预定的规则把较大的数据分解成适当大小的数据包，再将这些数据包分别通过相同或不同路径发送出去。当这些数据包按先后秩序到达目的地后，再把分解的数据包按照一定顺序包装成原有的报文形式。路由器的分层寻址功能是路由器的重要功能之一，该功能可以帮助具有很多节点站的网络来存储寻址信息，同时还能在网络间截获发送到远地网段的报文，起转发作用；选择最合理的路由，引导通信也是路由器基本功能；多协议路由器还可以连接使用不同通信协议的网络段，成为不同通信协议网络段之间的通信平台。

路由和交换之间的主要区别就是交换发生在 OSI 参考模型第二层（数据链路层），而路由发生在第三层，即网络层。这一区别决定了路由和交换在移动信息的过程 中需使用不同的控制信息，所以两者实现各自功能的方式是不同的。

路由表(Routing Table)

在计算机网络中，路由表或称路由择域信息库（RIB）是一个存储在路由器或者联网计算机中的电子表格（文件）或类数据库。路由表存储着指向特定网络地址的路径。

路由条目

路由表中的一行，每个条目主要由目的网络地址、子网掩码、下一跳地址、发送接口四部分组成，如果要发送的数据包的目的网络地址匹配路由表中的某一行，就按规定的接口发送到下一跳地址。

缺省路由条目

路由表中的最后一行，主要由下一跳地址和发送接口两部分组成，当目的地址与路由表中其它行都不匹配时，就按缺省路由条目规定的接口发送到下一跳地址。

路由节点

一个具有路由能力的主机或路由器，它维护一张路由表，通过查询路由表来决定向哪个接口发送数据包。

以太网交换机工作原理

以太网交换机是基于以太网传输数据的交换机，以太网采用共享总线型传输媒体方式的局域网。以太网交换机的结构是每个端口都直接与主机相连，并且一般都工作在全双工方式。交换机能同时连通许多对端口，使每一对相互通信的主机都能像独占通信媒体那样，进行无冲突地传输数据。

以太网交换机工作于 OSI 网络参考模型的第二层（即数据链路层），是一种基于 MAC（Media Access Control，介质访问控制）地址识别、完成以太网数据帧转发的网络设备。

hub 工作原理

集线器实际上就是中继器的一种，其区别仅在于集线器能够提供更多的端口服务，所以集线器又叫多口中继器。

集线器功能是随机选出某一端口的设备，并让它独占全部带宽，与集线器的上联设备（交换机、路由器或服务器等）进行通信。从 Hub 的工作方式可以看出，它在网络中只起到信号放大和重发作用，其目的是扩大网络的传输范围，而不具备信号的定向传送能力，是一个标准的共享式设备。其次是 Hub 只与它的上联设备(如上层 Hub、交换机或服务器)进行通信，同层的各端口之间不会直接进行通信，而是通过上联设备再将信息广播到所有端口上。 由此可见，即使是在同一 Hub 的不同两个端口之间进行通信，都必须要经过两步操作：

第一步是将信息上传到上联设备；

第二步是上联设备再将该信息广播到所有端口上。

半双工/全双工

Full-duplex（全双工）全双工是在通道中同时双向数据传输的能力。

Half-duplex（半双工）在通道中同时只能沿着一个方向传输数据。

DNS 服务器

DNS 是域名系统 (Domain Name System) 的缩写，是因特网的一项核心服务，它作为可以将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便的访问互联网，而不用去记住能够被机器直接读取的 IP 地址串。

它是由解析器以及域名服务器组成的。域名服务器是指保存有该网络中所有主机的域名和对应 IP 地址，并具有将域名转换为 IP 地址功能的服务器。

局域网(LAN)

local area network，一种覆盖一座或几座大楼、一个校园或者一个厂区等地理区域的小范围的计算机网。

1. 覆盖的地理范围较小，只在一个相对独立的局部范围内联，如一座或集中的建筑群内。
2. 使用专门铺设的传输介质进行联网，数据传输速率高（10Mb/s～10Gb/s）
3. 通信延迟时间短，可靠性较高
4. 局域网可以支持多种传输介质

广域网(WAN)

wide area network，一种用来实现不同地区的局域网或城域网的互连，可提供不同地区、城市和国家之间的计算机通信的远程计算机网。

覆盖的范围比局域网（LAN）和城域网（MAN）都广。广域网的通信子网主要使用分组交换技术。

广域网的通信子网可以利用公用分组交换网、卫星通信网和无线分组交换网，它将分布在不同地区的局域网或计算机系统互连起来，达到资源共享的目的。如互联网是世界范围内最大的广域网。

1. 适应大容量与突发性通信的要求；
2. 适应综合业务服务的要求；
3. 开放的设备接口与规范化的协议；
4. 完善的通信服务与网络管理。

端口

逻辑意义上的端口，一般是指 TCP/IP 协议中的端口，端口号的范围从 0 到 65535，比如用于浏览网页服务的 80 端口，用于 FTP 服务的 21 端口等等。

1. 端口号小于 256 的定义为常用端口，服务器一般都是通过常用端口号来识别的。
2. 客户端只需保证该端口号在本机上是惟一的就可以了。客户端口号因存在时间很短暂又称临时端口号；
3. 大多数 TCP/IP 实现给临时端口号分配 1024—5000 之间的端口号。大于 5000 的端口号是为其他服务器预留的。

我们应该在自定义端口时，避免使用 well-known 的端口。如：80、21 等等。

MTU

MTU:通信术语 最大传输单元（Maximum Transmission Unit，MTU）

是指一种通信协议的某一层上面所能通过的最大数据包大小（以字节为单位）。最大传输单元这个参数通常与通信接口有关（网络接口卡、串口等）。

以下是一些协议的 MTU：

FDDI 协议：4352 字节
以太网（Ethernet）协议：1500 字节
PPPoE（ADSL）协议：1492 字节
X.25 协议（Dial Up/Modem）： 576 字节
Point-to-Point：4470 字节

常见网络知识面试题

1. TCP 如何建立链接
2. TCP 如何通信
3. TCP 如何关闭链接
4. 什么是滑动窗口
5. 什么是半关闭
6. 局域网内两台机器如何利用 TCP/IP 通信
7. internet 上两台主机如何进行通信
8. 如何在 internet 上识别唯一一个进程

   答：通过“IP 地址+端口号”来区分不同的服务
9. 为什么说 TCP 是可靠的链接，UDP 不可靠
10. 路由器和交换机的区别
11. 点到点，端到端

Socket 编程

套接字概念

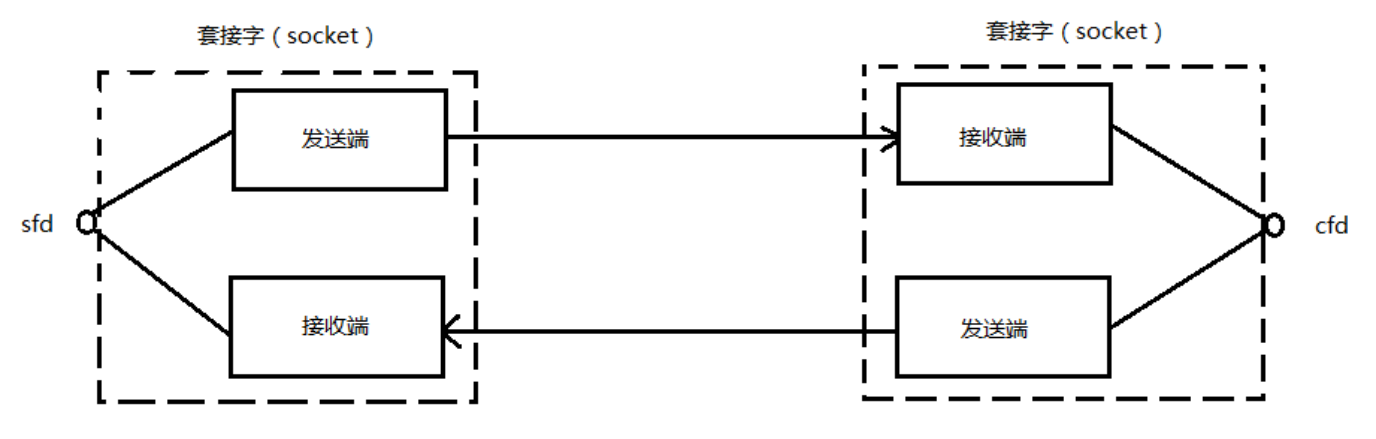
Socket 本身有“插座”的意思，在 Linux 环境下，用于表示进程间网络通信的特殊文件类型。本质为内核借助缓冲区形成的伪文件。

既然是文件，那么理所当然的，我们可以使用文件描述符引用套接字。与管道类似的，Linux 系统将其封装成文件的目的是为了统一接口，使得读写套接字和读写文件的操作一致。区别是管道主要应用于本地进程间通信，而套接字多应用于网络进程间数据的传递。

套接字的内核实现较为复杂，不宜在学习初期深入学习。

在 TCP/IP 协议中，“IP 地址+TCP 或 UDP 端口号”唯一标识网络通讯中的一个进程。“IP 地址+端口号”就对应一个 socket。欲建立连接的两个进程各自有一个 socket 来标识，那么这两个 socket 组成的 socket pair 就唯一标识一个连接。因此可以用 Socket 来描述网络连接的一对一关系。

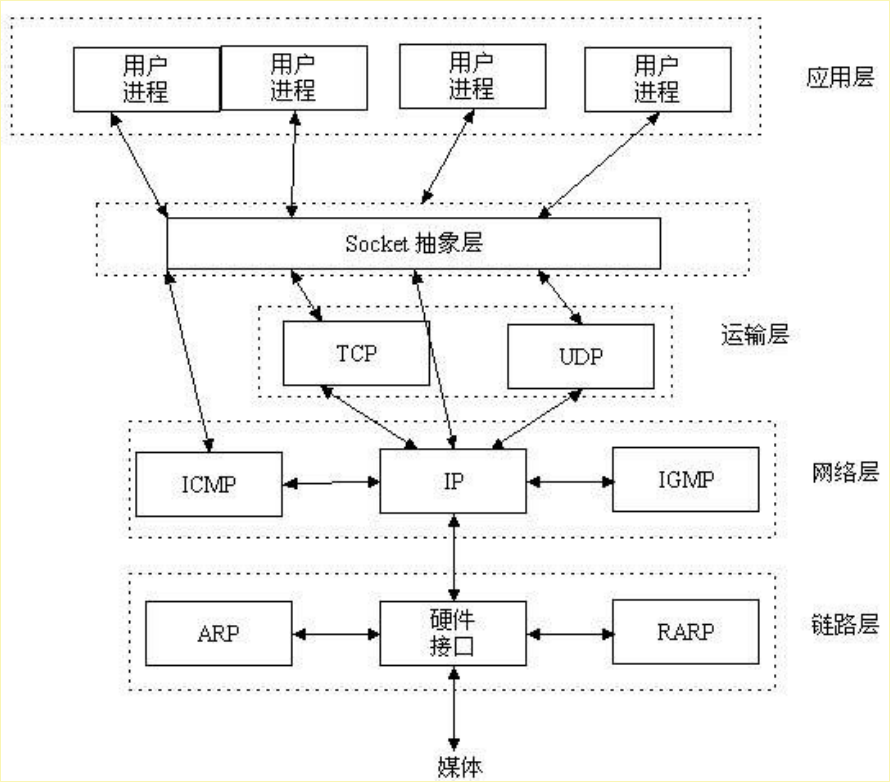
套接字通信原理如下图所示：



套接字通讯原理示意

在网络通信中，套接字一定是成对出现的。一端的发送缓冲区对应对端的接收缓冲区。我们使用同一个文件描述符索发送缓冲区和接收缓冲区。

TCP/IP 协议最早在 BSD UNIX 上实现，为 TCP/IP 协议设计的应用层编程接口称为 socket API。本章的主要内容是 socket API，主要介绍 TCP 协议的函数接口，最后介绍 UDP 协议和 UNIX Domain Socket 的函数接口。



网络编程接口

预备知识

网络字节序

我们已经知道，内存中的多字节数据相对于内存地址有大端和小端之分，磁盘文件中的多字节数据相对于文件中的偏移地址也有大端小端之分。网络数据流同样有大端小端之分，那么如何定义网络数据流的地址呢？发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出，接收主机把从网络上接到的字节依次保存在接收缓冲区中，也是按内存地址从低到高的顺序保存，因此，网络数据流的地址应这样规定：先发出的数据是低地址，后发出的数据是高地址。

TCP/IP 协议规定，网络数据流应采用大端字节序，即低地址高字节。例如上一节的 UDP 段格式，地址 0-1 是 16 位的源端口号，如果这个端口号是 1000（0x3e8），则地址 0 是 0x03，地址 1 是 0xe8，也就是先发 0x03，再发 0xe8，这 16 位在发送主机的缓冲区中也应该是低地址存 0x03，高地址存 0xe8。但是，如果发送主机是小端字节序的，这 16 位被解释成 0xe803，而不是 1000。因此，发送主机把 1000 填到发送缓冲区之前需要做字节序的转换。同样地，接收主机如果是小端字节序的，接到 16 位的源端口号也要做字节序的转换。如果主机是大端字节序的，发送和接收都不需要做转换。同理，32 位的 IP 地址也要考虑网络字节序和主机字节序的问题。

为使网络程序具有可移植性，使同样的 C 代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做**网络字节序和主机字节序的转换**。

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

h 表示 host，n 表示 network，l 表示 32 位长整数，s 表示 16 位短整数。

如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。

IP 地址转换函数

早期：

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
只能处理 IPv4 的 ip 地址

不可重入函数
注意参数是 struct in_addr

现在：

```
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

支持 IPv4 和 IPv6
可重入函数

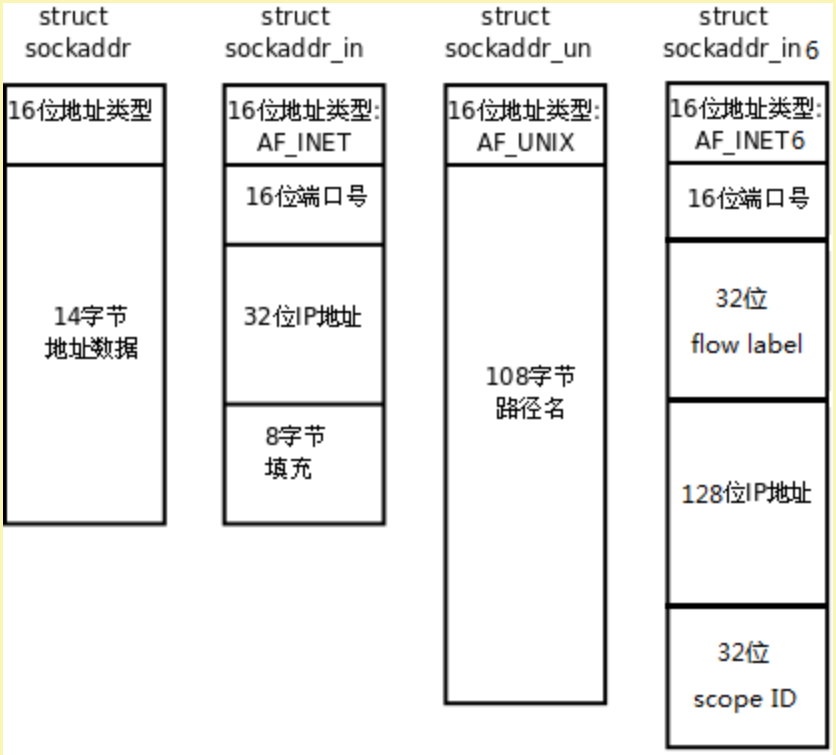
其中 inet_pton 和 inet_ntop 不仅可以转换 IPv4 的 in_addr，还可以转换 IPv6 的 in6_addr。

因此函数接口是 void *addrptr。

sockaddr 数据结构

strcut sockaddr 很多网络编程函数诞生早于 IPv4 协议，那时候都使用的是 sockaddr 结构体,为了向前兼容，现在 sockaddr 退化成了（void *）的作用，传递一个地址给函数，至于这个函数是 sockaddr_in 还是 sockaddr_in6，由地址族确定，然后函数内部再强制类型转化为所需的地址类型。

可参看 man 7 ip。



sockaddr 数据结构

```
struct sockaddr {
    sa_family_t sa_family;      /* address family, AF_xxx */
    char sa_data[14];           /* 14 bytes of protocol address */
};
```

使用 sudo grep -r "struct sockaddr_in {" /usr 命令可查看到 struct sockaddr_in 结构体的定义。一般其默认的存储位置：/usr/include/linux/in.h 文件中。

```
struct sockaddr_in {
    __kernel_sa_family_t sin_family;          /* Address family */    地址结构类型
    __be16 sin_port;                          /* Port number */       端口号
    struct in_addr sin_addr;                  /* Internet address */  IP 地址
    /* Pad to size of `struct sockaddr'. */
    unsigned char __pad[__SOCK_SIZE__ - sizeof(short int) -
    sizeof(unsigned short int) - sizeof(struct in_addr)];
};

struct in_addr {                      /* Internet address. */
    __be32 s_addr;
};

struct sockaddr_in6 {
    unsigned short int sin6_family;        /* AF_INET6 */
    __be16 sin6_port;                  /* Transport layer port # */
    __be32 sin6_flowinfo;              /* IPv6 flow information */
    struct in6_addr sin6_addr;         /* IPv6 address */
    __u32 sin6_scope_id;               /* scope id (new in RFC2553) */
};
```

```
struct in6_addr {
    union {
        __u8 u6_addr8[16];
        __be16 u6_addr16[8];
        __be32 u6_addr32[4];
    } in6_u;
    #define s6_addr         in6_u.u6_addr8
    #define s6_addr16       in6_u.u6_addr16
    #define s6_addr32       in6_u.u6_addr32
};

