# 第一章、网络基础

## 一、网络基础

### 1、协议的概念

#### 1.1、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，接收方接收到数据完成后应答OK表示文件接收成功。

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，不断地增加、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。

#### 1.2、典型协议

传输层：常见协议有TCP/UDP协议。

应用层：常见协议有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超文本传输协议是互联网上应用最广泛的一种网络协议。

FTP文件传输协议。

ICMP协议是Internet控制报文协议，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，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协议是面向地址解析协议，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。

### 2、网络应用程序设计模式

#### 2.1、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。

#### 2.2、B/S模式

浏览器(browser)/服务器(server)模式。只需在一端部署服务器，而另一端使用每台PC都默认配置的浏览器即可完成数据的传输。

#### 2.3、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协议的修改裁剪版。

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由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，从用户的角度出发，需要将客户端安装在用户主机上，对用户主机的**安全性构成威胁**。这也是很多用户不愿意使用C/S模式应用程序的重要原因。

B/S模式相对于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**工作开发量小**，只需开发服务器端即可。另外由于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。根据实际需要选择应用程序设计模式。

### 3、分层模型

#### 3.1、OSI七层模型

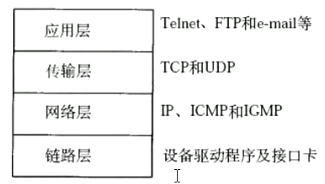


OSI模型

1. **物理层**：主要定义物理标准设备，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和模数转换）。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得用户数大大增加，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协议特性恰恰相反，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得）。主要是将从下层接收得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收会话请求（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用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。

#### 3.2、TCP/IP四层模型

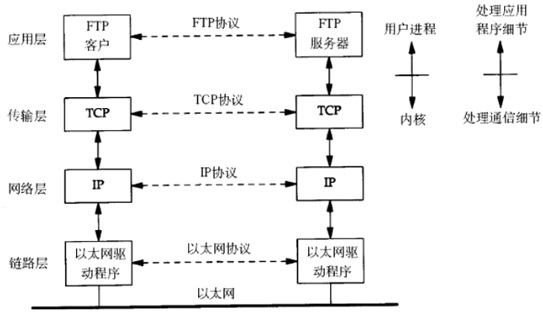
TCP/IP网络协议栈分为应用层、传输层、网络层、网络接口层。



一般在应用开发过程中，讨论最多的是TCP/IP模型。

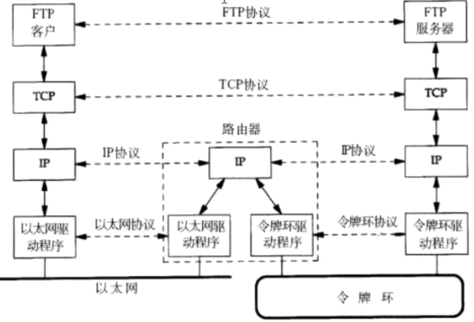
### 4、通信过程

两台计算机通过TCP/IP协议通讯的过程。



TCP/IP通讯过程

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。



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）、数据差错检验等工作。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10M以太网和百M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地址来表示，Internet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。

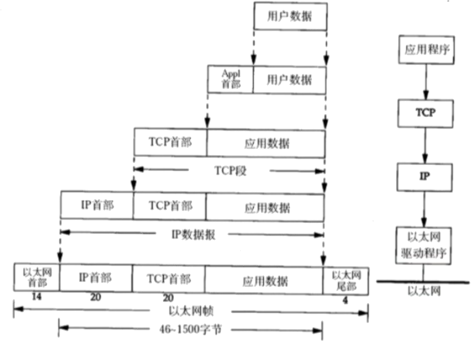
网络层负责点到点的传输（这里的“点”指主机或路由器），而传输层负责端到端的传输（这里的“端”指源主机和目的主机）。传输层可选择TCP或UDP协议。

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

### 5、协议格式

#### 5.1、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，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，每层协议都要加上一个数据首部，成为封装。

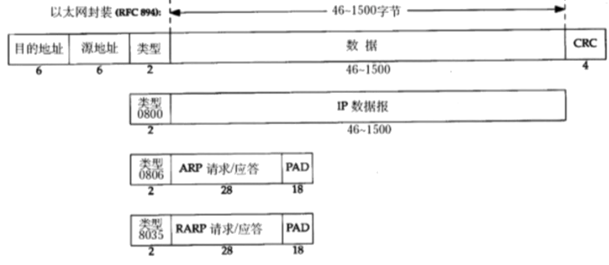


TCP/IP数据包封装

不同的协议层对数据包有不同的称谓，在传输层叫**段**，在网络层叫**数据报**，在链路层叫**帧**。数据封装成帧后发送到传输介质上，达到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。