#define UNIX_PATH_MAX 108
    struct sockaddr_un {
    __kernel_sa_family_t sun_family;   /* AF_UNIX */
    char sun_path[UNIX_PATH_MAX];  /* pathname */
};
```

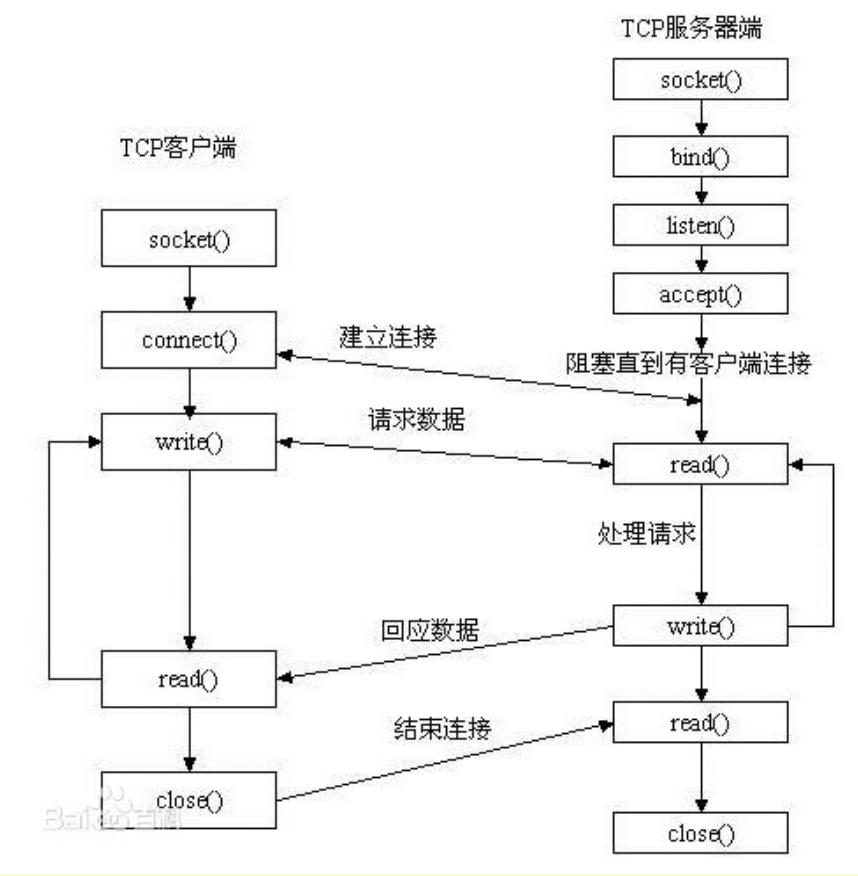
Pv4 和 IPv6 的地址格式定义在 netinet/in.h 中，IPv4 地址用 sockaddr_in 结构体表示，包括 16 位端口号和 32 位 IP 地址，IPv6 地址用 sockaddr_in6 结构体表示，包括 16 位端口号、128 位 IP 地址和一些控制字段。UNIX Domain Socket 的地址格式定义在 sys/un.h 中，用 sock-addr_un 结构体表示。各种 socket 地址结构体的开头都是相同的，前 16 位表示整个结构体的长度（并不是所有 UNIX 的实现都有长度字段，如 Linux 就没有），后 16 位表示地址类型。IPv4、IPv6 和 Unix Domain Socket 的地址类型分别定义为常数 AF_INET、AF_INET6、AF_UNIX。这样，只要取得某种 sockaddr 结构体的首地址，不需要知道具体是哪种类型的 sockaddr 结构体，就可以根据地址

类型字段确定结构体中的内容。因此，socket API 可以接受各种类型的 sockaddr 结构体指针做参数，例如 bind、accept、connect 等函数，这些函数的参数应该设计成 void *类型以便接受各种类型的指针，但是 sock API 的实现早于 ANSI C 标准化，那时还没有 void *类型，因此这些函数的参数都用 struct sockaddr *类型表示，在传递参数之前要强制类型转换一下，例如：

```
struct sockaddr_in servaddr;
bind(listen_fd, (struct sockaddr *)&servaddr, sizeof(servaddr));      /* initialize servaddr */
```

网络套接字函数

socket 模型创建流程图



socket API

socket 函数

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
domain:
```

```
    AF_INET 这是大多数用来产生 socket 的协议，使用 TCP 或 UDP 来传输，用 IPv4 的地址
    AF_INET6 与上面类似，不过是来用 IPv6 的地址
    AF_UNIX 本地协议，使用在 Unix 和 Linux 系统上，一般都是当客户端和服务器在同一台及其上的时候使用
type:
    SOCK_STREAM 这个协议是按照顺序的、可靠的、数据完整的基于字节流的连接。这是一个使用最多的 socket 类型，这个 socket 是使用 TCP 来进行传输。
    SOCK_DGRAM 这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用 UDP 来进行它的连接。
    SOCK_SEQPACKET 该协议是双线路的、可靠的连接，发送固定长度的数据包进行传输。必须把这个包完整的接受才能进行读取。
    SOCK_RAW socket 类型提供单一的网络访问，这个 socket 类型使用 ICMP 公共协议。（ping、traceroute 使用该协议）
    SOCK_RDM 这个类型是很少使用的，在大部分的操作系统上没有实现，它是提供给数据链路层使用，不保证数据包的顺序
protocol:
    传 0 表示使用默认协议。
返回值:
    成功：返回指向新创建的 socket 的文件描述符，失败：返回-1，设置 errno
```

socket()打开一个网络通讯端口，如果成功的话，就像 open()一样返回一个文件描述符，应用程序可以像读写文件一样用 read/write 在网络上收发数据，如果 socket()调用出错则返回-1。对于 IPv4，domain 参数指定为 AF_INET。对于 TCP 协议，type 参数指定为 SOCK_STREAM，表示面向流的传输协议。如果是 UDP 协议，则 type 参数指定为 SOCK_DGRAM，表示面向数据报的传输协议。protocol 参数的介绍从略，指定为 0 即可。

bind 函数

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
sockfd:
    socket 文件描述符
addr:
    构造出 IP 地址加端口号
addrlen:
    sizeof(addr)长度
返回值:
    成功返回 0，失败返回-1, 设置 errno
```

服务器程序所监听的网络地址和端口号通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用 bind 绑定一个固定的网络地址和端口号。

bind()的作用是将参数 sockfd 和 addr 绑定在一起，使 sockfd 这个用于网络通讯的文件描述符监听 addr 所描述的地址和端口号。前面讲过，struct sockaddr *是一个通用指针类型，addr 参数实际上可以接受多种协议的 sockaddr 结构体，而它们的长度各不相同，所以需要第三个参数 addrlen 指定结构体的长度。如：

```
struct sockaddr_in servaddr;
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(6666);
```

首先将整个结构体清零，然后设置地址类型为 AF_INET，**网络地址为 INADDR_ANY，这个宏表示本地的任意 IP 地址**，因为服务器可能有多个网卡，每个网卡也可能绑定多个 IP 地址，这样设置可以在所有的 IP 地址上监听，直到与某个客户端建立了连接时才确定下来到底用哪个 IP 地址，端口号为 6666。

listen 函数

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int listen(int sockfd, int backlog);
sockfd:
    socket 文件描述符
backlog:
    排队建立 3 次握手队列和刚刚建立 3 次握手队列的链接数和
```

查看系统默认 backlog

```
cat /proc/sys/net/ipv4/tcp_max_syn_backlog
```

典型的服务器程序可以同时服务于多个客户端，当有客户端发起连接时，服务器调用的 accept()返回并接受这个连接，如果有大量的客户端发起连接而服务器来不及处理，尚未 accept 的客户端就处于连接等待状态，listen()声明 sockfd 处于监听状态，并且最多允许有 backlog 个客户端处于连接待状态，如果接收到更多的连接请求就忽略。listen()成功返回 0，失败返回-1。

accept 函数

```
#include <sys/types.h>      /* See NOTES */
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
sockdf:
    socket 文件描述符
addr:
    传出参数，返回链接客户端地址信息，含 IP 地址和端口号
addrlen:
    传入传出参数（值-结果）,传入 sizeof(addr)大小，函数返回时返回真正接收到地址结构体的大小
返回值:
    成功返回一个新的 socket 文件描述符，用于和客户端通信，失败返回-1，设置 errno
```

三方握手完成后，服务器调用 accept()接受连接，如果服务器调用 accept()时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。addr 是一个传出参数，accept()返回时传出客户端的地址和端口号。addrlen 参数是一个传入传出参数（value-result argument），传入的是调用者提供的缓冲区 addr 的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给 addr 参数传 NULL，表示不关心客户端的地址。

我们的服务器程序结构是这样的：

```
while (1) {
```

```
    cliaddr_len = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
    n = read(connfd, buf, MAXLINE);
    ......
    close(connfd);
}
```

整个是一个 while 死循环，每次循环处理一个客户端连接。由于 cliaddr_len 是传入传出参数，每次调用 accept()之前应该重新赋初值。accept()的参数 listenfd 是先前的监听文件描述符，而 accept()的返回值是另外一个文件描述符 connfd，之后与客户端之间就通过这个 connfd 通讯，最后关闭 connfd 断开连接，而不关闭 listenfd，再次回到循环开头 listenfd 仍然用作 accept 的参数。accept()成功返回一个文件描述符，出错返回-1。

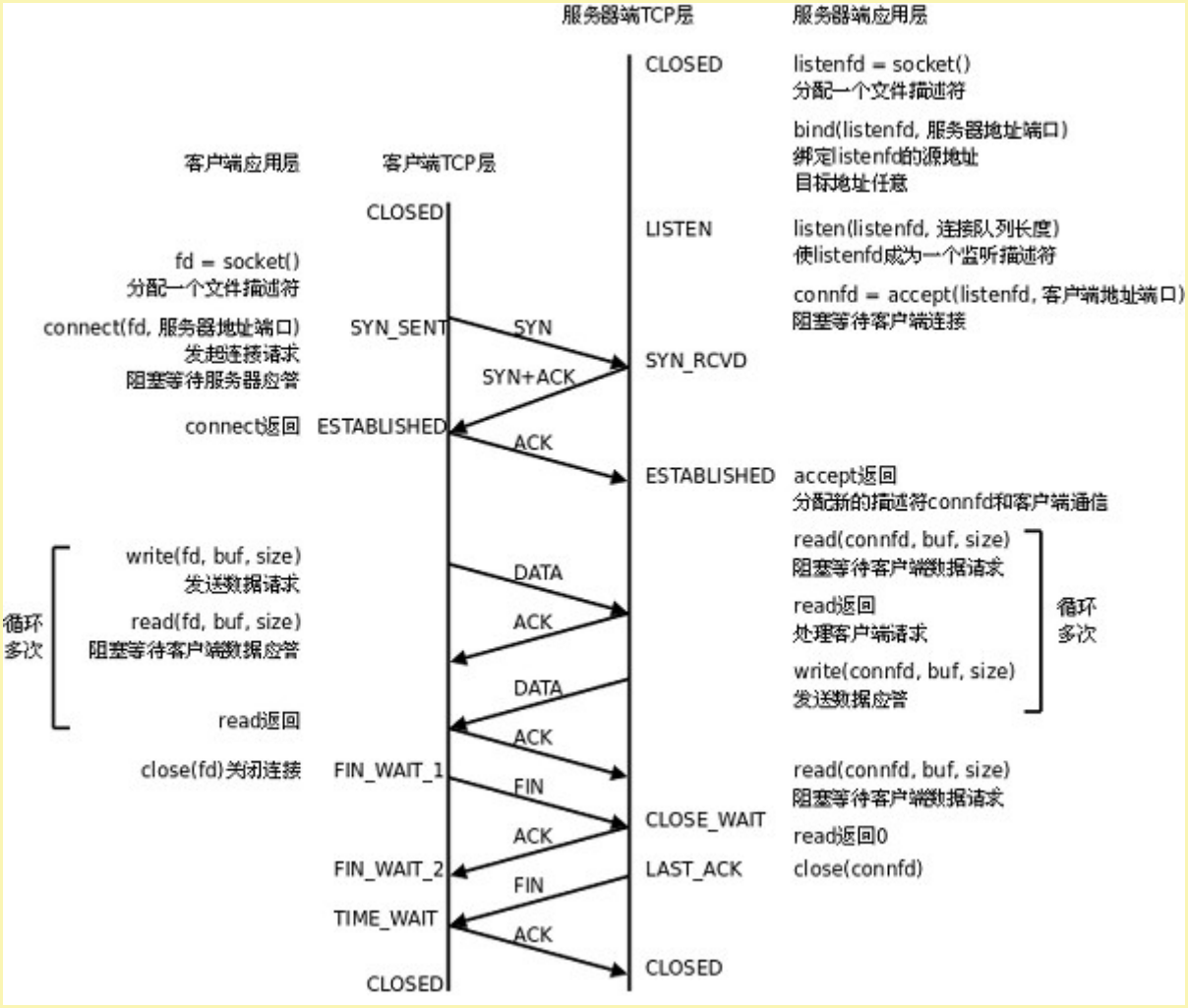
connect 函数

```
#include <sys/types.h>                /* See NOTES */
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
sockdf:
    socket 文件描述符
addr:
    传入参数，指定服务器端地址信息，含 IP 地址和端口号
addrlen:
    传入参数,传入 sizeof(addr)大小
返回值:
    成功返回 0，失败返回-1，设置 errno
```

客户端需要调用 connect()连接服务器，connect 和 bind 的参数形式一致，区别在于 bind 的参数是自己的地址，而 connect 的参数是对方的地址。connect()成功返回 0，出错返回-1。

C/S 模型-TCP

下图是基于 TCP 协议的客户端/服务器程序的一般流程：



TCP 协议通讯流程

服务器调用 socket()、bind()、listen()完成初始化后，调用 accept()阻塞等待，处于监听端口的状态，客户端调用 socket()初始化后，调用 connect()发出 SYN 段并阻塞等待服务器应答，服务器应答一个 SYN-ACK 段，客户端收到后从 connect()返回，同时应答一个 ACK 段，服务器收到后从 accept()返回。

数据传输的过程：

建立连接后，TCP 协议提供全双工的通信服务，但是一般的客户端/服务器程序的流程是由客户端主动发起请求，服务器被动处理请求，一问一答的方式。因此，服务器从 accept()返回后立刻调用 read()，读 socket 就像读管道一样，如果没有数据到达就阻塞等待，这时客户端调用 write()发送请求给服务器，服务器收到后从 read()返回，对客户端的请求进行处理，在此期间客户端调用 read()阻塞等待服务器的应答，服务器调用 write()将处理结果发回给客户端，再次调用 read()阻塞等待下一条请求，客户端收到后从 read()返回，发送下一条请求，如此循环下去。

如果客户端没有更多的请求了，就调用 close()关闭连接，就像写端关闭的管道一样，服务器的 read()返回 0，这样服务器就知道客户端关闭了连接，也调用 close()关闭连接。注意，任何一方调用 close()后，连接的两个传输方向都关闭，不能再发送数据了。如果一方调用 shutdown()则连接处于半关闭状态，仍可接收对方发来的数据。

在学习 socket API 时要注意应用程序和 TCP 协议层是如何交互的： 应用程序调用某个 socket 函数时 TCP 协议层完成什么动作，比如调用 connect()会发出 SYN 段 应用程序如何知道 TCP 协议层的状态变化，比如从某个阻塞的 socket 函数返回就表明 TCP 协议收到了某些段，再比如 read()返回 0 就表明收到了 FIN 段

server

下面通过最简单的客户端/服务器程序的实例来学习 socket API。

server.c 的作用是从客户端读字符，然后将每个字符转换为大写并回送给客户端。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAXLINE 80
#define SERV_PORT 6666

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, n;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    listen(listenfd, 20);

    printf("Accepting connections ...\n");
    while (1) {
        cliaddr_len = sizeof(cliaddr);
        connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
        n = read(connfd, buf, MAXLINE);
        printf("received from %s at PORT %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
        ntohs(cliaddr.sin_port));
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
```

```
        write(connfd, buf, n);
        close(connfd);
    }
    return 0;
}
```

client

client.c 的作用是从命令行参数中获得一个字符串发给服务器，然后接收服务器返回的字符串并打印。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MAXLINE 80
#define SERV_PORT 6666

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;
    char *str;

    if (argc != 2) {
        fputs("usage: ./client message\n", stderr);
        exit(1);
    }
    str = argv[1];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    write(sockfd, str, strlen(str));

    n = read(sockfd, buf, MAXLINE);
```

```
    printf("Response from server:\n");
    write(STDOUT_FILENO, buf, n);
    close(sockfd);

    return 0;
}
```

由于客户端不需要固定的端口号，因此不必调用 bind()，客户端的端口号由内核自动分配。注意，客户端不是不允许调用 bind()，只是没有必要调用 bind()固定一个端口号，服务器也不是必须调用 bind()，但如果服务器不调用 bind()，内核会自动给服务器分配监听端口，每次启动服务器时端口号都不一样，客户端要连接服务器就会遇到麻烦。

客户端和服务器启动后可以使用 netstat 命令查看链接情况：

netstat -apn|grep 6666

出错处理封装函数

上面的例子不仅功能简单，而且简单到几乎没有什么错误处理，我们知道，系统调用不能保证每次都成功，必须进行出错处理，这样一方面可以保证程序逻辑正常，另一方面可以迅速得到故障信息。

为使错误处理的代码不影响主程序的可读性，我们把与 socket 相关的一些系统函数加上错误处理代码包装成新的函数，做成一个模块 wrap.c：

wrap.c

```
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
void perr_exit(const char *s)
{
    perror(s);
    exit(1);
}
int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)
{
    int n;
    again:
    if ( (n = accept(fd, sa, salenptr)) < 0) {
        if ((errno == ECONNABORTED) || (errno == EINTR))
            goto again;
        else
            perr_exit("accept error");
    }
    return n;
```

```
}
int Bind(int fd, const struct sockaddr *sa, socklen_t salen)
{
    int n;
    if ((n = bind(fd, sa, salen)) < 0)
        perr_exit("bind error");
    return n;
}
int Connect(int fd, const struct sockaddr *sa, socklen_t salen)
{
    int n;
    if ((n = connect(fd, sa, salen)) < 0)
        perr_exit("connect error");
    return n;
}
int Listen(int fd, int backlog)
{
    int n;
    if ((n = listen(fd, backlog)) < 0)
        perr_exit("listen error");
    return n;
}
int Socket(int family, int type, int protocol)
{
    int n;
    if ( (n = socket(family, type, protocol)) < 0)
        perr_exit("socket error");
    return n;
}
ssize_t Read(int fd, void *ptr, size_t nbytes)
{
    ssize_t n;
again:
    if ( (n = read(fd, ptr, nbytes)) == -1) {
        if (errno == EINTR)
            goto again;
        else
            return -1;
    }
    return n;
}
ssize_t Write(int fd, const void *ptr, size_t nbytes)
{
    ssize_t n;
again:
    if ( (n = write(fd, ptr, nbytes)) == -1) {
        if (errno == EINTR)
```

```
            goto again;
        else
            return -1;
    }
    return n;
}
int Close(int fd)
{
    int n;
    if ((n = close(fd)) == -1)
        perr_exit("close error");
    return n;
}
ssize_t Readn(int fd, void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nread;
    char *ptr;

    ptr = vptr;
    nleft = n;

    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;
            else
                return -1;
        } else if (nread == 0)
            break;
        nleft -= nread;
        ptr += nread;
    }
    return n - nleft;
}

ssize_t Writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr;
    nleft = n;

    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
```

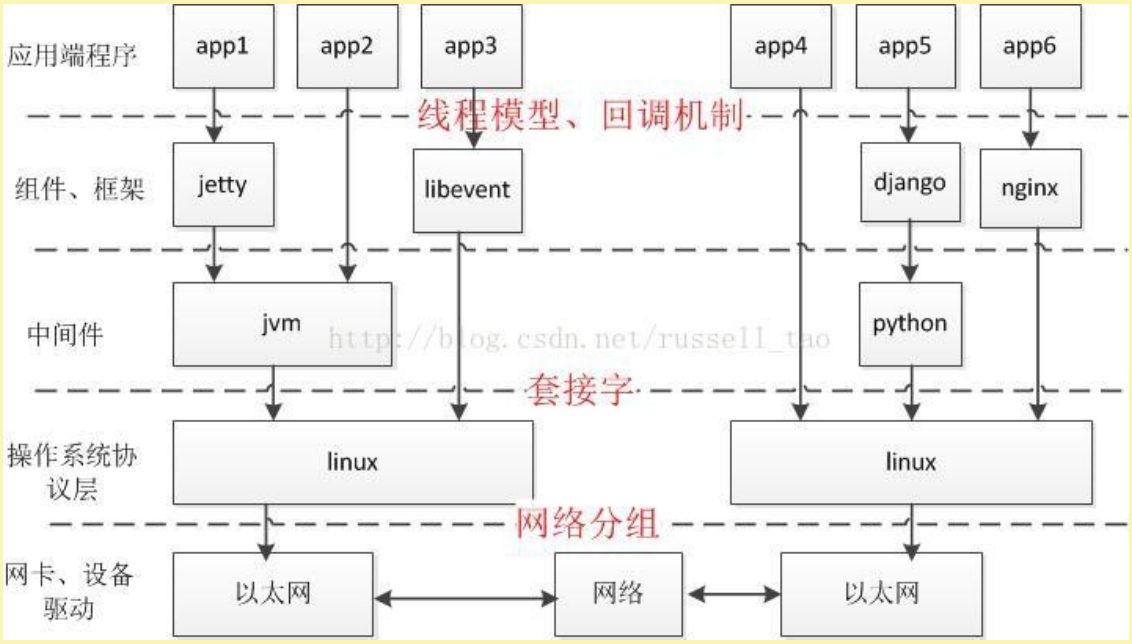
```
            if (nwritten < 0 && errno == EINTR)
                nwritten = 0;
            else
                return -1;
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return n;
}

static ssize_t my_read(int fd, char *ptr)
{
    static int read_cnt;
    static char *read_ptr;
    static char read_buf[100];

    if (read_cnt <= 0) {
again:
        if ((read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            if (errno == EINTR)
                goto again;
            return -1;
        } else if (read_cnt == 0)
            return 0;
        read_ptr = read_buf;
    }
    read_cnt--;
    *ptr = *read_ptr++;
    return 1;
}

ssize_t Readline(int fd, void *vptr, size_t maxlen)
{
    ssize_t n, rc;
    char c, *ptr;
    ptr = vptr;

    for (n = 1; n < maxlen; n++) {
        if ( (rc = my_read(fd, &c)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            *ptr = 0;
            return n - 1;
        } else
```

```
            return -1;
    }
    *ptr = 0;
    return n;
}
```

wrap.h

```
#ifndef __WRAP_H_
#define __WRAP_H_
void perr_exit(const char *s);
int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr);
int Bind(int fd, const struct sockaddr *sa, socklen_t salen);
int Connect(int fd, const struct sockaddr *sa, socklen_t salen);
int Listen(int fd, int backlog);
int Socket(int family, int type, int protocol);
ssize_t Read(int fd, void *ptr, size_t nbytes);
ssize_t Write(int fd, const void *ptr, size_t nbytes);
int Close(int fd);
ssize_t Readn(int fd, void *vptr, size_t n);
ssize_t Writen(int fd, const void *vptr, size_t n);
ssize_t my_read(int fd, char *ptr);
ssize_t Readline(int fd, void *vptr, size_t maxlen);
#endif
```

高并发服务器



多进程并发服务器

使用多进程并发服务器时要考虑以下几点：

1. 父进程最大文件描述个数(父进程中需要 close 关闭 accept 返回的新文件描述符)
2. 系统内创建进程个数(与内存大小相关)
3. 进程创建过多是否降低整体服务性能(进程调度)

server

```
/* server.c */
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include "wrap.h"
```

```
#define MAXLINE 80
#define SERV_PORT 800

void do_sigchild(int num)
{
    while (waitpid(0, NULL, WNOHANG) > 0)
        ;
}
int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, n;
    pid_t pid;

    struct sigaction newact;
    newact.sa_handler = do_sigchild;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGCHLD, &newact, NULL);

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    Listen(listenfd, 20);

    printf("Accepting connections ...\n");
    while (1) {
        cliaddr_len = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);

        pid = fork();
        if (pid == 0) {
            Close(listenfd);
            while (1) {
                n = Read(connfd, buf, MAXLINE);
                if (n == 0) {
```

```
                printf("the other side has been closed.\n");
                break;
            }
            printf("received from %s at PORT %d\n",
                    inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                    ntohs(cliaddr.sin_port));
            for (i = 0; i < n; i++)
                buf[i] = toupper(buf[i]);
            Write(connfd, buf, n);
        }
        Close(connfd);
        return 0;
    } else if (pid > 0) {
        Close(connfd);
    } else
        perr_exit("fork");
  }
  Close(listenfd);
  return 0;
}
```

client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 6666

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);
```

```
    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0) {
            printf("the other side has been closed.\n");
            break;
        } else
            Write(STDOUT_FILENO, buf, n);
    }
    Close(sockfd);
    return 0;
}
```

多线程并发服务器

在使用线程模型开发服务器时需考虑以下问题：

1. 调整进程内最大文件描述符上限
2. 线程如有共享数据，考虑线程同步
3. 服务于客户端线程退出时，退出处理。（退出值，分离态）
4. 系统负载，随着链接客户端增加，导致其它线程不能及时得到 CPU

server

```
/* server.c */
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>

#include "wrap.h"
#define MAXLINE 80
#define SERV_PORT 6666

struct s_info {
    struct sockaddr_in cliaddr;
    int connfd;
```

```
};
void *do_work(void *arg)
{
    int n,i;
    struct s_info *ts = (struct s_info*)arg;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    /* 可以在创建线程前设置线程创建属性,设为分离态,哪种效率高呢?  */
    pthread_detach(pthread_self());
    while (1) {
        n = Read(ts->connfd, buf, MAXLINE);
        if (n == 0) {
            printf("the other side has been closed.\n");
            break;
        }
        printf("received from %s at PORT %d\n",
                inet_ntop(AF_INET, &(*ts).cliaddr.sin_addr, str, sizeof(str)),
                ntohs((*ts).cliaddr.sin_port));
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        Write(ts->connfd, buf, n);
    }
    Close(ts->connfd);
}

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    int i = 0;
    pthread_t tid;
    struct s_info ts[256];

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    Listen(listenfd, 20);

    printf("Accepting connections ...\n");
    while (1) {
        cliaddr_len = sizeof(cliaddr);
```

```
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
        ts[i].cliaddr = cliaddr;
        ts[i].connfd = connfd;
        /* 达到线程最大数时，pthread_create出错处理，增加服务器稳定性 */
        pthread_create(&tid, NULL, do_work, (void*)&ts[i]);
        i++;
    }
    return 0;
}
```

client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"
#define MAXLINE 80
#define SERV_PORT 6666
int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }
    Close(sockfd);
    return 0;
```

```
}
```

多路 I/O 转接服务器

多路 IO 转接服务器也叫做多任务 IO 服务器。该类服务器实现的主旨思想是，不再由应用程序自己监视客户端连接，取而代之由内核替应用程序监视文件。

主要使用的方法有三种

select

1. select 能监听的文件描述符个数受限于 FD_SETSIZE，一般为 1024，单纯改变进程打开的文件描述符个数并不能改变 select 监听文件个数
2. 解决 1024 以下客户端时使用 select 是很合适的，但如果链接客户端过多，select 采用的是轮询模型，会大大降低服务器响应效率，不应在 select 上投入更多精力。

```
#include <sys/select.h>
/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

    nfds:       监控的文件描述符集里最大文件描述符加 1，因为此参数会告诉内核检测前多少个文件描述符的状态
    readfds:    监控有读数据到达文件描述符集合，传入传出参数
    writefds:   监控写数据到达文件描述符集合，传入传出参数
    exceptfds:  监控异常发生达文件描述符集合,如带外数据到达异常，传入传出参数
    timeout:    定时阻塞监控时间，3 种情况
                1.NULL，永远等下去
                2.设置 timeval，等待固定时间
                3.设置 timeval 里时间均为 0，检查描述字后立即返回，轮询
    struct timeval {
        long tv_sec; /* seconds */
        long tv_usec; /* microseconds */
    };
    void FD_CLR(int fd, fd_set *set);  //把文件描述符集合里 fd 位清 0
    int FD_ISSET(int fd, fd_set *set);     //测试文件描述符集合里 fd 是否置 1
    void FD_SET(int fd, fd_set *set);  //把文件描述符集合里 fd 位置 1
    void FD_ZERO(fd_set *set);         //把文件描述符集合里所有位清 0
```

server

```
/* server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 6666

int main(int argc, char *argv[])
{
    int i, maxi, maxfd, listenfd, connfd, sockfd;
    int nready, client[FD_SETSIZE];    /* FD_SETSIZE 默认为 1024 */
    ssize_t n;
    fd_set rset, allset;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];         /* #define INET_ADDRSTRLEN 16 */
    socklen_t cliaddr_len;
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    Listen(listenfd, 20);       /* 默认最大128 */

    maxfd = listenfd;           /* 初始化 */
    maxi = -1;                  /* client[]的下标 */

    for (i = 0; i < FD_SETSIZE; i++)
        client[i] = -1;         /* 用-1初始化client[] */

    FD_ZERO(&allset);
    FD_SET(listenfd, &allset); /* 构造select监控文件描述符集 */

    for ( ; ; ) {
```

```
    rset = allset;          /* 每次循环时都从新设置select监控信号集 */
    nready = select(maxfd+1, &rset, NULL, NULL, NULL);

    if (nready < 0)
        perr_exit("select error");
    if (FD_ISSET(listenfd, &rset)) { /* new client connection */
        cliaddr_len = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
        printf("received from %s at PORT %d\n",
                inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                ntohs(cliaddr.sin_port));
        for (i = 0; i < FD_SETSIZE; i++) {
            if (client[i] < 0) {
                client[i] = connfd; /* 保存accept返回的文件描述符到client[]里 */
                break;
            }
        }
        /* 达到select能监控的文件个数上限 1024 */
        if (i == FD_SETSIZE) {
            fputs("too many clients\n", stderr);
            exit(1);
        }

        FD_SET(connfd, &allset);    /* 添加一个新的文件描述符到监控信号集里 */
        if (connfd > maxfd)
            maxfd = connfd;         /* select第一个参数需要 */
        if (i > maxi)
            maxi = i;               /* 更新client[]最大下标值 */

        if (--nready == 0)
            continue;               /* 如果没有更多的就绪文件描述符继续回到上面select阻塞监听,
                                       负责处理未处理完的就绪文件描述符 */
    }
    for (i = 0; i <= maxi; i++) {  /* 检测哪个clients 有数据就绪 */
        if ( (sockfd = client[i]) < 0)
            continue;
        if (FD_ISSET(sockfd, &rset)) {
            if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
                Close(sockfd);      /* 当client关闭链接时，服务器端也关闭对应链接 */
                FD_CLR(sockfd, &allset); /* 解除select监控此文件描述符 */
                client[i] = -1;
            } else {
                int j;
                for (j = 0; j < n; j++)
                    buf[j] = toupper(buf[j]);
                Write(sockfd, buf, n);
            }
```

```
                if (--nready == 0)
                    break;
            }
        }
    }
    close(listenfd);
    return 0;
}
```

client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 6666

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }
    Close(sockfd);
```

```
    return 0;
}
```

pselect

pselect 原型如下。此模型应用较少，有需要的同学可参考 select 模型自行编写 C/S

```
#include <sys/select.h>
int pselect(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, const struct timespec *timeout,
            const sigset_t *sigmask);
    struct timespec {
        long tv_sec; /* seconds */
        long tv_nsec; /* nanoseconds */
    };
    用 sigmask 替代当前进程的阻塞信号集，调用返回后还原原有阻塞信号集
```

poll

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
    struct pollfd {
        int fd; /* 文件描述符 */
        short events; /* 监控的事件 */
        short revents; /* 监控事件中满足条件返回的事件 */
    };
    POLLIN          普通或带外优先数据可读,即 POLLRDNORM | POLLRDBAND
    POLLRDNORM      数据可读
    POLLRDBAND      优先级带数据可读
    POLLPRI         高优先级可读数据
    POLLOUT         普通或带外数据可写
    POLLWRNORM      数据可写
    POLLWRBAND      优先级带数据可写
    POLLERR         发生错误
    POLLHUP         发生挂起
    POLLNVAL        描述字不是一个打开的文件

    nfds            监控数组中有多少文件描述符需要被监控

    timeout         毫秒级等待
        -1: 阻塞等，#define INFTIM -1               Linux 中没有定义此宏
        0: 立即返回，不阻塞进程
        >0: 等待指定毫秒数，如当前系统时间精度不够毫秒，向上取值
```

如果不再监控某个文件描述符时，可以把 pollfd 中，fd 设置为-1，poll 不再监控此 pollfd，下次返回时，把 revents 设置为 0。

相较于 select 而言，poll 的优势：

1. 传入、传出事件分离。无需每次调用时，重新设定监听事件。

2. 文件描述符上限，可突破 1024 限制。能监控的最大上限数可使用配置文件调整。

server

```
/* server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <poll.h>
#include <errno.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 6666
#define OPEN_MAX 1024

int main(int argc, char *argv[])
{
    int i, j, maxi, listenfd, connfd, sockfd;
    int nready;
    ssize_t n;
    char buf[MAXLINE], str[INET_ADDRSTRLEN];
    socklen_t clilen;
    struct pollfd client[OPEN_MAX];
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    Listen(listenfd, 20);
```

```
client[0].fd = listenfd;
client[0].events = POLLIN;                  /* listenfd监听普通读事件 */

for (i = 1; i < OPEN_MAX; i++)
    client[i].fd = -1;                          /* 用-1初始化client[]里剩下元素 */
maxi = 0;                                       /* client[]数组有效元素中最大元素下标 */

for ( ; ; ) {
    nready = poll(client, maxi+1, -1);              /* 阻塞 */
    if (client[0].revents & POLLIN) {       /* 有客户端链接请求 */
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
        printf("received from %s at PORT %d\n",
                inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                ntohs(cliaddr.sin_port));
        for (i = 1; i < OPEN_MAX; i++) {
            if (client[i].fd < 0) {
                client[i].fd = connfd; /* 找到client[]中空闲的位置，存放accept返回的connfd */
                break;
            }
        }

        if (i == OPEN_MAX)
            perr_exit("too many clients");

        client[i].events = POLLIN;      /* 设置刚刚返回的connfd，监控读事件 */
        if (i > maxi)
            maxi = i;                       /* 更新client[]中最大元素下标 */
        if (--nready <= 0)
            continue;                       /* 没有更多就绪事件时,继续回到poll阻塞 */
    }
    for (i = 1; i <= maxi; i++) {           /* 检测client[] */
        if ((sockfd = client[i].fd) < 0)
            continue;
        if (client[i].revents & POLLIN) {
            if ((n = Read(sockfd, buf, MAXLINE)) < 0) {
                if (errno == ECONNRESET) { /* 当收到 RST标志时 */
                    /* connection reset by client */
                    printf("client[%d] aborted connection\n", i);
                    Close(sockfd);
                    client[i].fd = -1;
                } else {
                    perr_exit("read error");
                }
            } else if (n == 0) {
                /* connection closed by client */
                printf("client[%d] closed connection\n", i);
```

```
                    Close(sockfd);
                    client[i].fd = -1;
                } else {
                    for (j = 0; j < n; j++)
                        buf[j] = toupper(buf[j]);
                        Writen(sockfd, buf, n);
                }
                if (--nready <= 0)
                    break;              /* no more readable descriptors */
            }
        }
    }
    return 0;
}
```

client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 6666

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
```

```
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }
    Close(sockfd);
    return 0;
}
```

ppoll

GNU 定义了 ppoll（非 POSIX 标准），可以支持设置信号屏蔽字，大家可参考 poll 模型自行实现 C/S。

```
#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <poll.h>
int ppoll(struct pollfd *fds, nfds_t nfds,
          const struct timespec *timeout_ts, const sigset_t *sigmask);
```

epoll

epoll 是 Linux 下多路复用 IO 接口 select/poll 的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率，因为它会复用文件描述符集合来传递结果而不用迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核 IO 事件异步唤醒而加入 Ready 队列的描述符集合就行了。

目前 epoll 是 linux 大规模并发网络程序中的热门首选模型。

epoll 除了提供 select/poll 那种 IO 事件的水平触发（Level Triggered）外，还提供了边沿触发（Edge Triggered），这就使得用户空间程序有可能缓存 IO 状态，减少 epoll_wait/epoll_pwait 的调用，提高应用程序效率。

可以使用 cat 命令查看一个进程可以打开的 socket 描述符上限。

```
cat /proc/sys/fs/file-max
```

如有需要，可以通过修改配置文件的方式修改该上限值。

```
sudo vi /etc/security/limits.conf
在文件尾部写入以下配置,soft 软限制，hard 硬限制。如下图所示。
* soft nofile 65536
* hard nofile 100000
```

```
39 #        - nice - max nice priority allowed to raise to values: [-20, 19]
40 #        - rtprio - max realtime priority
41 #        - chroot - change root to directory (Debian-specific)
42 #
43 #<domain>      <type>  <item>         <value>
44 #
45
46 #*               soft    core            0
47 #root            hard    core            100000
48 #*               hard    rss             10000
49 #@student        hard    nproc           20
50 #@faculty        soft    nproc           20
51 #@faculty        hard    nproc           50
52 #ftp            hard    nproc           0
53 #ftp            -       chroot          /ftp
54 #@student        -       maxlogins       4
55 *               soft    nofile          65536
56 *               hard    nofile          100000
57
58 # End of file
/etc/security/limits.conf
"/etc/security/limits.conf" 58L, 2243C 已写入
```

基础 API

1. 创建一个 epoll 句柄，参数 size 用来告诉内核监听的文件描述符的个数，跟内存大小有关。