#### 5.2、以太网帧格式

以太网帧格式如下：



以太网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，是在网卡出场时固话的。可在shell中使用ifconfig命令查看。协议字段有3种，分别是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，数据包的长度大于拨号链路的MTU，则需要对数据包进行分片。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协议：根据IP地址获取MAC地址**

**以太网帧协议：根据MAC地址，完成数据包传输**

#### 5.3、ARP数据报格式

在网络通讯中，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地址与本机不符，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。

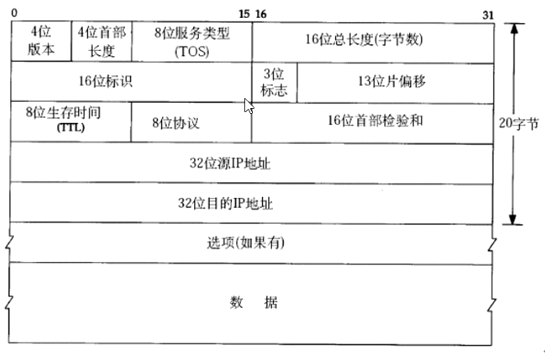
每台主机都维护一个ARP缓存表，可以用arp -a命令查看。缓存表中的表项有过期时间（一般为20分钟），如果20分钟内没有再次使用某个表项，则该表项失效，下次还要发ARP请求来获得目地主机的地址。想一想，为什么表项要有过期时间而不是一直有效？

ARP数据报的格式如下：



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。

#### 5.4、IP段格式



IP数据报的首部长度和数据长度都是可变的，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4×5 = 20字节，也就是不带任何选项的IP首部，4位能表示的最大值为15，也就是首部长度最大为60字节。

8位TOS字段有3个位来指定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。

16位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。

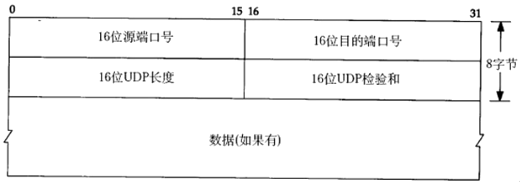
TTL（Time to live）是这样用的：源主机为数据包设定一个生存时间，比如64，每过一个路由器就把该值减1，如果减到0就表示路由已经太长了仍然找不到目的主机的网络，就丢弃该包，因此这个生存时间的单位不是秒，而是跳（hop）。

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地址长度为32位4字节。（IP地址是点分十进制，实际上是一个string）

#### 5.5、UDP数据报格式



UDP数据段

**IP地址**：可以在网络环境，唯一标识一台主机

**端口号**：可以在网络的一台主机上，唯一标识一个进程

**IP地址 + 端口号**：可以在网络环境中，唯一标识一个进程

端口号：16位，上限为

UDP协议不面向连接，也不保证传输的可靠性，例如：

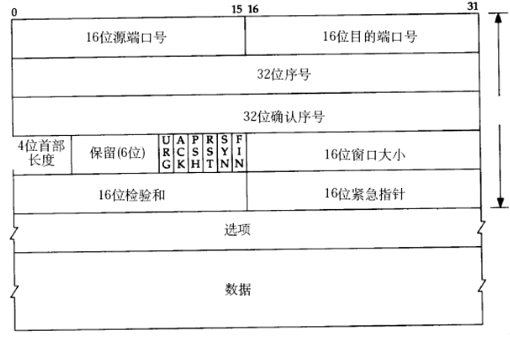
发送端的UDP协议层只管把应用层传来的数据封装成段交给IP协议层就算完成任务了，如果因为网络故障该段无法发送到对方，UDP协议层也不会给应用层返回任何错误信息。

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。

#### 5.6、TCP数据报格式

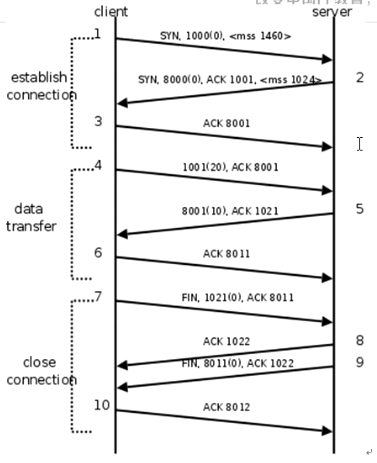


与UDP协议一样有源端口号和目的端口号，通讯的双方由IP地址和端口号标识。32位序号、32位确认序号、窗口大小稍后解释。4位首部长度和IP协议头类似，标识TCP协议头的长度，以4字节为单位，因此TCP协议头最长可以是4×15 = 60字节，如果没有选项字段，TCP协议头最短为20字节。URG、ACK、PSH、RST、SYN、FIN是六个控制位，稍后解释。16位检验和将TCP协议头和数据都计算在内。