```
#include <sys/epoll.h>
int epoll_create(int size)
    size：监听数目（内核参考值）
    返回值：成功：非负文件描述符；失败：-1，设置相应的 errno
```

2. 控制某个 epoll 监控的文件描述符上的事件：注册、修改、删除。

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
    epfd:   为 epoll_creat 的句柄
    op:     表示动作，用 3 个宏来表示：
        EPOLL_CTL_ADD (注册新的 fd 到 epfd),
        EPOLL_CTL_MOD (修改已经注册的 fd 的监听事件),
        EPOLL_CTL_DEL (从 epfd 删除一个 fd);
    event:  告诉内核需要监听的事件

    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;

        EPOLLIN ：   表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）
        EPOLLOUT:    表示对应的文件描述符可以写
        EPOLLPRI:    表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）
        EPOLLERR:    表示对应的文件描述符发生错误
        EPOLLHUP:    表示对应的文件描述符被挂断;
        EPOLLET:     将 EPOLL 设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)而言的
        EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个 socket 的话，需要再次把这个
socket 加入到 EPOLL 队列里
        返回值：成功：0；失败：-1，设置相应的 errno
```

3. 等待所监控文件描述符上有事件的产生，类似于 select()调用。

```
    #include <sys/epoll.h>
    int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)
        events:     用来存内核得到事件的集合，可简单看作数组。
        maxevents:  告之内核这个 events 有多大，这个 maxevents 的值不能大于创建 epoll_create()时的 size，
        timeout:    是超时时间
            -1: 阻塞
            0:  立即返回，非阻塞
            >0: 指定毫秒
        返回值:  成功返回有多少文件描述符就绪，时间到时返回 0，出错返回-1
```

server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <errno.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 6666
#define OPEN_MAX 1024

int main(int argc, char *argv[])
{
    int i, j, maxi, listenfd, connfd, sockfd;
    int nready, efd, res;
    ssize_t n;
    char buf[MAXLINE], str[INET_ADDRSTRLEN];
```

```
socklen_t clilen;
int client[OPEN_MAX];
struct sockaddr_in cliaddr, servaddr;
struct epoll_event tep, ep[OPEN_MAX];

listenfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

Bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

Listen(listenfd, 20);

for (i = 0; i < OPEN_MAX; i++)
    client[i] = -1;
maxi = -1;

efd = epoll_create(OPEN_MAX);
if (efd == -1)
    perr_exit("epoll_create");

tep.events = EPOLLIN; tep.data.fd = listenfd;

res = epoll_ctl(efd, EPOLL_CTL_ADD, listenfd, &tep);
if (res == -1)
    perr_exit("epoll_ctl");

while (1) {
    nready = epoll_wait(efd, ep, OPEN_MAX, -1); /* 阻塞监听 */
    if (nready == -1)
        perr_exit("epoll_wait");

    for (i = 0; i < nready; i++) {
        if (!(ep[i].events & EPOLLIN))
            continue;
        if (ep[i].data.fd == listenfd) {
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
            printf("received from %s at PORT %d\n",
                   inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                   ntohs(cliaddr.sin_port));
            for (j = 0; j < OPEN_MAX; j++) {
                if (client[j] < 0) {
                    client[j] = connfd; /* save descriptor */
```

```
                    break;
                }
            }

            if (j == OPEN_MAX)
                perr_exit("too many clients");
            if (j > maxi)
                maxi = j;       /* max index in client[] array */

            tep.events = EPOLLIN;
            tep.data.fd = connfd;
            res = epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &tep);
            if (res == -1)
                perr_exit("epoll_ctl");
        } else {
            sockfd = ep[i].data.fd;
            n = Read(sockfd, buf, MAXLINE);
            if (n == 0) {
                for (j = 0; j <= maxi; j++) {
                    if (client[j] == sockfd) {
                        client[j] = -1;
                        break;
                    }
                }
                res = epoll_ctl(efd, EPOLL_CTL_DEL, sockfd, NULL);
                if (res == -1)
                    perr_exit("epoll_ctl");

                Close(sockfd);
                printf("client[%d] closed connection\n", j);
            } else {
                for (j = 0; j < n; j++)
                    buf[j] = toupper(buf[j]);
                Writen(sockfd, buf, n);
            }
        }
    }
    close(listenfd);
    close(efd);
    return 0;
}
```

client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 6666

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }

    Close(sockfd);
    return 0;
}
```

epoll 进阶

事件模型

EPOLL 事件有两种模型：

Edge Triggered (ET) 边缘触发只有数据到来才触发，不管缓存区中是否还有数据。

Level Triggered (LT) 水平触发只要有数据都会触发。

思考如下步骤：

1. 假定我们已经把一个用来从管道中读取数据的文件描述符(rfd)添加到 epoll 描述符。
2. 管道的另一端写入了 2KB 的数据
3. 调用 epoll_wait，并且它会返回 rfd，说明它已经准备好读取操作
4. 读取 1KB 的数据
5. 调用 epoll_wait……

在这个过程中，有两种工作模式：

ET 模式

ET 模式即 Edge Triggered 工作模式。

如果我们在第 1 步将 rfd 添加到 epoll 描述符的时候使用了 EPOLLET 标志，那么在第 5 步调用 epoll_wait 之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候 ET 工作模式才会汇报事件。因此在第 5 步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。epoll 工作在 ET 模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。最好以下面的方式调用 ET 模式的 epoll 接口，在后面会介绍避免可能的缺陷。

1) 基于非阻塞文件句柄
2) 只有当 read 或者 write 返回 EAGAIN(非阻塞读，暂时无数据)时才需要挂起、等待。但这并不是说每次 read 时都需要循环读，直到读到产生一个 EAGAIN 才认为此次事件处理完成，当 read 返回的读到的数据长度小于请求的数据长度时，就可以确定此时缓冲中已没有数据了，也就可以认为此事读事件已处理完成。

LT 模式

LT 模式即 Level Triggered 工作模式。

与 ET 模式不同的是，以 LT 方式调用 epoll 接口的时候，它就相当于一个速度比较快的 poll，无论后面的数据

是否被使用。

比较

LT(level triggered)：LT 是**缺省**的工作方式，并且同时支持 block 和 no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。**传统的 select/poll 都是这种模型的代表。**

ET(edge-triggered)：ET 是高速工作方式，只支持 no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过 epoll 告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知。请注意，如果一直不对这个 fd 作 IO 操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once).

实例一：

基于管道 epoll ET 触发模式

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <errno.h>
#include <unistd.h>

#define MAXLINE 10

int main(int argc, char *argv[])
{
    int efd, i;
    int pfd[2];
    pid_t pid;
    char buf[MAXLINE], ch = 'a';

    pipe(pfd);
    pid = fork();
    if (pid == 0) {
        close(pfd[0]);
        while (1) {
            for (i = 0; i < MAXLINE/2; i++)
                buf[i] = ch;
            buf[i-1] = '\n';
            ch++;

            for (; i < MAXLINE; i++)
                buf[i] = ch;
```

```
            buf[i-1] = '\n';
            ch++;

            write(pfd[1], buf, sizeof(buf));
            sleep(2);
        }
        close(pfd[1]);
    } else if (pid > 0) {
        struct epoll_event event;
        struct epoll_event resevent[10];
        int res, len;
        close(pfd[1]);

        efd = epoll_create(10);
        /* event.events = EPOLLIN; */
        event.events = EPOLLIN | EPOLLET;       /* ET 边沿触发 ，默认是水平触发 */
        event.data.fd = pfd[0];
        epoll_ctl(efd, EPOLL_CTL_ADD, pfd[0], &event);

        while (1) {
            res = epoll_wait(efd, resevent, 10, -1);
            printf("res %d\n", res);
            if (resevent[0].data.fd == pfd[0]) {
                len = read(pfd[0], buf, MAXLINE/2);
                write(STDOUT_FILENO, buf, len);
            }
        }
        close(pfd[0]);
        close(efd);
    } else {
        perror("fork");
        exit(-1);
    }
    return 0;
}
```

实例二：

基于网络 C/S 模型的 epoll ET 触发模式

server

```
/* server.c */
```

```
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/epoll.h>
#include <unistd.h>

#define MAXLINE 10
#define SERV_PORT 8080

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, efd;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    listen(listenfd, 20);

    struct epoll_event event;
    struct epoll_event resevent[10];
    int res, len;
    efd = epoll_create(10);
    event.events = EPOLLIN | EPOLLET;      /* ET 边沿触发 ，默认是水平触发 */

    printf("Accepting connections ...\n");
    cliaddr_len = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
    printf("received from %s at PORT %d\n",
           inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
           ntohs(cliaddr.sin_port));

    event.data.fd = connfd;
```

```
    epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &event);

    while (1) {
        res = epoll_wait(efd, resevent, 10, -1);
        printf("res %d\n", res);
        if (resevent[0].data.fd == connfd) {
            len = read(connfd, buf, MAXLINE/2);
            write(STDOUT_FILENO, buf, len);
        }
    }
    return 0;
}
```

client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>

#define MAXLINE 10
#define SERV_PORT 8080

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, i;
    char ch = 'a';

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (1) {
        for (i = 0; i < MAXLINE/2; i++)
            buf[i] = ch;
        buf[i-1] = '\n';
        ch++;
```

```
        for (; i < MAXLINE; i++)
            buf[i] = ch;
        buf[i-1] = '\n';
        ch++;

        write(sockfd, buf, sizeof(buf));
        sleep(10);
    }
    Close(sockfd);
    return 0;
}
```

实例三：

基于网络 C/S 非阻塞模型的 epoll ET 触发模式

server

```
/* server.c */
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/epoll.h>
#include <unistd.h>
#include <fcntl.h>

#define MAXLINE 10
#define SERV_PORT 8080

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, efd, flag;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    listen(listenfd, 20);

    struct epoll_event event;
    struct epoll_event resevent[10];
    int res, len;
    efd = epoll_create(10);
    /* event.events = EPOLLIN; */
    event.events = EPOLLIN | EPOLLET;      /* ET 边沿触发 ，默认是水平触发 */

    printf("Accepting connections ...\n");
    cliaddr_len = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
    printf("received from %s at PORT %d\n",
            inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
            ntohs(cliaddr.sin_port));

    flag = fcntl(connfd, F_GETFL);
    flag |= O_NONBLOCK;
    fcntl(connfd, F_SETFL, flag);
    event.data.fd = connfd;
    epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &event);

    while (1) {
        printf("epoll_wait begin\n");
        res = epoll_wait(efd, resevent, 10, -1);
        printf("epoll_wait end res %d\n", res);

        if (resevent[0].data.fd == connfd) {
            while ((len = read(connfd, buf, MAXLINE/2)) > 0)
                write(STDOUT_FILENO, buf, len);
        }
    }
    return 0;
}
```

client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>

#define MAXLINE 10
#define SERV_PORT 8080

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, i;
    char ch = 'a';

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (1) {
        for (i = 0; i < MAXLINE/2; i++)
            buf[i] = ch;
        buf[i-1] = '\n';
        ch++;

        for (; i < MAXLINE; i++)
            buf[i] = ch;
        buf[i-1] = '\n';
        ch++;

        write(sockfd, buf, sizeof(buf));
        sleep(10);
    }
    Close(sockfd);
    return 0;
}
```

epoll 反应堆思想

epoll 还有一种更高级的使用方法，那就是借鉴封装的思想，简单的说就是当某个事情发生了，自动的去处理这个事情。这样的思想对我们的编码来说就是设置回调，将文件描述符，对应的事件，和事件产生时的处理函数封装到一起，这样当某个文件描述符的事件发生了，回调函数会自动被触发，这就是所谓的反应堆思想。

从我们之前对 epoll 的使用上如何去支持反应堆呢？需要重新再认识一下 struct epoll_event 中的 epoll_data_t 结构体:

```
typedef union epoll_data {
    void      *ptr;
    int       fd;
    uint32_t    u32;
    uint64_t    u64;
 } epoll_data_t;
```

我们之前使用的是共用体上的 fd 域,如果是要实现封装思想,光有 fd 是不够的,所以转换思路,看第一个域 ptr,是一个泛型指针,指针可以指向一块内存区域,这块区域可以代表一个结构体,既然是结构体,那么我们就可以自定义了,将我们非常需要的文件描述符,事件类型,回调函数都封装在结构体上(我们在信号一章就见识过 struct sigaction 上有掩码和回调函数等信息),这样当我们要监控的文件描述符对应的事件发生之后,我们去调用回调函数就可以了，这样就可以将文件描述符对应事件的处理代码梳理的非常清晰。

线程池并发服务器

首先什么是线程池?

线程池是一个抽象概念，可以简单的认为若干线程在一起运行，线程不退出，等待有任务处理。

为什么要有线程池?

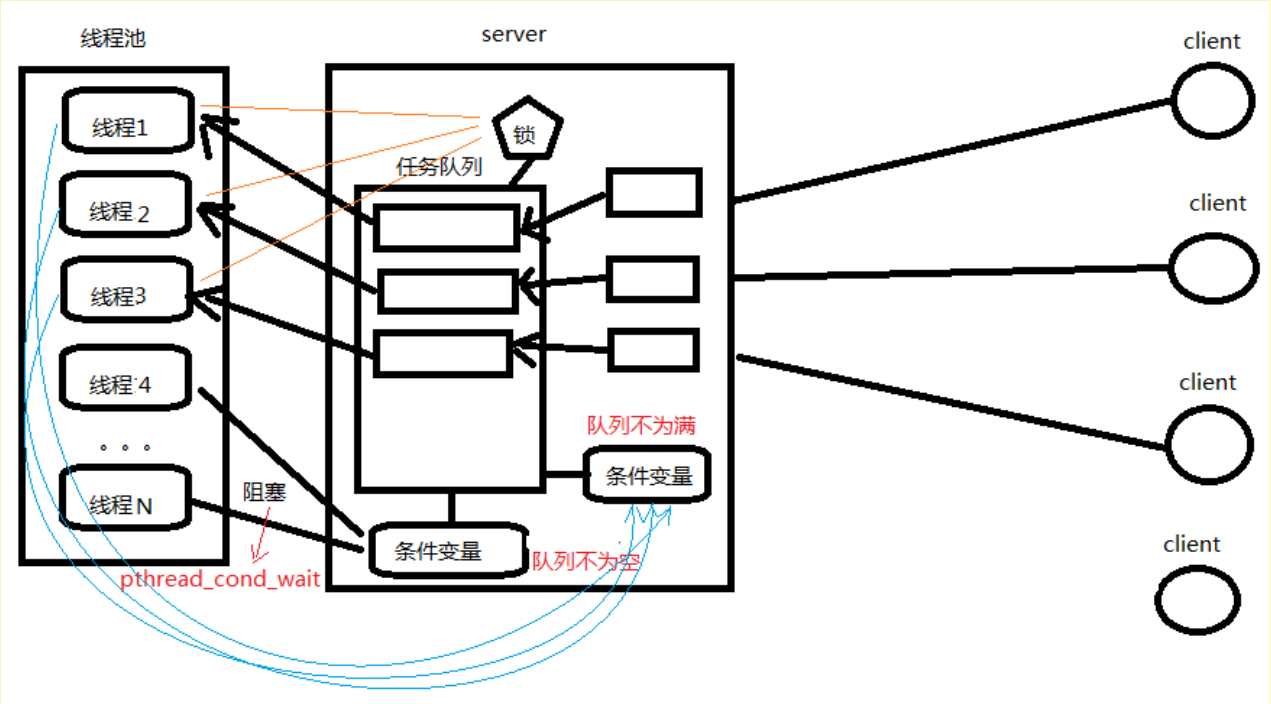
1. 以网络编程服务器端为例,作为服务器端支持高并发,可以有多个客户端连接,发出请求,对于多个请求我们每次都去建立线程,这样线程会创建很多,而且线程执行完销毁也会有很大的系统开销,使用上效率很低。
2. 之前在线程篇章中，我们也知道创建线程并非多多益善，所以我们的思路是提前创建好若干个线程，不退出，等待任务的产生，去接收任务处理后等待下一个任务。

线程池如何实现？需要思考 2 个问题?

1. 假设线程池创建了，线程们如何去协调接收任务并且处理?
2. 线程池上的线程如何能够执行不同的请求任务?

上述问题 1 就很像我们之前学过的生产者和消费者模型，客户端对应生产者，服务器端这边的线程池对应消费者，需要借助互斥锁和条件变量来搞定。

问题 2 解决思路就是利用回调机制，我们同样可以借助结构体的方式，对任务进行封装，比如任务的数据和任务处理回调都封装在结构体上，这样线程池的工作线程拿到任务的同时，也知道该如何执行了。



UDP 服务器

传输层主要应用的协议模型有两种，一种是 TCP 协议，另外一种则是 UDP 协议。TCP 协议在网络通信中占主导地位，绝大多数的网络通信借助 TCP 协议完成数据传输。但 UDP 也是网络通信中不可或缺的重要通信手段。

相较于 TCP 而言，UDP 通信的形式更像是发短信。不需要在数据传输之前建立、维护连接。只专心获取数据就好。省去了三次握手的过程，通信速度可以大大提高，但与之伴随的通信的稳定性和正确率便得不到保证。因此，我们称 UDP 为“无连接的不可靠报文传递”。

那么与我们熟知的 TCP 相比，UDP 有哪些优点和不足呢？由于无需创建连接，所以 UDP 开销较小，数据传输速度快，实时性较强。多用于对实时性要求较高的通信场合，如视频会议、电话会议等。但随之也伴随着数据传输不可靠，传输数据的正确率、传输顺序和流量都得不到控制和保证。所以，通常情况下，使用 UDP 协议进行数据传输，为保证数据的正确性，我们需要在应用层添加辅助校验协议来弥补 UDP 的不足，以达到数据可靠传输的目的。

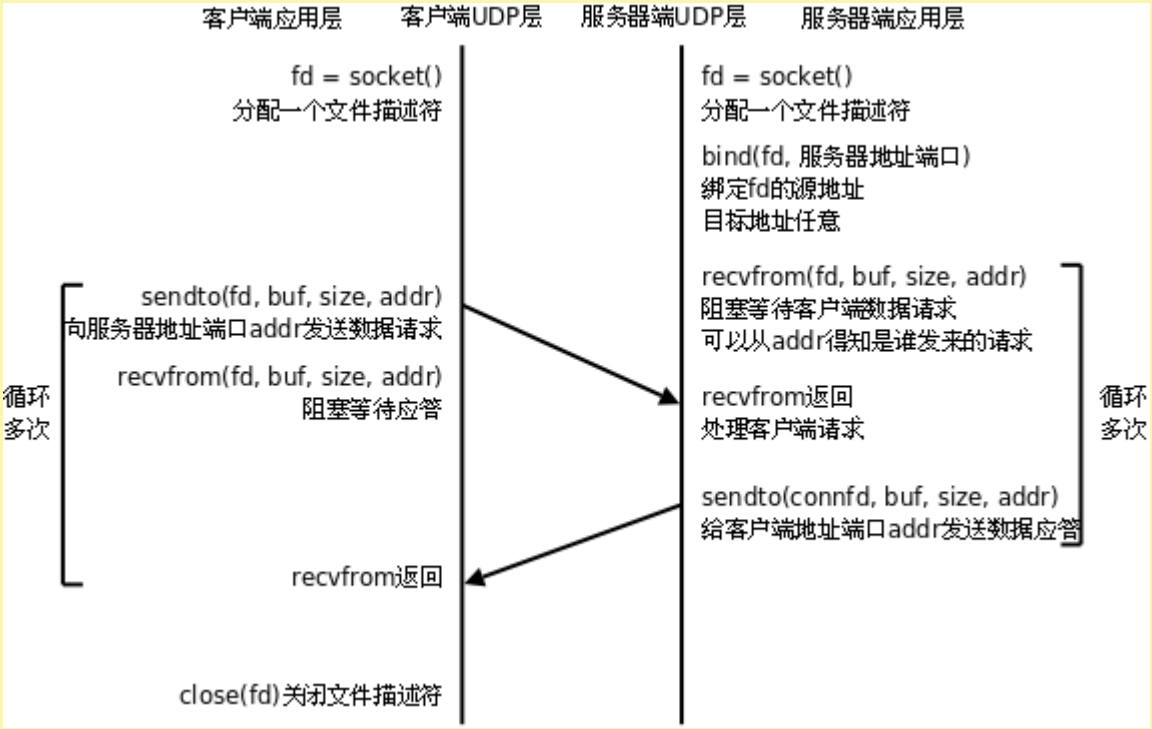
与 TCP 类似的，UDP 也有可能出现缓冲区被填满后，再接收数据时丢包的现象。由于它没有 TCP 滑动窗口的机制，通常采用如下两种方法解决：

1) 服务器应用层设计流量控制，控制发送数据速度。

2) 借助 setsockopt 函数改变接收缓冲区大小。如：

```
#include <sys/socket.h>
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
    int n = 220x1024
    setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));
```

C/S 模型-UDP



UDP 处理模型

由于 UDP 不需要维护连接，程序逻辑简单了很多，但是 UDP 协议是不可靠的，保证通讯可靠性的机制需要在应用层实现。

编译运行 server，在两个终端里各开一个 client 与 server 交互，看看 server 是否具有并发服务的能力。用 Ctrl+C 关闭 server，然后再运行 server，看此时 client 还能否和 server 联系上。和前面 TCP 程序的运行结果相比较，体会无连接的含义。

server

```
#include <string.h>
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>
#include <strings.h>
#include <arpa/inet.h>
#include <ctype.h>

#define MAXLINE 80
#define SERV_PORT 6666

int main(void)
{
```

```
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int sockfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, n;

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    printf("Accepting connections ...\n");

    while (1) {
        cliaddr_len = sizeof(cliaddr);
        n = recvfrom(sockfd, buf, MAXLINE,0, (struct sockaddr *)&cliaddr, &cliaddr_len);
        if (n == -1)
            perror("recvfrom error");
        printf("received from %s at PORT %d\n",
                inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                ntohs(cliaddr.sin_port));
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);

        n = sendto(sockfd, buf, n, 0, (struct sockaddr *)&cliaddr, sizeof(cliaddr));
        if (n == -1)
            perror("sendto error");
    }
    close(sockfd);
    return 0;
}
```

client

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <strings.h>
#include <ctype.h>
```

```
#define MAXLINE 80
#define SERV_PORT 6666

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    int sockfd, n;
    char buf[MAXLINE];

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        n = sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr, sizeof(servaddr));
        if (n == -1)
            perror("sendto error");
        n = recvfrom(sockfd, buf, MAXLINE, 0, NULL, 0);
        if (n == -1)
            perror("recvfrom error");
        write(STDOUT_FILENO, buf, n);
    }
    close(sockfd);
    return 0;
}
```

多播(组播)（了解）

组播组可以是永久的也可以是临时的。组播组地址中，有一部分由官方分配的，称为永久组播组。永久组播组保持不变的是它的 ip 地址，组中的成员构成可以发生变化。永久组播组中成员的数量都可以是任意的，甚至可以为零。那些没有保留下来供永久组播组使用的 ip 组播地址，可以被临时组播组利用。

```
224.0.0.0～224.0.0.255      为预留的组播地址（永久组地址），地址 224.0.0.0 保留不做分配，其它地址供路由协议使用；
224.0.1.0～224.0.1.255      是公用组播地址，可以用于 Internet；欲使用需申请。
224.0.2.0～238.255.255.255  为用户可用的组播地址（临时组地址），全网范围内有效；
239.0.0.0～239.255.255.255  为本地管理组播地址，仅在特定的本地范围内有效。
```

可使用 ip ad 命令查看网卡编号，如：

```
itcast$ ip ad
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
```

```
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN group default
qlen 1000
    link/ether 00:0c:29:0a:c4:f4 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::20c:29ff:fe0a:c4f4/64 scope link
       valid_lft forever preferred_lft forever
```

if_nametoindex() 函数可以根据网卡名，获取网卡序号。

server

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <net/if.h>

#define SERVER_PORT 6666
#define CLIENT_PORT 9000
#define MAXLINE 1500
#define GROUP "239.0.0.2"

int main(void)
{
    int sockfd, i ;
    struct sockaddr_in serveraddr, clientaddr;
    char buf[MAXLINE] = "itcast\n";
    char ipstr[INET_ADDRSTRLEN]; /* 16 Bytes */
    socklen_t clientlen;
    ssize_t len;
    struct ip_mreqn group;

    /* 构造用于UDP通信的套接字 */
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET; /* IPv4 */
```

```
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY); /* 本地任意 IP INADDR_ANY = 0 */
    serveraddr.sin_port = htons(SERVER_PORT);

    bind(sockfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));

    /*设置组地址*/
    inet_pton(AF_INET, GROUP, &group.imr_multiaddr);
    /*本地任意 IP*/
    inet_pton(AF_INET, "0.0.0.0", &group.imr_address);
    /* eth0 --> 编号 命令: ip ad */
    group.imr_ifindex = if_nametoindex("eth0");
    setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_IF, &group, sizeof(group));

    /*构造 client 地址 IP+端口 */
    bzero(&clientaddr, sizeof(clientaddr));
    clientaddr.sin_family = AF_INET; /* IPv4 */
    inet_pton(AF_INET, GROUP, &clientaddr.sin_addr.s_addr);
    clientaddr.sin_port = htons(CLIENT_PORT);

    while (1) {
        //fgets(buf, sizeof(buf), stdin);
        sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&clientaddr, sizeof(clientaddr));
        sleep(1);
    }
    close(sockfd);
    return 0;
}
```

client

```
#include <netinet/in.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <net/if.h>

#define SERVER_PORT 6666
#define MAXLINE 4096
#define CLIENT_PORT 9000
```

```
#define GROUP "239.0.0.2"

int main(int argc, char *argv[])
{
    struct sockaddr_in serveraddr, localaddr;
    int confd;
    ssize_t len;
    char buf[MAXLINE];

    /* 定义组播结构体 */
    struct ip_mreqn group;
    confd = socket(AF_INET, SOCK_DGRAM, 0);

    //初始化本地端地址
    bzero(&localaddr, sizeof(localaddr));
    localaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "0.0.0.0" , &localaddr.sin_addr.s_addr);
    localaddr.sin_port = htons(CLIENT_PORT);

    bind(confd, (struct sockaddr *)&localaddr, sizeof(localaddr));

    /*设置组地址*/
    inet_pton(AF_INET, GROUP, &group.imr_multiaddr);
    /*本地任意IP*/
    inet_pton(AF_INET, "0.0.0.0", &group.imr_address);
    /* eth0 --> 编号 命令: ip ad */
    group.imr_ifindex = if_nametoindex("eth0");
    /*设置client 加入多播组 */
    setsockopt(confd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &group, sizeof(group));

    while (1) {
        len = recvfrom(confd, buf, sizeof(buf), 0, NULL, 0);
        write(STDOUT_FILENO, buf, len);
    }
    close(confd);
    return 0;
}
```

socket IPC（本地套接字 domain）

socket API 原本是为网络通讯设计的，但后来在 socket 的框架上发展出一种 IPC 机制，就是 UNIX Domain

Socket。虽然网络 socket 也可用于同一台主机的进程间通讯（通过 loopback 地址 127.0.0.1），但是 UNIX Domain Socket 用于 IPC 更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。这是因为，IPC 机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。UNIX Domain Socket 也提供面向流和面向数据包两种 API 接口，类似于 TCP 和 UDP，但是面向消息的 UNIX Domain Socket 也是可靠的，消息既不会丢失也不会顺序错乱。

UNIX Domain Socket 是全双工的，API 接口语义丰富，相比其它 IPC 机制有明显的优越性，目前已成为使用最广泛的 IPC 机制，比如 X Window 服务器和 GUI 程序之间就是通过 UNIXDomain Socket 通讯的。

使用 UNIX Domain Socket 的过程和网络 socket 十分相似，也要先调用 socket()创建一个 socket 文件描述符，address family 指定为 AF_UNIX，type 可以选择 SOCK_DGRAM 或 SOCK_STREAM，protocol 参数仍然指定为 0 即可。

UNIX Domain Socket 与网络 socket 编程最明显的不同在于地址格式不同，用结构体 sockaddr_un 表示，网络编程的 socket 地址是 IP 地址加端口号，而 UNIX Domain Socket 的地址是一个 socket 类型的文件在文件系统中的路径，这个 socket 文件由 bind()调用创建，如果调用 bind()时该文件已存在，则 bind()错误返回。

对比网络套接字地址结构和本地套接字地址结构：

```
struct sockaddr_in {
    __kernel_sa_family_t sin_family;        /* Address family */    地址结构类型
    __be16 sin_port;                        /* Port number */           端口号
    struct in_addr sin_addr;                /* Internet address */ IP 地址
};
struct sockaddr_un {
    __kernel_sa_family_t sun_family;        /* AF_UNIX */               地址结构类型
    char sun_path[UNIX_PATH_MAX];       /* pathname */          socket 文件名(含路径)
};
```

以下程序将 UNIX Domain socket 绑定到一个地址。

```
size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
#define offsetof(type, member) ((int)&((type *)0)->member)
```

server

```
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>

#define QLEN 10
/*
```

```
* Create a server endpoint of a connection.
* Returns fd if all OK, <0 on error.
*/
int serv_listen(const char *name)
{
    int fd, len, err, rval;
    struct sockaddr_un un;

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);
    /* in case it already exists */
    unlink(name);

    /* fill in socket address structure */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    /* bind the name to the descriptor */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }
    if (listen(fd, QLEN) < 0) { /* tell kernel we're a server */
        rval = -3;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
int serv_accept(int listenfd, uid_t *uidptr)
{
    int clifd, len, err, rval;
    time_t staletime;
    struct sockaddr_un un;
    struct stat statbuf;

    len = sizeof(un);
    if ((clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0)
        return(-1); /* often errno=EINTR, if signal caught */
```

```
    /* obtain the client's uid from its calling address */
    len -= offsetof(struct sockaddr_un, sun_path); /* len of pathname */
    un.sun_path[len] = 0; /* null terminate */

    if (stat(un.sun_path, &statbuf) < 0) {
        rval = -2;
        goto errout;
    }
    if (S_ISSOCK(statbuf.st_mode) == 0) {
        rval = -3; /* not a socket */
        goto errout;
    }
    if (uidptr != NULL)
        *uidptr = statbuf.st_uid; /* return uid of caller */
    /* we're done with pathname now */
    unlink(un.sun_path);
    return(clifd);

errout:
    err = errno;
    close(clifd);
    errno = err;
    return(rval);
}
int main(void)
{
    int lfd, cfd, n, i;
    uid_t cuid;
    char buf[1024];
    lfd = serv_listen("foo.socket");

    if (lfd < 0) {
        switch (lfd) {
            case -3:perror("listen"); break;
            case -2:perror("bind"); break;
            case -1:perror("socket"); break;
        }
        exit(-1);
    }
    cfd = serv_accept(lfd, &cuid);
    if (cfd < 0) {
        switch (cfd) {
            case -3:perror("not a socket"); break;
            case -2:perror("a bad filename"); break;
            case -1:perror("accept"); break;
        }
```

```
        exit(-1);
    }
    while (1) {
r_again:
        n = read(cfd, buf, 1024);
        if (n == -1) {
        if (errno == EINTR)
        goto r_again;
    }
    else if (n == 0) {
        printf("the other side has been closed.\n");
        break;
    }
    for (i = 0; i < n; i++)
        buf[i] = toupper(buf[i]);
        write(cfd, buf, n);
    }
    close(cfd);
    close(lfd);
    return 0;
}
```

client

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define CLI_PATH "/var/tmp/" /* +5 for pid = 14 chars */
/*
* Create a client endpoint and connect to a server.
* Returns fd if all OK, <0 on error.
*/
int cli_conn(const char *name)
{
    int fd, len, err, rval;
    struct sockaddr_un un;
```

```
    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    /* fill socket address structure with our address */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    sprintf(un.sun_path, "%s%05d", CLI_PATH, getpid());
    len = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);

    /* in case it already exists */
    unlink(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }

    /* fill socket address structure with server's address */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);
    if (connect(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -4;
        goto errout;
    }
return(fd);
    errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
int main(void)
{
    int fd, n;
    char buf[1024];

    fd = cli_conn("foo.socket");
    if (fd < 0) {
        switch (fd) {
            case -4:perror("connect"); break;
            case -3:perror("listen"); break;
            case -2:perror("bind"); break;
            case -1:perror("socket"); break;
        }
        exit(-1);
```

```
    }
    while (fgets(buf, sizeof(buf), stdin) != NULL) {
        write(fd, buf, strlen(buf));
        n = read(fd, buf, sizeof(buf));
        write(STDOUT_FILENO, buf, n);
    }
    close(fd);
    return 0;
}
```

其它常用函数(了解)

名字与地址转换

gethostbyname 根据给定的主机名，获取主机信息。

过时，仅用于 IPv4，且线程不安全。

```
#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>

extern int h_errno;

int main(int argc, char *argv[])
{
    struct hostent *host;
    char str[128];
    host = gethostbyname(argv[1]);
    printf("%s\n", host->h_name);

    while (*(host->h_aliases) != NULL)
        printf("%s\n", *host->h_aliases++);

    switch (host->h_addrtype) {
        case AF_INET:
            while (*(host->h_addr_list) != NULL)
            printf("%s\n", inet_ntop(AF_INET, (*host->h_addr_list++), str, sizeof(str)));
        break;
        default:
            printf("unknown address type\n");
```

```
            break;
    }
    return 0;
}
```

gethostbyaddr 函数。

此函数只能获取域名解析服务器的 url 和/etc/hosts 里登记的 IP 对应的域名。

```
#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>

extern int h_errno;

int main(int argc, char *argv[])
{
    struct hostent *host;
    char str[128];
    struct in_addr addr;

    inet_pton(AF_INET, argv[1], &addr);
    host = gethostbyaddr((char *)&addr, 4, AF_INET);
    printf("%s\n", host->h_name);

    while (*(host->h_aliases) != NULL)
        printf("%s\n", *host->h_aliases++);
    switch (host->h_addrtype) {
        case AF_INET:
            while (*(host->h_addr_list) != NULL)
            printf("%s\n", inet_ntop(AF_INET, (*host->h_addr_list++), str, sizeof(str)));
            break;
        default:
            printf("unknown address type\n");
            break;
    }
    return 0;
}
```

getservbyname

getservbyport

根据服务程序名字或端口号获取信息。使用频率不高。

getaddrinfo

getnameinfo

freeaddrinfo

可同时处理 IPv4 和 IPv6，线程安全的。

套接口和地址关联

getsockname

根据 accpet 返回的 sockfd，得到临时端口号

getpeername

根据 accpet 返回的 sockfd，得到远端链接的端口号，在 exec 后可以获取客户端信息。

Libevent 的使用

了解 libevent

什么是 libevent

Libevent 是一个用 C 语言编写的、轻量级的开源高性能事件通知库，主要有以下几个亮点：事件驱动（event-driven），高性能;轻量级，专注于网络，不如 ACE 那么臃肿庞大；源代码相当精炼、易读；跨平台，支持 Windows、Linux、*BSD 和 Mac Os；支持多种 I/O 多路复用技术， epoll、poll、dev/poll、select 和 kqueue 等；支持 I/O，定时器和信号等事件；注册事件优先级。

Chromium、Memcached、NTP、HTTPSQS 等著名的开源程序都使用 libevent 库，足见 libevent 的稳定。更多使用 libevent 的程序可以到 libevent 的官网查看。

Libevent 主要组成

libevent 包括事件管理、缓存管理、DNS、HTTP、缓存事件几大部分。事件管理包括各种 IO（socket）、定时器、信号等事件；缓存管理是指 evbuffer 功能；DNS 是 libevent 提供的一个异步 DNS 查询功能；HTTP 是 libevent 的一个轻量级 http 实现，包括服务器和客户端。libevent 也支持 ssl，这对于有安全需求的网络程序非常的重要，但是其支持不是很完善，比如 http server 的实现就不支持 ssl。

Libevent 的核心实现

Reactor（反应堆）模式是 libevent 的核心框架，libevent 以事件驱动，自动触发回调功能。之前介绍的 epoll 反应堆的源码，就是从 libevent 中抽取出来的。

安装 libevent

官方网站: http://libevent.org

源码下载主要分 2 个大版本：

1. 1.4.x 系列，较为早期版本，适合源码学习
2. 2.x 系列，较新的版本，代码量比 1.4 版本多很多，功能也更完善。

源码包的安装，以 2.0.22 版本为例，在官网可以下载到源码包 libevent-2.0.22-stable.tar.gz，基本安装步骤与第三方库源码包安装方式基本一致。

1. 解压源码包

```
itheima@ubuntu:~/install$
itheima@ubuntu:~/install$ tar zxvf libevent-2.0.22-stable.tar.gz
libevent-2.0.22-stable/
libevent-2.0.22-stable/evmap-internal.h
```

2. 进入到源码目录

```
itheima@ubuntu:~/install$
itheima@ubuntu:~/install$ cd libevent-2.0.22-stable/
itheima@ubuntu:~/install/libevent-2.0.22-stable$
itheima@ubuntu:~/install/libevent-2.0.22-stable$
```

3. 执行配置./configure，生成 makefile

```
itheima@ubuntu:~/install/libevent-2.0.22-stable$
itheima@ubuntu:~/install/libevent-2.0.22-stable$
itheima@ubuntu:~/install/libevent-2.0.22-stable$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
```

也可以指定具体路径，这样安装的时候，将统一安装到指定路径 例如：./configure --prefix=/usr/local/libevent,这样的好处是以后打包安装好的文件方便，不好的地方是由于安装的目录有可能不是系统头文件或库文件的目录，使用的时候需要增加 gcc 选项来包含头文件路径和库文件路径，以及需要解决动态库不能加载的问题，参考第三天内容，动态库不能加载怎么办?