### 6、TCP协议

#### 6.1、TCP通信时序

下图是一次通讯的时序图。TCP连接建立断开。包含大家熟知的三次握手和四次挥手。



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),ACK,1001，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（最大报文长度）选项值为1024。

**建立连接（三次握手）的过程：**

1. **客户端发送一个带有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。

1. **服务器回应客户端，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信。**

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，同时也给客户端发送一个连接请求，同时声明最大尺寸为1024。

1. **客户端必须再次回应服务器端一个ACK报文，这是报文段3。**

客户发出段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送了连接请求，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”。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。

**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送到了对方，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的数据包重发。

**关闭连接（四次挥手）的过程：**

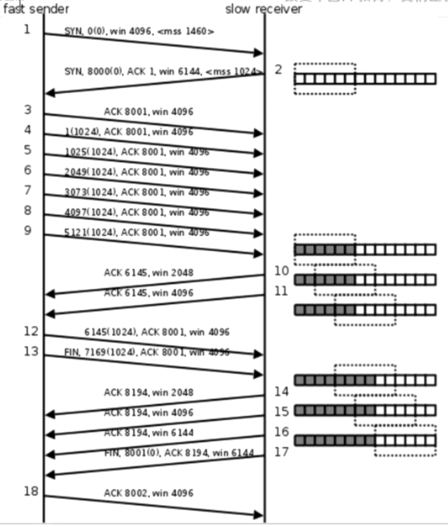
由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接，收到一个FIN就意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭：

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，向客户端关闭发送连接请求。
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次握手，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。

#### 6.2、滑动窗口（TCP流量控制）

如果发送端发送的速度比较快，接收端接收到的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”机制解决这一问题。



1. 发送端发起连接，声明数据最大尺寸是1460，初始序号是0，窗口大小为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1K，初始序号是8000，窗口大小是6K。发送端应答，三次握手结束。
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，接收缓冲区有了4K空闲，接收端发出段11，重新声明窗口大小为4K。
5. 接收端发出段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**。

#### 6.3、TCP状态转换

这个图N多人都知道，它派出和定位网络或系统故障时大有帮助，但是怎样laolaode 将这张图刻在脑海中呢？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中。下面对这张图的11种状态详细解析一下。



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SYN\_SENT状态，并等待服务器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**：表示已经建立连接

**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状态一般比较难见到，而FIN\_WAIT\_2状态可用netstat看到。

**FIN\_WAIT\_2**：**主动关闭连接的一方，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，而无需经过FIN\_WAIT\_2状态。

**CLOSING**：这种状态比较特殊，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，系统会回收一个ACK报文交给对方，此时则进入到CLOSE\_WAIT状态。接下来呢，查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。

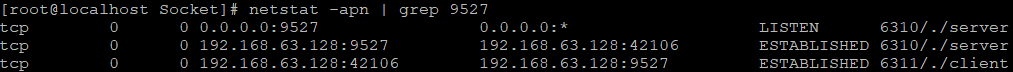
**LAST\_ACK**：该状态是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，即可以进入到CLOSED可用状态。

**三次握手建立连接**



1. 主动发起连接请求端：CLOSED —> 发送SYN —> SYN\_SENT —> 接收SYN、ACK —> SYN\_SENT —> 发送ACK —> ESTABLISHED（数据传输状态）

**示例：**启动server和client后，一个用于通信的服务端进程和客户端进程是ESTABLISHED状态，一个用于监听的服务端进程是LISTEN状态



1. 被动接收连接请求端：CLOSED —> LISTEN —> 接收SYN —> LISTEN —> 发送ACK、SYN —> SYN\_RCVD —> 接收ACK —> ESTABLISHED（数据传输状态）

**四次挥手关闭连接**



1. 主动关闭连接请求端：ESTABLISHED（数据传输状态）—> 发送FIN —> FIN\_WAIT\_1 —> 接收ACK —> FIN\_WAIT\_2（半关闭）—> 接收对端发送FIN —> FIN\_WAIT\_2（半关闭）—> 回发ACK —> **TIME\_WAIT（只有主动关闭连接方才有该状态）** —> 等2MSL时长 —> CLOSED