4. 编译源码

```
itheima@ubuntu:~/install/libevent-2.0.22-stable$
itheima@ubuntu:~/install/libevent-2.0.22-stable$ make
test -d include/event2 || /bin/mkdir -p include/event2
/bin/sed -f ./make-event-config.sed < config.h > include/event2/event-confi
```

5. 编译后安装，输入本用户的密码

```
itheima@ubuntu:~/install/libevent-2.0.22-stable$
itheima@ubuntu:~/install/libevent-2.0.22-stable$ sudo make install
[sudo] itheima 的密码：
```

6. 安装后验证，简单的先编译一个文件

```
//01_getmethods.c
#include <event.h>
#include <stdio.h>

int main()
{
   char ** methods = event_get_supported_methods();//获取 libevent 后端支持的方法
   int i =0;
   for(i = 0;methods[i] != NULL ;i++)
   {
      printf("%s\n",methods[i]);
   }
   return 0;
}
```

编译：

```
itheima@ubuntu:~/linux/0512/day15$ gcc 01_getmethods.c -o 01_getmethods -levent
01_getmethods.c: In function 'main':
01_getmethods.c:6:23: warning: initialization from incompatible pointer type [-Wincompatible-pointer-types]
     char ** methods = event_get_supported_methods();
                       ^
```

可以忽略该警告，代表编译完成

执行

```
itheima@ubuntu:~/linux/0512/day15$ ./01_getmethods
epoll
poll
select
itheima@ubuntu:~/linux/0512/day15$
```

同时也能看到 libevent 在当前主机上后端支持的多路 IO 方法。

Libevent 的入门级使用

Libevent 的地基-event_base

在使用 libevent 的函数之前，需要先申请一个或 event_base 结构，相当于盖房子时的地基。在 event_base 基础上会有一个事件集合，可以检测哪个事件是激活的（就绪）。
通常情况下可以通过 event_base_new 函数获得 event_base 结构。

struct event_base *event_base_new(void);

申请到 event_base 结构指针可以通过 event_base_free 进行释放。

void event_base_free(struct event_base *);

如果 fork 出子进程，想在子进程继续使用 event_base，那么子进程需要对 event_base 重新初始化，函数如下：

int event_reinit(struct event_base *base);

对于不同系统而言，event_base 就是调用不同的多路 IO 接口去判断事件是否已经被激活，对于 linux 系统而言，核心调用的就是 epoll，同时支持 poll 和 select。

等待事件产生-循环等待 event_loop

Libevent 在地基打好之后，需要等待事件的产生，也就是等待想要等待的事件的激活，那么程序不能退出，对于 epoll 来说，我们需要自己控制循环，而在 libevent 中也给我们提供了 api 接口，类似 where(1)的功能.函数如下：

```
int event_base_loop(struct event_base *base, int flags);
```

flags 的取值：

```
#define EVLOOP_ONCE 0x01
```

只触发一次，如果事件没有被触发，阻塞等待

```
#define EVLOOP_NONBLOCK      0x02
```

非阻塞方式检测事件是否被触发，不管事件触发与否，都会立即返回

而大多数我们都调用 libevent 给我们提供的另外一个 api：

```
int event_base_dispatch(struct event_base *base);
```

调用该函数，相当于没有设置标志位的 event_base_loop。程序将会一直运行，直到没有需要检测的事件了，或者被结束循环的 api 终止。

```
int event_base_loopexit(struct event_base *base, const struct timeval *tv);
int event_base_loopbreak(struct event_base *base);
struct timeval {
    long    tv_sec;
    long    tv_usec;
};
```

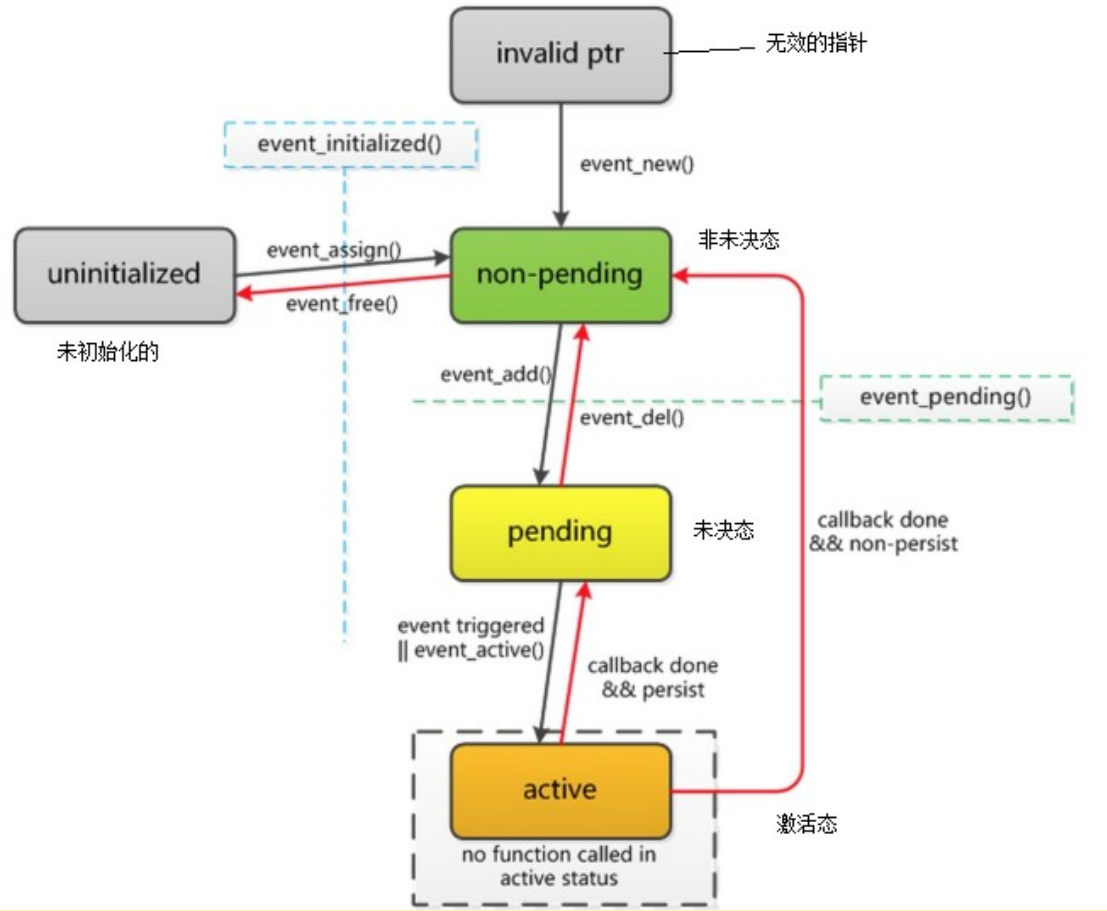
两个函数的区别是如果正在执行激活事件的回调函数，那么 event_base_loopexit 将在事件回调执行结束后终止循环（如果 tv 时间非 NULL，那么将等待 tv 设置的时间后立即结束循环），而 event_base_loopbreak

会立即终止循环。

事件驱动-event

事件驱动实际上是 libevent 的核心思想，本小节主要介绍基本的事件 event。

主要的状态转化：



主要几个状态：

无效的指针 此时仅仅是定义了 struct event *ptr；

非未决：相当于创建了事件，但是事件还没有处于被监听状态，类似于我们使用 epoll 的时候定义了 struct epoll_event ev 并且对 ev 的两个字段进行了赋值，但是此时尚未调用 epoll_ctl。

未决：就是对事件开始监听，暂时未有事件产生。相当于调用 epoll_ctl。

激活：代表监听的事件已经产生，这时需要处理，相当于我们 epoll 所说的事件就绪。

Libevent 的事件驱动对应的结构体为 struct event，对应的函数在图上也比较清晰，下面介绍一下主要的函数

```
1. struct event *event_new(struct event_base *base, evutil_socket_t fd, short events, event_callback_fn cb, void *arg);
```

event_new 负责新创建 event 结构指针，同时指定对应的地基 base，还有对应的文件描述符，事件，以及回调函数和回调函数的参数。参数说明：

base 对应的根节点

fd 要监听的文件描述符

events 要监听的事件

```
#define  EV_TIMEOUT      0x01  //超时事件
#define  EV_READ         0x02 //读事件
#define  EV_WRITE        0x04  //写事件
#define  EV_SIGNAL       0x08   //信号事件
#define  EV_PERSIST       0x10  //周期性触发
#define  EV_ET           0x20 //边缘触发，如果底层模型支持
```

cb 回调函数，原型如下：

```
typedef void (*event_callback_fn)(evutil_socket_t fd, short events, void *arg);
```

arg 回调函数的参数

2. int event_add(struct event *ev, const struct timeval *timeout);

   将非未决态事件转为未决态，相当于调用 epoll_ctl 函数，开始监听事件是否产生。

   参数说明：

   Ev 就是前面 event_new 创建的事件

   Timeout 限时等待事件的产生，也可以设置为 NULL，没有限时。

3. int event_del(struct event *ev);

   将事件从未决态变为非未决态，相当于 epoll 的下树（epoll_ctl 调用 EPOLL_CTL_DEL 操作）操作。

4. void event_free(struct event *ev);

   释放 event_new 申请的 event 节点。

基于以上函数，可以写一个简易的事件驱动的网络服务器端程序。

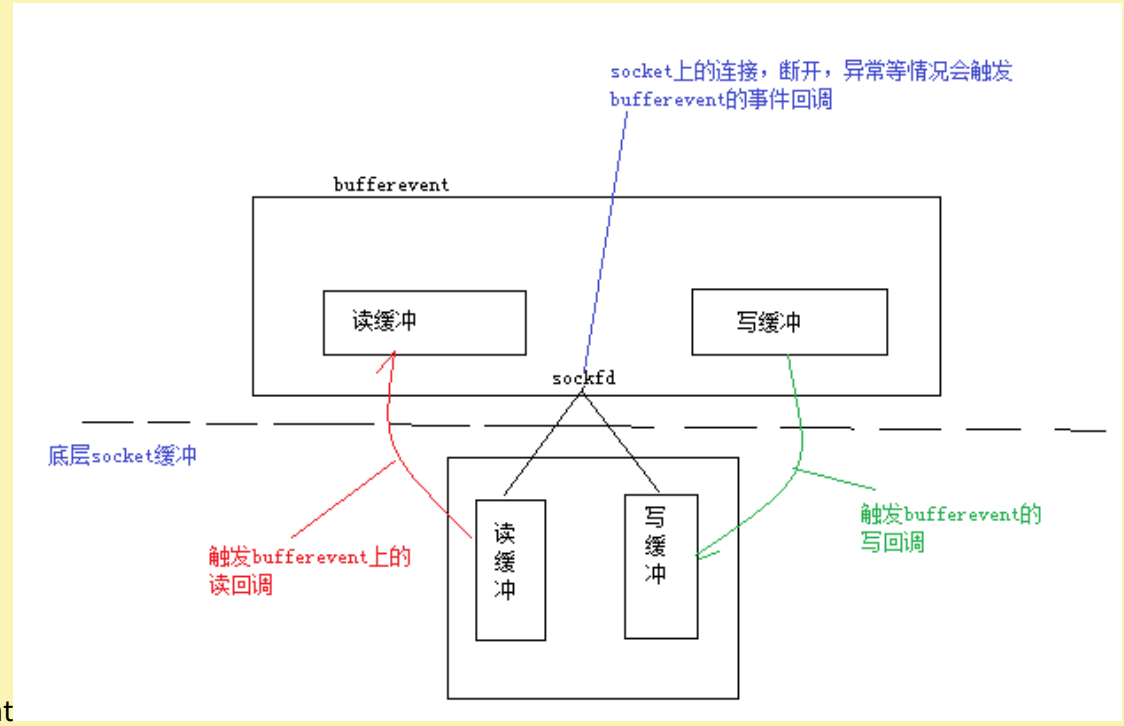
```
//02_event_server.c
//用 libevent 写一下简易的服务器
#include <event.h>
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>

struct event *readev = NULL;

void readcb(evutil_socket_t fd, short event, void * arg)
{
    //处理读事件
    char buf[256] = { 0 };
```

```
	int ret = read(fd, buf, sizeof(buf));
	if (ret < 0){
		perror("read err");
		close(fd);
		event_del(readev);
	}
	else if (ret == 0){
		printf("client closed\n");
		close(fd);
		event_del(readev);
	}
	else{
		write(fd, buf, ret);//反射
	}
}
void conncb(evutil_socket_t fd, short event, void * arg)
{
	//处理连接
	struct event_base *base = (struct event_base*)arg;
	struct sockaddr_in client;
	socklen_t  len = sizeof(client);
	int cfd = accept(fd, (struct sockaddr*)&client, &len);
	if (cfd > 0){
		//连接成功
		//需要将新的文件描述符上树
		readev = event_new(base, cfd, EV_READ | EV_PERSIST, readcb, base);
		event_add(readev, NULL);
	}
}

int main()
{
	int fd = socket(AF_INET, SOCK_STREAM, 0);
	struct sockaddr_in serv;
	bzero(&serv, sizeof(serv));
	serv.sin_addr.s_addr = htonl(INADDR_ANY);
	serv.sin_port = htons(8888);
	serv.sin_family = AF_INET;

	bind(fd, (struct sockaddr*)&serv, sizeof(serv));

	listen(fd, 120);
	struct event_base *base = event_base_new();//创建根节点

	//struct event *event_new(struct event_base *, evutil_socket_t, short, event_callback_fn, void *);
	struct event *connev = event_new(base, fd, EV_READ | EV_PERSIST, conncb, base);
```

```
    event_add(connev, NULL);//开始监听
    //循环
    event_base_dispatch(base);
    event_base_free(base);//释放
    event_free(connev);
    event_free(readev);
    return 0;
}
```

自带 buffer 的事件-bufferevent

Bufferevent 实际上也是一个 event，只不过比普通的 event 高级一些，它的内部有两个缓冲区，以及一个文件描述符（网络套接字）。我们都知道一个网络套接字有读和写两个缓冲区，bufferevent 同样也带有两个缓冲区，还有就是 libevent 事件驱动的核心回调函数，那么四个缓冲区以及触发回调的关系如下：



bufferevent

bufferevent 有三个回调函数：

1. 读回调 – 当 bufferevent 将底层读缓冲区的数据读到自身的读缓冲区时触发读事件回调
2. 写回调 – 当 bufferevent 将自身写缓冲的数据写到底层写缓冲区的时候出发写事件回调
3. 事件回调 – 当 bufferevent 绑定的 socket 连接，断开或者异常的时候触发事件回调

主要使用的函数如下：

1. struct bufferevent *bufferevent_socket_new(struct event_base *base, evutil_socket_t fd, int options);

bufferevent_socket_new 对已经存在 socket 创建 bufferevent 事件，可用于后面讲到的链接监听器的回调函数中，参数说明：

base – 对应根节点

fd -- 文件描述符

options – bufferevent 的选项

BEV_OPT_CLOSE_ON_FREE -- 释放 bufferevent 自动关闭底层接口

BEV_OPT_THREADSAFE -- 使 bufferevent 能够在多线程下是安全的

2. **int bufferevent_socket_connect(struct bufferevent *bev, struct sockaddr *serv, int socklen);**

bufferevent_socket_connect 封装了底层的 socket 与 connect 接口，通过调用此函数，可以将 bufferevent 事件与通信的 socket 进行绑定，参数如下：

bev – 需要提前初始化的 bufferevent 事件

serv – 对端的 ip 地址，端口，协议的结构指针

socklen – 描述 serv 的长度

3. **void bufferevent_free(struct bufferevent *bufev);**

释放 bufferevent

4. **void bufferevent_setcb(struct bufferevent *bufev,**
bufferevent_data_cb readcb, bufferevent_data_cb writecb,
bufferevent_event_cb eventcb, void *cbarg);

bufferevent_setcb 用于设置 bufferevent 的回调函数，readcb，writecb，eventcb 分别对应了读回调，写回调，事件回调，cbarg 代表回调函数的参数。

回调函数的原型：

typedef void (*bufferevent_data_cb)(struct bufferevent *bev, void *ctx);

typedef void (*bufferevent_event_cb)(struct bufferevent *bev, short what, void *ctx);

- What 代表对应的事件 BEV_EVENT_EOF, BEV_EVENT_ERROR , BEV_EVENT_TIMEOUT, BEV_EVENT_CONNECTED

5. **int bufferevent_write(struct bufferevent *bufev, const void *data, size_t size);**
bufferevent_write是将data的数据写到bufferevent的写缓冲区
6. **int bufferevent_write_buffer(struct bufferevent *bufev, struct evbuffer *buf);**
bufferevent_write_buffer 是将数据写到写缓冲区另外一个写法，实际上bufferevent的内部的两个缓冲区结构就是 struct evbuffer。
7. **size_t bufferevent_read(struct bufferevent *bufev, void *data, size_t size);**
bufferevent_read 是将bufferevent的读缓冲区数据读到data中，同时将读到的数据从bufferevent的读缓冲清除。
8. **int bufferevent_read_buffer(struct bufferevent *bufev, struct evbuffer *buf);**
bufferevent_read_buffer 将bufferevent读缓冲数据读到buf中，接口的另外一种。
9. **int bufferevent_enable(struct bufferevent *bufev, short event);**
10. **int bufferevent_disable(struct bufferevent *bufev, short event);**
bufferevent_enable与bufferevent_disable是设置事件是否生效，如果设置为disable，事件回调将不会被触发。

链接监听器-evconnlistener

链接监听器封装了底层的 socket 通信相关函数，比如 socket，bind，listen，accept 这几个函数。链接监听器创建后实际上相当于调用了 socket，bind，listen，此时等待新的客户端连接到来，如果有新的客户端连接，那么内部先进行 accept 处理，然后调用用户指定的回调函数。可以先看看函数原型，了解一下它是怎么运作的：

1. struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
   evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
   const struct sockaddr *sa, int socklen);
   evconnlistener_new_bind 是在当前没有套接字的情况下对链接监听器进行初始化，看最后 2 个参数实际上就是 bind 使用的关键参数，backlog 是 listen 函数的关键参数（略有不同的是，如果 backlog 是-1，那么监听器会自动选择一个合适的值，如果填 0，那么监听器会认为 listen 函数已经被调用过了），ptr 是回调函数的参数，cb 是有新连接之后的回调函数，但是注意这个回调函数触发的时候，链接器已经处理好新连接了，并将与新连接通信的描述符交给回调函数。Flags 需要参考几个值：
   LEV_OPT_LEAVE_SOCKETS_BLOCKING 文件描述符为阻塞的
   LEV_OPT_CLOSE_ON_FREE 关闭时自动释放
   LEV_OPT_REUSEABLE 端口复用
   LEV_OPT_THREADSAFE 分配锁，线程安全
2. struct evconnlistener *evconnlistener_new(struct event_base *base,
   evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
   evutil_socket_t fd);
   evconnlistener_new 函数与前一个函数不同的地方在与后 2 个参数，使用本函数时，认为 socket 已经初始化好，并且 bind 完成，甚至也可以做完 listen，所以大多数时候，我们都可以使用第一个函数。
3. 两个函数的回调函数
   typedef void (*evconnlistener_cb)(struct evconnlistener *evl, evutil_socket_t fd, struct sockaddr *cliaddr, int socklen, void *ptr);
   主要回调函数 fd 参数会与客户端通信的描述符，并非是等待连接的监听的那个描述符，所以 cliaddr 对应的也是新连接的对端地址信息，已经是 accept 处理好的。
4. void evconnlistener_free(struct evconnlistener *lev);
   释放链接监听器
5. int evconnlistener_enable(struct evconnlistener *lev);
   使链接监听器生效
6. int evconnlistener_disable(struct evconnlistener *lev);
   使链接监听器失效

如果上述函数都较为了解了，可以尝试去看懂 hello-world.c 的代码，在安装包的 sample 目录下，其中有涉及到信号的函数，看看自己能否找到函数的原型在哪？实际上就是一个宏定义，也是我们之前介绍的 event_new 函数，只是对应一个信号事件而已，处理机制略有不同。

```
#define evsignal_new(b, x, cb, arg)                 \
    event_new((b), (x), EV_SIGNAL|EV_PERSIST, (cb), (arg))
```

基于 bufferevent 的服务器和客户端实现

基于 bufferevent 的服务器

```
// 03_bufferevent_server.c
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/listener.h>
#include <event2/util.h>
#include <event2/event.h>
#include <ctype.h>

static const char MESSAGE[] = "Hello, World!\n";

static const int PORT = 9995;

static void listener_cb(struct evconnlistener *, evutil_socket_t,
   struct sockaddr *, int socklen, void *);
static void conn_writecb(struct bufferevent *, void *);
static void conn_readcb(struct bufferevent *, void *);
static void conn_eventcb(struct bufferevent *, short, void *);
static void signal_cb(evutil_socket_t, short, void *);

int
main(int argc, char **argv)
{
	struct event_base *base;//根节点定义
	struct evconnlistener *listener;//监听器定义
	struct event *signal_event;//信号事件

	struct sockaddr_in sin;

	base = event_base_new();//创建根节点
	if (!base) {
		fprintf(stderr, "Could not initialize libevent!\n");
		return 1;
	}

	memset(&sin, 0, sizeof(sin));
```

```
    sin.sin_family = AF_INET;
    sin.sin_port = htons(PORT);

  //创建监听器-端口复用-关闭自动释放
    listener = evconnlistener_new_bind(base, listener_cb, (void *)base,
      LEV_OPT_REUSEABLE|LEV_OPT_CLOSE_ON_FREE, -1,
      (struct sockaddr*)&sin,
      sizeof(sin));

    if (!listener) {
        fprintf(stderr, "Could not create a listener!\n");
        return 1;
    }

  //定义信号回调事件 -SIGINT
    signal_event = evsignal_new(base, SIGINT, signal_cb, (void *)base);
  //event_add 上树 -开始监听信号事件
    if (!signal_event || event_add(signal_event, NULL)<0) {
        fprintf(stderr, "Could not create/add a signal event!\n");
        return 1;
    }

  //循环等待事件
    event_base_dispatch(base);
  //释放链接侦听器
    evconnlistener_free(listener);
    event_free(signal_event);
    event_base_free(base);

    printf("done\n");
    return 0;
}
//链接监听器帮助处理了 accept 连接，得到新的文件描述符，作为参数传入
static void
listener_cb(struct evconnlistener *listener, evutil_socket_t fd,
  struct sockaddr *sa, int socklen, void *user_data)
{
  printf("---call------%s----\n",__FUNCTION__);
    struct event_base *base = user_data;
    struct bufferevent *bev;//定义 bufferevent 事件

  //创建新的 bufferevent 事件，对应的与客户端通信的 socket
    bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FREE);
    if (!bev) {
        fprintf(stderr, "Error constructing bufferevent!");
        event_base_loopbreak(base);
```

```
            return;
        }
    //设置回调函数 只设置了写回调和事件产生回调
        bufferevent_setcb(bev, conn_readcb, conn_writecb, conn_eventcb, NULL);
    //启用读写缓冲区
        bufferevent_enable(bev, EV_WRITE|EV_READ);
    //禁用读缓冲
        //bufferevent_disable(bev, EV_READ);
    //将 MESSAGE 写到输出缓冲区
        //bufferevent_write(bev, MESSAGE, strlen(MESSAGE));
}

//自定义读回调函数
static void
conn_readcb(struct bufferevent *bev, void *user_data)
{
    printf("---calll-----%s\n",__FUNCTION__);
    //何时被触发？读入缓冲区有数据的时候，非底层的
    char buf[256]={0};
    size_t ret = bufferevent_read(bev, buf, sizeof(buf));
    if(ret > 0){
        //转为大写
        int i;
        for(i = 0; i < ret ; i ++){
            buf[i] = toupper(buf[i]);
        }
        //写到 bufferevent 的输出缓冲区
        bufferevent_write(bev, buf, ret);
    }
}

static void
conn_writecb(struct bufferevent *bev, void *user_data)
{

    printf("---call------%s----\n",__FUNCTION__);
        struct evbuffer *output = bufferevent_get_output(bev);
        if (evbuffer_get_length(output) == 0) {
                printf("flushed answer\n");
        //      bufferevent_free(bev);
        }
}

static void
conn_eventcb(struct bufferevent *bev, short events, void *user_data)
{
```

```
   printf("---call------%s----\n",__FUNCTION__);
      if (events & BEV_EVENT_EOF) {
            printf("Connection closed.\n");
      } else if (events & BEV_EVENT_ERROR) {
            printf("Got an error on the connection: %s\n",
              strerror(errno));/*XXX win32*/
      }
      /* None of the other events can happen here, since we haven't enabled
       * timeouts */
      bufferevent_free(bev);
}

static void
signal_cb(evutil_socket_t sig, short events, void *user_data)
{
   printf("---call------%s----\n",__FUNCTION__);
      struct event_base *base = user_data;
      struct timeval delay = { 2, 0 };//设置延迟时间 2s

      printf("Caught an interrupt signal; exiting cleanly in two seconds.\n");

      event_base_loopexit(base, &delay);//延时 2s 退出
}
```

基于 bufferevent 的客户端

```
//bufferevent 建立客户端的过程
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <event.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/util.h>

int tcp_connect_server(const char* server_ip, int port);
void cmd_msg_cb(int fd, short events, void* arg);
void server_msg_cb(struct bufferevent* bev, void* arg);
```

```
void event_cb(struct bufferevent *bev, short event, void *arg);

int main(int argc, char** argv)
{
   if( argc < 3 )
   {
      //两个参数依次是服务器端的 IP 地址、端口号
      printf("please input 2 parameter\n");
      return -1;
   }
   //创建根节点
   struct event_base *base = event_base_new();
   //创建并且初始化 buffer 缓冲区
   struct bufferevent* bev = bufferevent_socket_new(base, -1,
                                 BEV_OPT_CLOSE_ON_FREE);

   //监听终端输入事件
   struct event* ev_cmd = event_new(base, STDIN_FILENO,
                          EV_READ | EV_PERSIST,
                          cmd_msg_cb, (void*)bev);
   //上树 开始监听标准输入的读事件
   event_add(ev_cmd, NULL);

   struct sockaddr_in server_addr;
   memset(&server_addr, 0, sizeof(server_addr) );
   server_addr.sin_family = AF_INET;
   server_addr.sin_port = htons(atoi(argv[2]));
   //将 ip 地址转换为网络字节序
   inet_aton(argv[1], &server_addr.sin_addr);

   //连接到 服务器 ip 地址和端口 初始化了 socket 文件描述符
   bufferevent_socket_connect(bev, (struct sockaddr *)&server_addr,
                       sizeof(server_addr));
   //设置 buffer 的回调函数 主要设置了读回调 server_msg_cb ,传入参数是标准输入的读事件
   bufferevent_setcb(bev, server_msg_cb, NULL, event_cb, (void*)ev_cmd);
   bufferevent_enable(bev, EV_READ | EV_PERSIST);

   event_base_dispatch(base);

   printf("finished \n");
   return 0;
}
//终端输入回调
void cmd_msg_cb(int fd, short events, void* arg)
{
   char msg[1024];
```

```
    int ret = read(fd, msg, sizeof(msg));
    if( ret < 0 )
    {
        perror("read fail ");
        exit(1);
    }

    struct bufferevent* bev = (struct bufferevent*)arg;

    //把终端的消息发送给服务器端
    bufferevent_write(bev, msg, ret);
}

void server_msg_cb(struct bufferevent* bev, void* arg)
{
    char msg[1024];

    size_t len = bufferevent_read(bev, msg, sizeof(msg));
    msg[len] = '\0';

    printf("recv %s from server\n", msg);
}

void event_cb(struct bufferevent *bev, short event, void *arg)
{

    if (event & BEV_EVENT_EOF)
        printf("connection closed\n");
    else if (event & BEV_EVENT_ERROR)
        printf("some other error\n");
    else if( event & BEV_EVENT_CONNECTED)
    {
        printf("the client has connected to server\n");
        return ;
    }

    //这将自动 close 套接字和 free 读写缓冲区
    bufferevent_free(bev);
    //释放 event 事件 监控读终端
    struct event *ev = (struct event*)arg;
    event_free(ev);
}
```

网络编程阶段项目

项目目标

实现一个 web 服务器

可以在浏览器页面请求资源页面

Web 服务器开发准备

为了编写 web 服务器，我们需要学会编写 html 页面，以及掌握部分 http 协议知识，这两部分内容将在接下来进行介绍。这两个准备工作之后，还需要知道 web 服务器的通信流程是什么？还需要思考如何支持多浏览器并发访问!

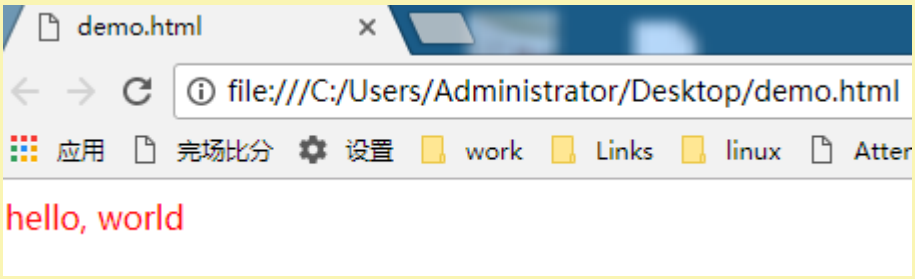
Html 语言基础

Html 简介

Html（Hyper Texture Markup Language）是超文本标记语言，在计算机中以 .html 或者.htm 作为扩展名,可以被浏览器识别,就是经常见到的网页.

Html 的语法非常简洁,比较松散,以相应的英语单词关键字进行组合,html 标签不区分大小写,标签大多数成对出现,有开始,有结束,例如 <html></html>,但是并没有要求必须成对出现.同时也有固定的短标签,例如
,<hr/>.

学习 html 基本可以认为就是学习各种标签,标签也可以设置属性,例如 hello, world,示例中 color 代表标签的颜色属性,red 代表标签是红色字体,hello,world 为实际显示的内容.可以新建一个文本文档,然后将后缀名修改.html 文件,用代码编辑器打开该 html 文件可以编辑文件(例如 notepad++),将上述内容保存到文件中,双击该文件可以看到如下效果:



Html 的组成可以分为如下部分:

1. <!doctype html> 声明文档类型,可以不写
2. <html> 开始 和</html> 结束,属于 html 的根标签
3. <head></head> 头部标签,头部标签内一般有 <title></title>
4. <body></body> 主体标签,一般用于显示内容

例如:

```
<html>
    <head>
        <title>这是一个标题</title>
    </head>

    <body>
        <font color="red" size="5">hello, world</font>
    </body>
</html>
```

如果想要添加注释,可以使用 <!—我是注释 -->的方式.

也可以指定页面类型和字符编码,下面设置页面类型为 html,并且字符编码为 utf8

<meta http-equiv="content-Type" content="text/html; charset=utf8">

Html 标签属性,可以双引号,单引号,或者不写

Html 标签介绍

题目标签

共有 6 种,<h1>,<h2>,…<h6>,其中<h1>最大,<h6>最小

文本标签

<font>标签,可以设置颜色和字体大小属性

颜色表示方法(可以参考网站: http://tool.oschina.net/commons?type=3):

1. 英文单词 red green blue …
2. 使用 16 进制的形式表示颜色:#ffffff
3. 使用 rgb(255,255,0)

字体大小可以使用 size 属性,大小范围为 1-7,其中 7 最大,1 最小.

有时候需要使用换行标签 ,这是一个短标签
与之对应另外还有一个水平线也是短标签, <hr/>,水平线也可以设置颜色和大小

列表标签

列表标签分无序列表和有序列表,分别对应<ul>和<ol>.
无序列表的格式如下:

```
<ul>
    <li>列表内容 1</li>
    <li>列表内容 2</li>
    …
</ul>
```

无序列表可以设置 type 属性:
实心圆圈:type=disc
空心圆圈:type=circle
小方块: type=square
有序列表的格式如下:

```
<ol>
    <li>列表内容 1</li>
    <li>列表内容 2</li>
    …
</ol>
```

有序列表同样可以设置 type 属性
数字:type=1,也是默认方式
英文字母:type=a 或 type=A
罗马数字:type=i 或 type=I

图片标签

图片标签使用<img>,内部需要设置若干属性,可以不必写结束标签
属性:

1. src=”3.gif” 图片来源,必写
2. alt=”小岳岳” 图片不显示时,显示的内容
3. title=”我的天呐” 鼠标移动到图片上时显示的文字
4. width=”600” 图片显示的宽度
5. height=”400” 图片显示的高度

例如:

```
<img src="3.gif" alt="小岳岳" title="我的天呐！" width="300" height="200" />
```

注意:当图片未定义宽高,图片百分百比例显示,如果只改变图片宽度或者高度,会等比例缩放

超链接标签

超链接标签使用<a>,同样需要设置属性表明要链接到哪里.
属性:
1. href=”http://www.itcast.cn”,前往地址,必填,注意要写 http://
2. title=”前往传智” 鼠标移动到链接上时显示的文字
3. target=”_self”或者”_blank”,_self 是默认值,在自身页面打开,_blank 是新开页面前往连接地址
示例:
来传智

当我们访问某个网站的时候,当请求的资源不存在,经常会给我们报告一个错误,显示为 404 错误,一般会给请求用户返回一个错误页,大家可以自行尝试一下编写一个我们自己的错误页.

http 超文本传输协议

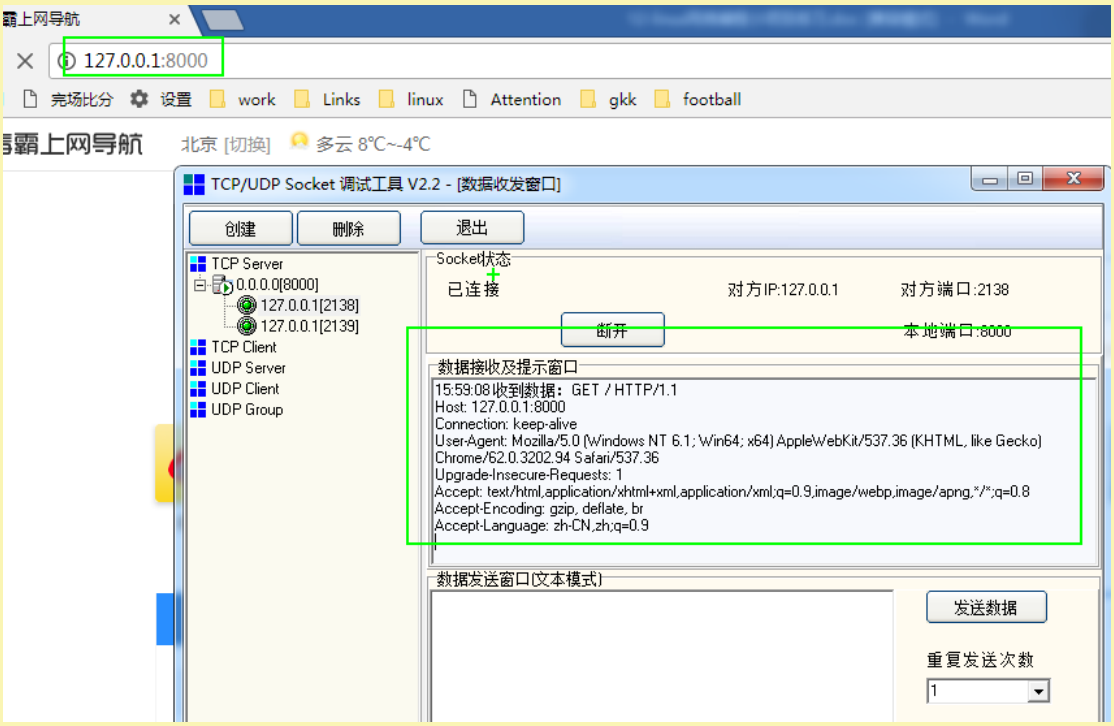
http 协议和 html 前面的 ht 都是超文本的意思,所以 http 与 html 是配合非常紧密的一对,我们可以认为 http 就是为了传输 html 这样的文件,http 位于应用层,侧重于解释.

http 协议对消息区分可以分为请求消息和响应消息.