**示例：**如果先结束server，然后再想立即重启server，就会重启失败



此时查看进程，处于TIME\_WAIT状态



等待一段时间（2MSL时长）后再启动server就会成功，可以使用端口复用来解决这个问题

1. 被动关闭连接请求端：ESTABLISHED（数据通信状态）—> 接收FIN —> ESTABLISHED —> 发送ACK —> CLOSE\_WAIT（说明对端【主动关闭连接端】处于半关闭状态）—> 发送FIN —> LAST\_ACK —> 接收ACK —> CLOSED

**【注意】：**2MSL时长是为了保证最后一个ACK能成功被对端接收。（等待期间，对端没收到我发的ACK，对端会再次发送FIN请求）

#### 6.4、半关闭

当TCP连接中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，A方处在半连接状态，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操作选择一下几种方式：

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收缓冲区的数据将被丢弃

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

再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SHUT\_RD，

然后是SHUT\_WR。

使用close中止一个连接，但它只是减少描述符的引用次数，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，也就是所有进程都调用了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)将不会影响到其他进程。

#### 6.5、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没有完全断开，而我们重新监听的是listenfd，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，表示是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));*

## 二、Socket编程

### 1、套接字概念

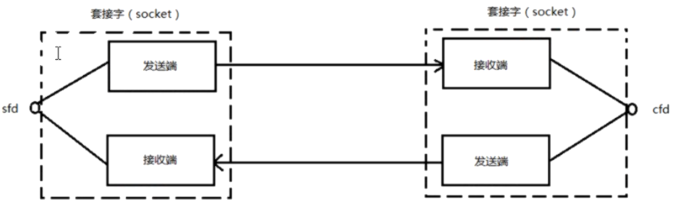
Socket本身有“插座”的意思，在Linux环境下，用于表示进程间通信的特殊文件类型。本质为内核借助缓冲区形成的**伪文件**。

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或者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：



**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。一个文件描述符指向一个套接字，该套接字内部由内核借助发送缓冲区和接收缓冲区实现。

### 2、预备知识

#### 2.1、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，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。同样的，接收主机如果是小端字节序的，接收到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) 本地转网络无符号32位整数（IP）

uint16\_t htons(uint16\_t hostshort) 本地转网络无符号16位整数（port）

uint32\_t ntohl(uint32\_t netlong) 网络转本地无符号32位整数（IP）

uint16\_t ntohs(uint16\_t netshort) 网络转本地无符号16位整数（port）

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，将参数原封不动返回。

#### 2.2、IP地址转换函数

本地字节序（string IP）——> 网络字节序（二进制）



参数：

af：协议类型，IPv4（AF\_INET），IPv6（AF\_INET6）

src：传入参数，本地字节序的IP地址（点分十进制）

dst：传出参数，转换后的网络字节序的IP地址

返回值：

成功：1

异常：0，说明src指向的不是一个有效的IP地址

失败：-1，设置errno

网络字节序——> 本地字节序（string IP）



参数：

af：协议类型，IPv4（AF\_INET），IPv6（AF\_INET6）

src：传入参数，网络字节序IP地址

dst：传出参数，本地字节序的IP地址

size：dst的大小

返回值：

成功：dst

失败：NULL，设置errno

支持IPv4和IPv6

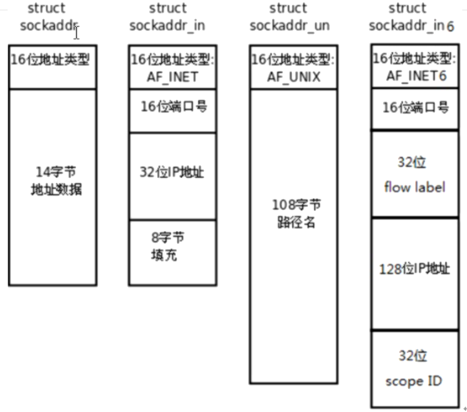
可重入函数

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。

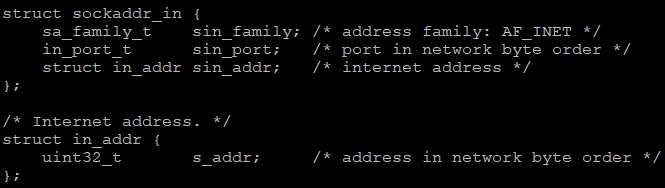
因此函数接口是void\* addrptr

#### 2.3、sockaddr数据结构

struct sockaddr很多网络编程函数诞生早于IPv4协议，那时候使用的是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由地址族决定，然后函数内部再强制转化为所需的地址类型。



可使用man 7 ip查看sockaddr\_in结构体



示例：

struct sockaddr\_in addr;

addr.sin\_family = AF\_INET;

addr.sin\_port = htons(9527);

/\*

int dst;

inet\_pton(AF\_INET, “192.168.63.128”, (void\*)dst);

addr.sin\_addr.s\_addr = dst;

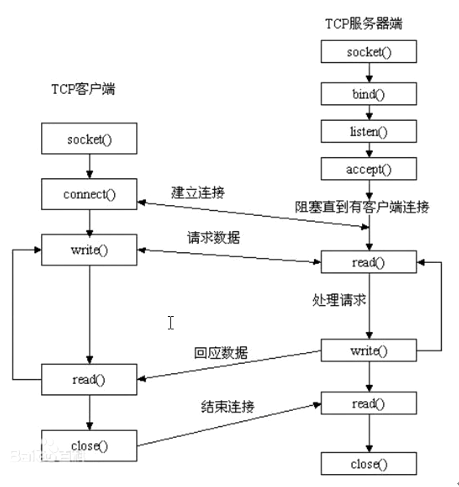
\*/

addr.sin\_addr.s\_addr = htonl(INADDR\_ANY); //取出系统中有效的任意IP地址，二进制类型。

bind(fd, (struct sockaddr \*)&addr, size);

### 3、网络套接字函数

#### 3.1、socket模型创建流程图



socket API

有三个套接字，其中一对是客户端和服务端配对的，剩下一个独立的用于监听。

socket函数：创建套接字

bind函数：绑定IP和port

listen函数：设置同时监听上限

accept函数：阻塞监听客户端连接

connect函数：建立连接

#### 3.2、socket函数

创建一个套接字，头文件<sys/socket.h>



参数：

domain：协议类型，AF\_INET、AF\_INET6、AF\_UNIX

type：协议，SOCK\_STREAM、SOCK\_DGRAM

protocol：0

返回值：

成功：新套接字所对应的文件描述符fd

失败：-1，设置errno

#### 3.3、bind函数

给socket绑定一个IP + 端口号，头文件<arpa/inet.h>



参数：

sockfd：socket函数返回值

addr：(struct sockaddr \*)&addr

addrlen：sizeof(addr)，地址结构大小

返回值：

成功：0

失败：-1，设置errno

#### 3.4、listen函数

设置同时与服务器建立连接的上限数。（同时进行3次握手的客户端数量）



参数：

sockfd：socket函数返回值

backlog：上限数值，最大值为128

返回值：

成功：0

失败：-1，设置errno

#### 3.5、accept函数

阻塞等待客户端建立连接，成功的话返回一个与客户端成功连接的socket文件描述符



参数：

sockfd：socket函数返回值，服务器的socket

addr：传出参数。成功与服务器建立连接的那个客户端的地址结构

addrlen：传入传出参数，地址结构大小。入：addr的大小；出：客户端addr实际大小

返回值：

成功：能与服务器进行数据通信的socket对应的文件描述符

失败：-1，设置errno

#### 3.6、connect函数

使用现有的socket与服务器建立连接



参数：

sockfd：socket函数返回值，客户端的socket

addr：传入参数。服务器的地址结构

addrlen：服务器地址结构大小。

返回值：

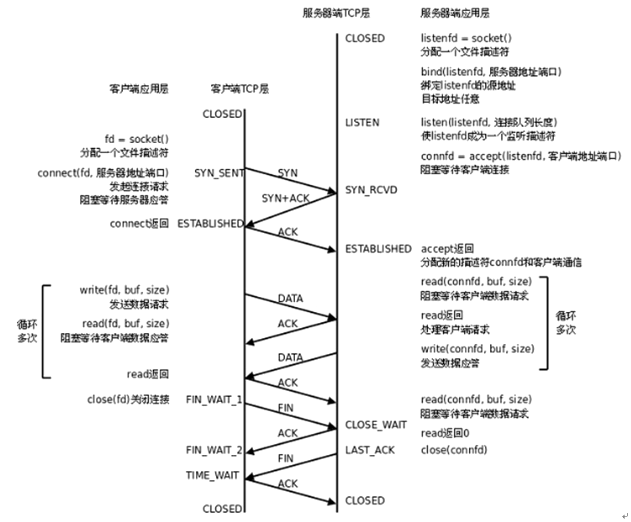
成功：0

失败：-1，设置errno

【注意】：如果不使用bind绑定客户端地质结构，采用“隐式绑定”。

### 4、C/S模型 – TCP通信

建立小写转大写的C/S模型



#### 4.1、server

1. socket() 创建socket

2. bind() 绑定服务器地址结构

3. listen() 设置监听上限

4. accept() 阻塞监听客户端连接

5. read() 读socket获取客户端数据

6. 小—>大写 toupper()

7. write(fd)

8. close()

#### 4.2、client

1. socket() 创建socket

2. connect() 与服务器建立连接

3. write() 写数据到socket

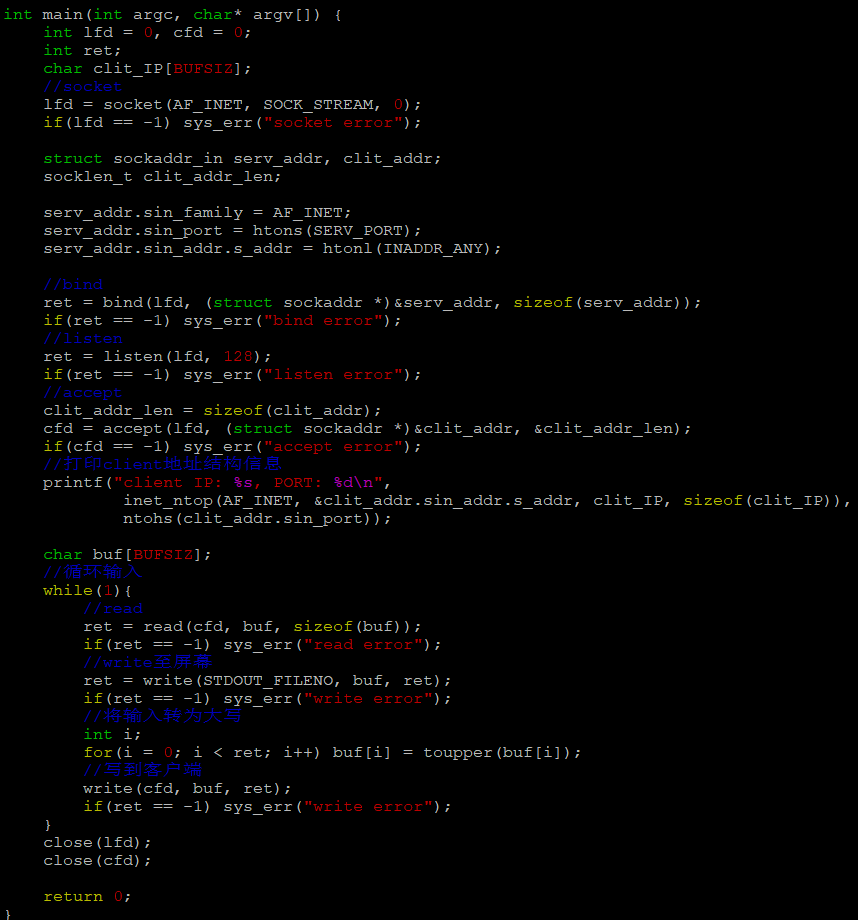
4. read() 读转换后的数据

5. 显示读取结果

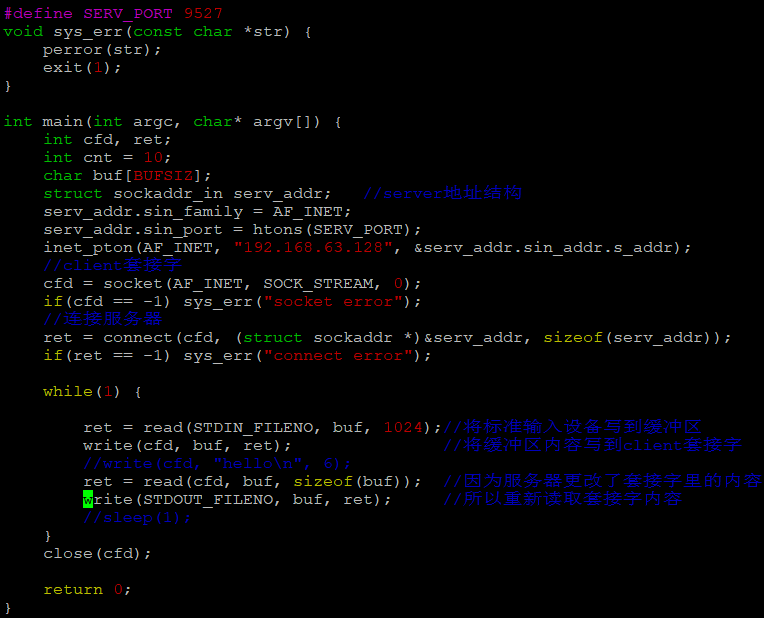
6. close()

#### 4.3、代码实现：

##### server

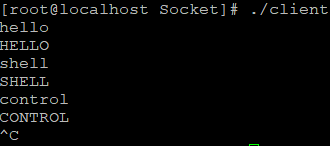


##### client

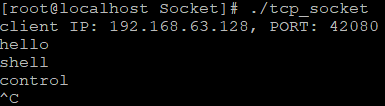


##### 实验结果

**client：**



**server：**



可以看出，输入小写单词，客户端输出的是大写单词

### 5、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，另一方面也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。

**封装目的：**

在server.c编程过程中突出逻辑，将错误处理与逻辑分开，可以直接跳转man手册

【wrap.c】

存放网络通信相关常用自定义函数

命名方式：系统调用函数首字母大写，方便查看man手册，如Listen()、Accept()

函数功能：调用系统调用函数，处理出错场景

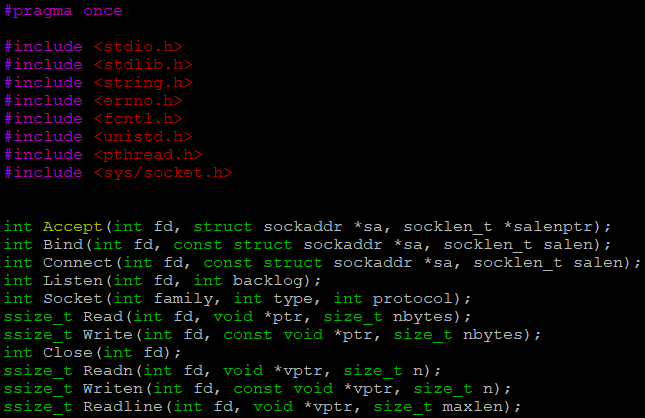
在server.c和client.c中调用自定义函数

联合编译server.c和wrap.c生成server、联合编译client.c和wrap.c生成client

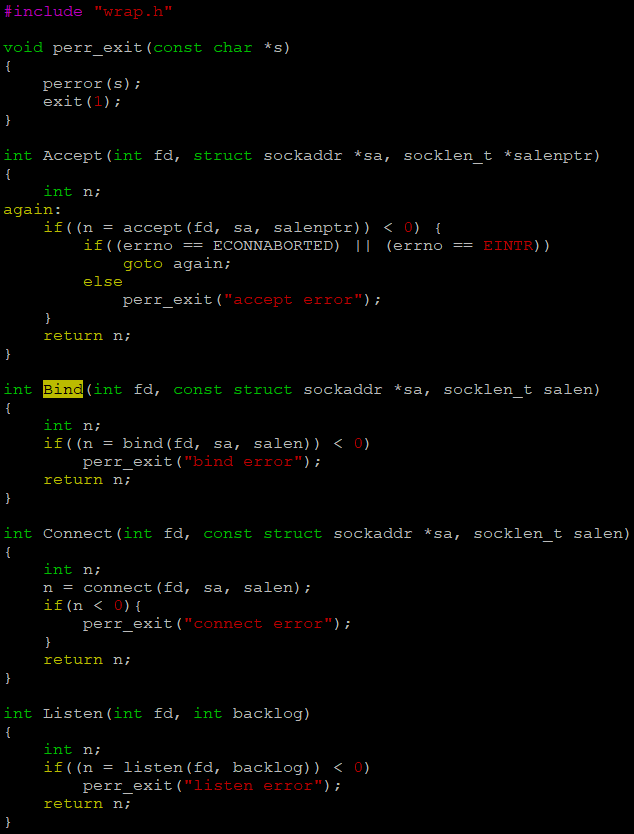
【wrap.h】

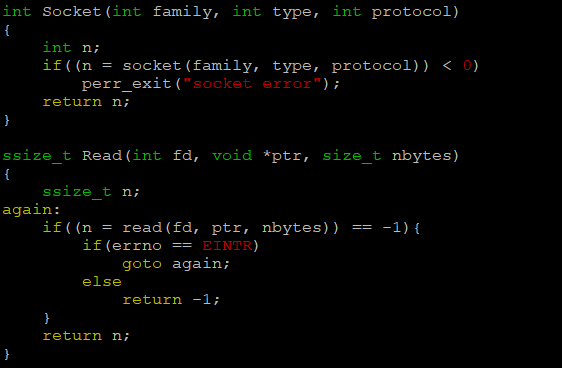
存放网络通信相关常用自定义函数原型（声明）

#### 5.1、wrap.h

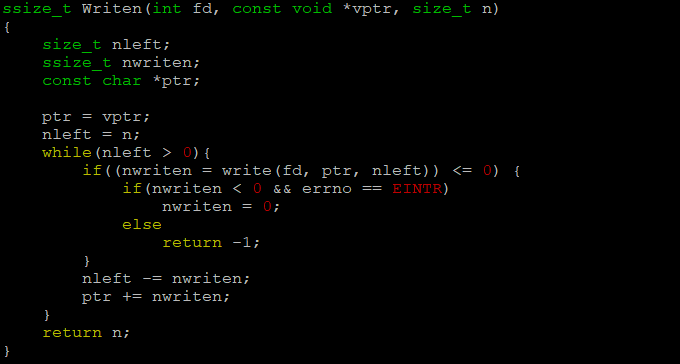


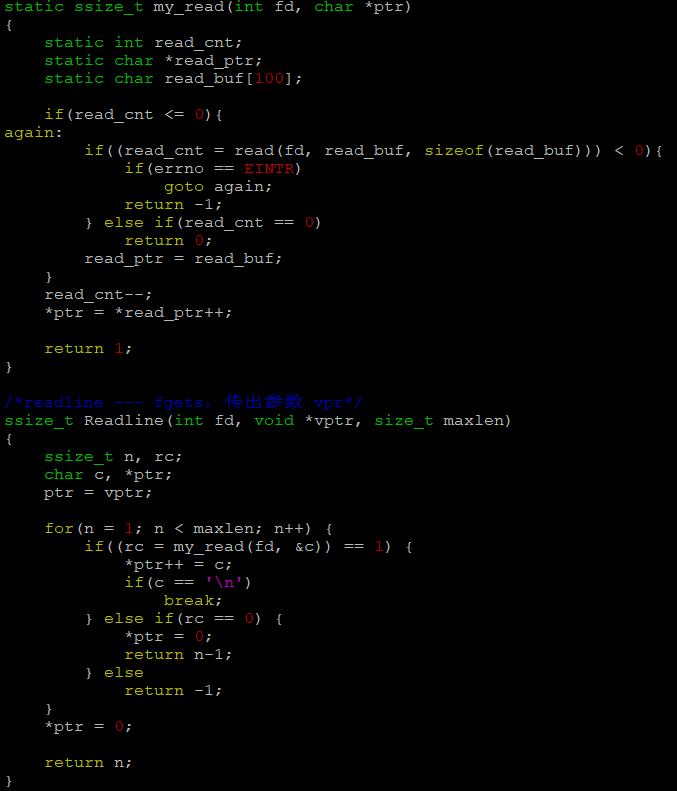
#### 5.2、wrap.c



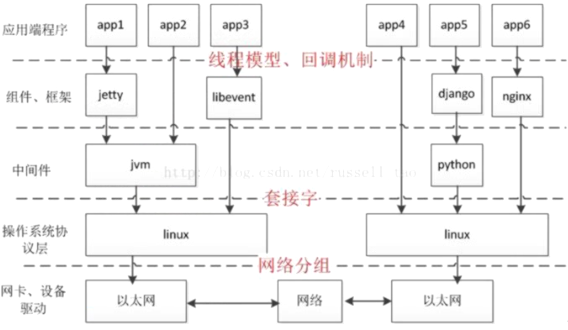








## 三、高并发服务器



### 1、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器性能（任务调度）

#### 1.1、思路：

1.Socket(); 创建监听套接字lfd

2. Bind(); 绑定地址结构

3. Listen(); 设置监听上限

4. Accept()

while (1) {

cfd = Accep(); 接收客户端连接请求

pid = fork();

if (pid == 0) { 子进程read(cfd) --- 小 ——> 大 --- write(cfd)

//

} else if (pid > 0) { 继续监听，并回收子进程

//

}

}

5. 子进程：

close(lfd); 关闭用于建立连接的套接字lfd

read(cfd);

小 ——> 大

write(cfd);

父进程：

close(cfd); 关闭用于与客户端通信的套接字cfd

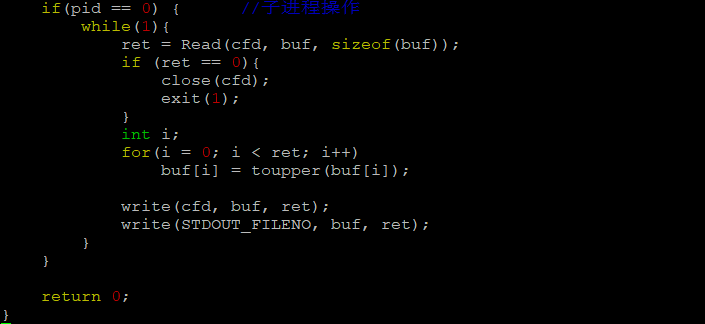
注册信号捕捉函数：SIGCHLD

在回调函数中，完成子进程回收

while (waitpid())

#### 1.2、代码





### 2、多线程并发服务器

#### 2.1、思路

1.*Socket();* 创建监听套接字lfd

2