http 请求消息

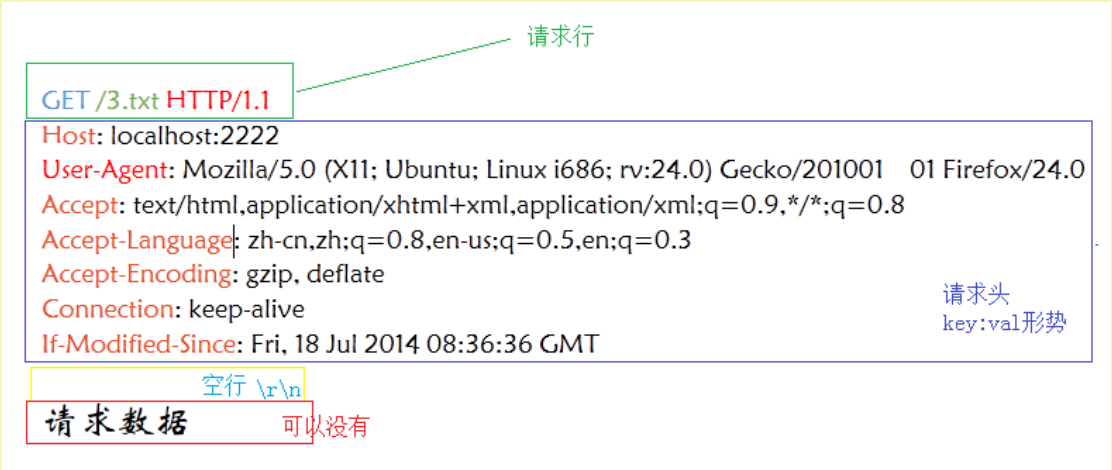
我们要开发的服务器与浏览器通信采用的就是 http 协议,在浏览器想访问一个资源的时候,在浏览器输入访问地址(例如 http://127.0.0.1:8000),地址输入完成后当敲击回车键的时候,浏览器就将请求消息发送给服务器
我们可以先用测试工具创建一个 socket 服务器

之后通过浏览器请求地址,就会看到浏览器发送过来的请求消息



这个消息看起来很乱很复杂,对应的就是我们说的请求消息.
请求消息分为四部分内容:

1. 请求行 说明请求类型,要访问的资源,以及使用的 http 版本
2. 请求头 说明服务器使用的附加信息,都是键值对,比如表明浏览器类型
3. 空行 不能省略-而且是\r\n,包括请求行和请求头都是以\r\n 结尾
4. 请求数据 表明请求的特定数据内容,可以省略-如登陆时,会将用户名和密码内容作为请求数据



请求类型

http 协议有很多种请求类型,对我们来说常见的用的最多的是 get 和 post 请求。常见的请求类型如下:

1. Get 请求指定的页面信息，并返回实体主体
2. Post 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。
3. Head 类似于 get 请求，但是响应消息没有内容，只是获得报头
4. Put 从客户端向浏览器传送的数据取代指定的文档内容

5. Delete 请求服务器删除指定的页面
6. Connect HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器
7. Options 允许客户端查看浏览器的性能
8. Trace 回显服务器收到的请求，主要用于测试和诊断

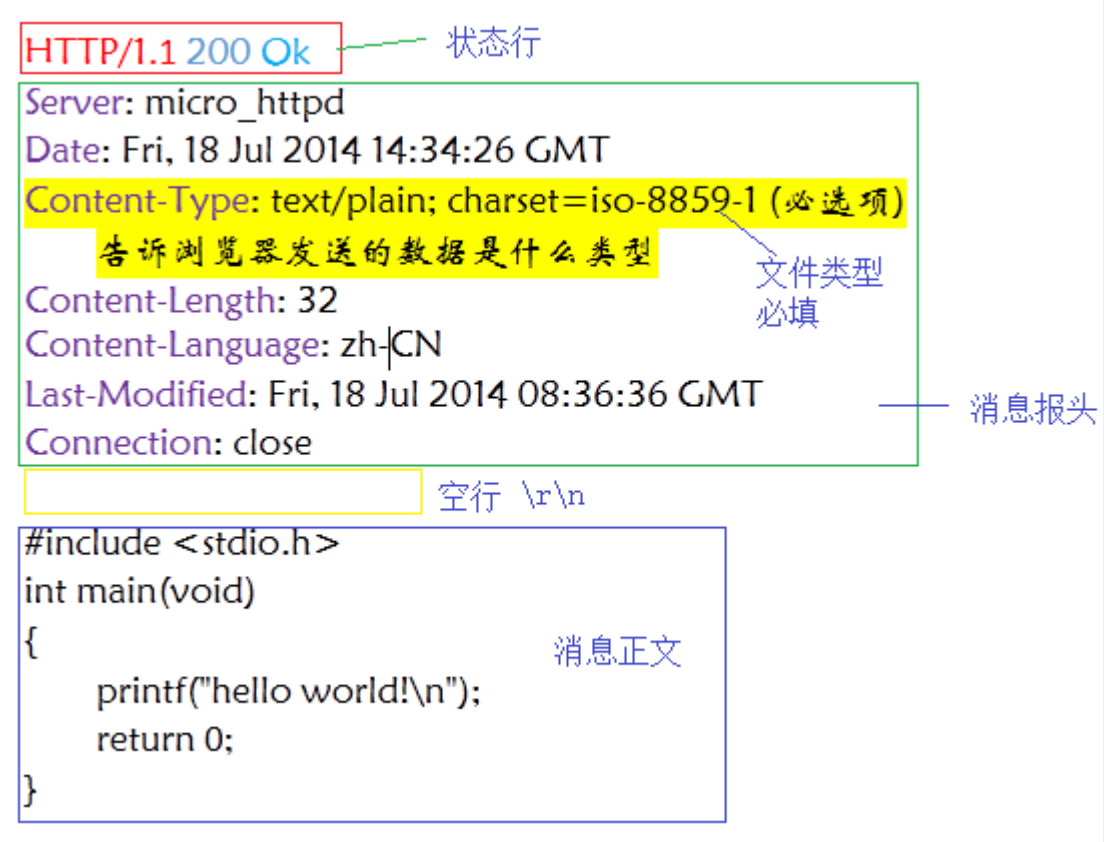
get 和 post 请求都是请求资源,而且都会提交数据,如果提交密码信息用 get 请求,就会明文显示,而 post 则不会显示出涉密信息.

http 响应消息

响应消息是代表服务器收到请求消息后,给浏览器做的反馈,所以响应消息是服务器发送给浏览器的,响应消息也分为四部分:

1. 状态行 包括 http 版本号,状态码,状态信息
2. 消息报头 说明客户端要使用的一些附加信息,也是键值对
3. 空行 \r\n 同样不能省略
4. 响应正文 服务器返回给客户端的文本信息

示例:



http 常见状态码

http状态码由三位数字组成,第一个数字代表响应的类别,有五种分类:

1. 1xx 指示信息--表示请求已接收，继续处理
2. 2xx 成功--表示请求已被成功接收、理解、接受
3. 3xx 重定向--要完成请求必须进行更进一步的操作
4. 4xx 客户端错误--请求有语法错误或请求无法实现
5. 5xx 服务器端错误--服务器未能实现合法的请求

常见的状态码如下:

- 200 OK 客户端请求成功
- 301 Moved Permanently 重定向
- 400 Bad Request 客户端请求有语法错误，不能被服务器所理解
- 401 Unauthorized 请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用
- 403 Forbidden 服务器收到请求，但是拒绝提供服务
- 404 Not Found 请求资源不存在，eg：输入了错误的 URL
- 500 Internal Server Error 服务器发生不可预期的错误
- 503 Server Unavailable 服务器当前不能处理客户端的请求，一段时间后可能恢复正常

http 常见文件类型分类

http 与浏览器交互时,为使浏览器能够识别文件信息,所以需要传递文件类型,这也是响应消息必填项,常见的类型如下:

- 普通文件: text/plain; charset=utf-8
- *.html: text/html; charset=utf-8
- *.jpg: image/jpeg
- *.gif: image/gif
- *.png: image/png
- *.wav: audio/wav
- *.avi: video/x-msvideo
- *.mov: video/quicktime

- *.mp3: audio/mpeg

特别说明

charset=iso-8859-1 西欧的编码，说明网站采用的编码是英文；

charset=gb2312 说明网站采用的编码是简体中文；

charset=utf-8 代表世界通用的语言编码；可以用到中文、韩文、日文等世界上所有语言编码上

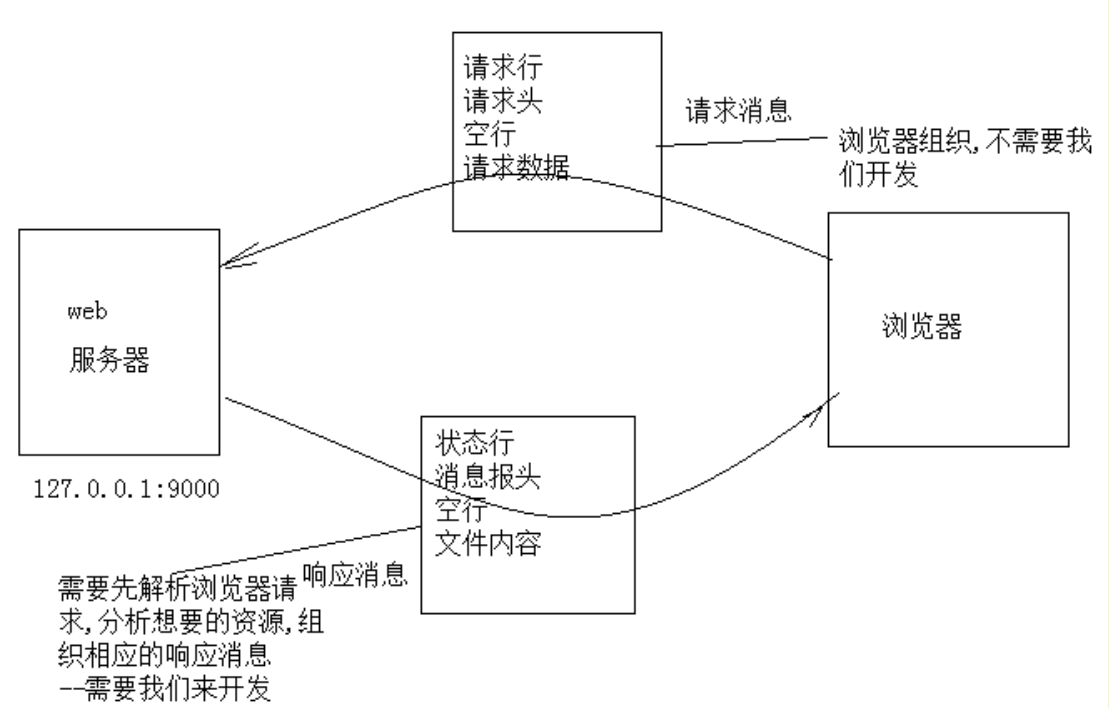
charset=euc-kr 说明网站采用的编码是韩文；

charset=big5 说明网站采用的编码是繁体中文；

web 服务器开发

我们要开发 web 服务器已经明确要使用 http 协议传送 html 文件,那么我们如何搭建我们的服务器呢?注意 http 只是应用层协议,我们仍然需要选择一个传输层的协议来完成我们的传输数据工作,所以开发协议选择是 TCP+HTTP,也就是说服务器搭建浏览依照 TCP,对数据进行解析和响应工作遵循 HTTP 的原则.

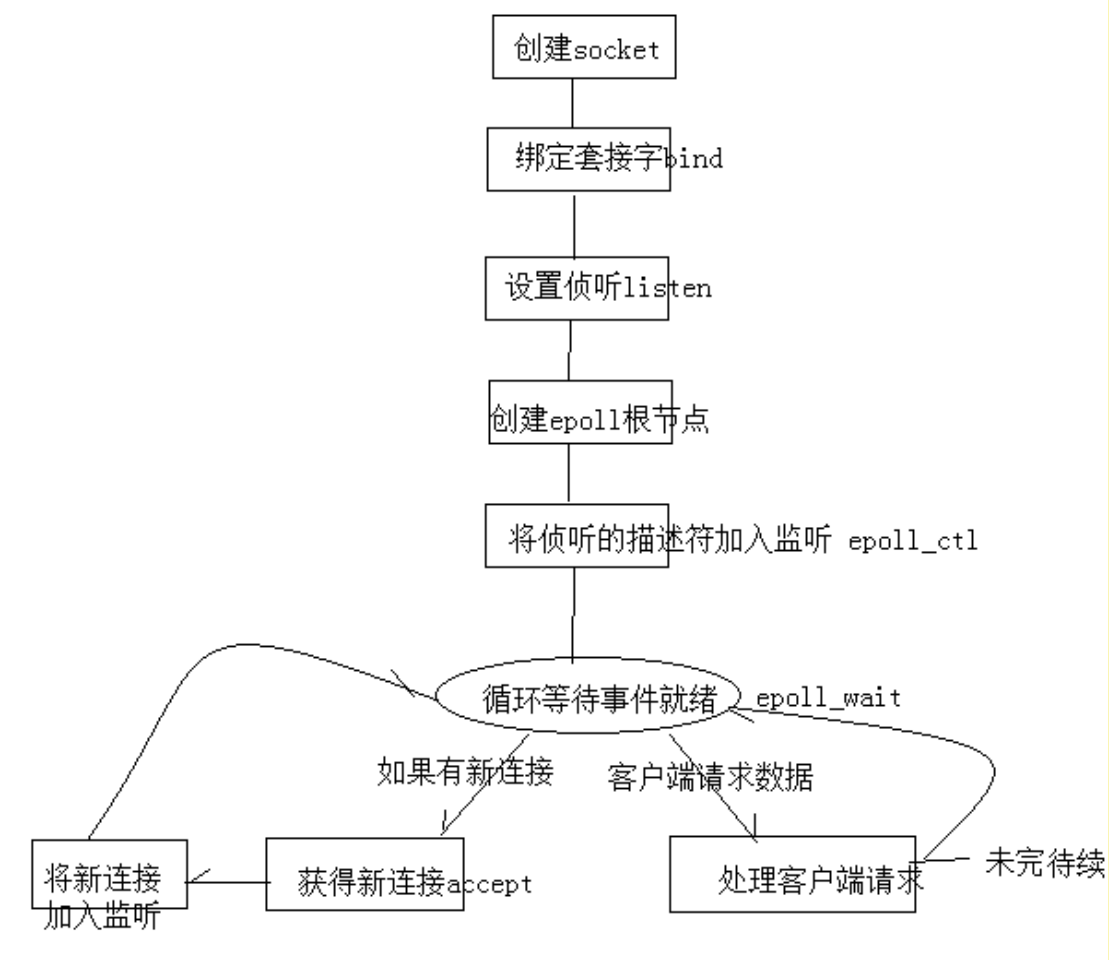
这样我们的思路很清晰,编写一个 TCP 并发服务器,只不过收发消息的格式采用的是 HTTP 协议,如下图:



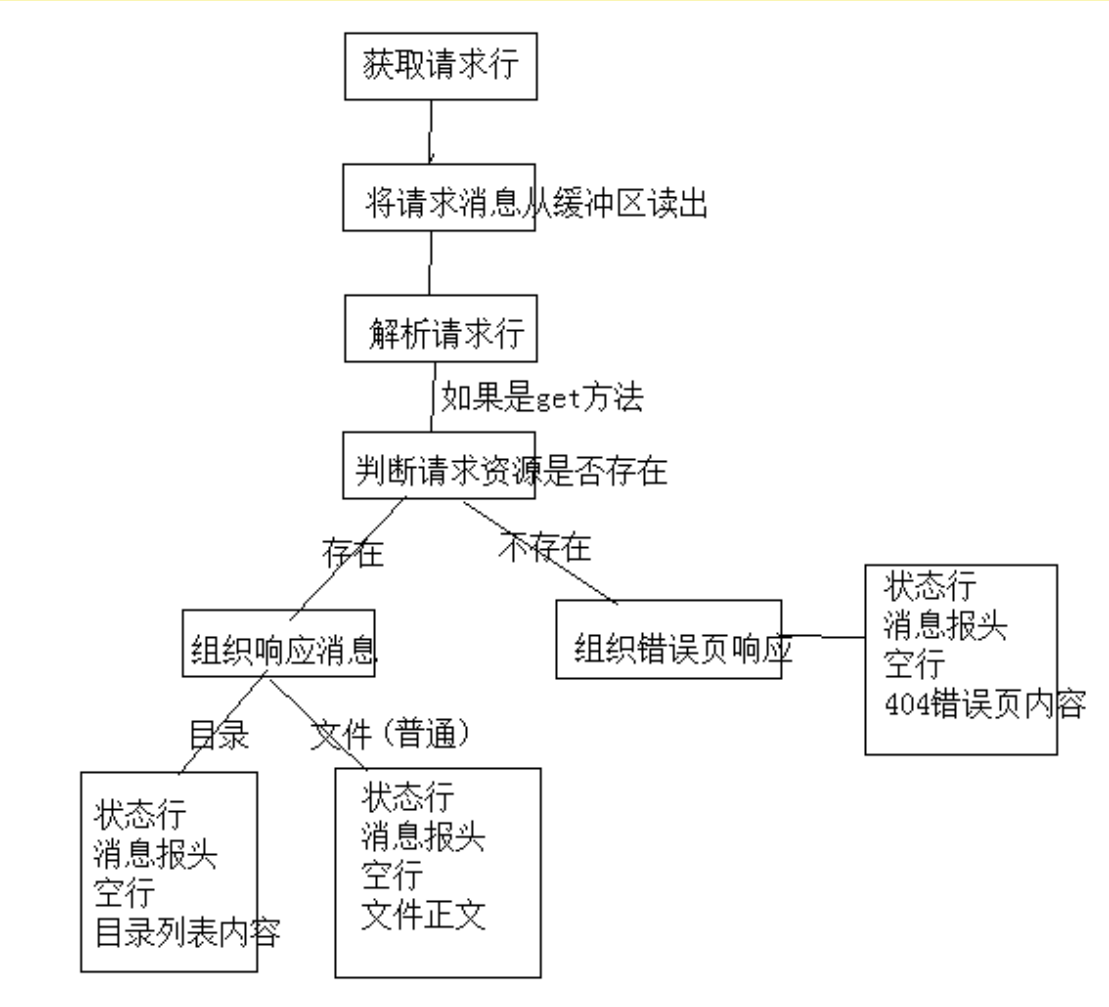
为了支持并发服务器,我们可以有多个选择,比如多进程服务器,多线程服务器,select,poll,epoll 等多路 IO 工具都可以,甚至如果读者觉得 libevent 非常熟练的话,也可以使用 libevent 进行开发.

基于 epoll 的 web 服务器

由于我们知道 epoll 在大量并发少量活跃的情况下效率很高,所以本文以 epoll 为例,介绍 epoll 开发的主体流程:



对于我们来说,上述的框架基本没问题,除了处理客户端请求部分,我们可以考虑封装成一个函数,思考:函数参数如何设计?
处理客户端请求流程:



思考题:

1. 由于每个响应消息都是分为四部分,可以考虑封装为函数,封装几个函数更方便?都分别对应什么功能?
2. 目录请求相对复杂,需要遍历目录内的内容,也就是读内容,思考如何做?读到内容形成正文发送的时候,对应的文件类型应该是什么?

epoll_web_server.zip

参考代码: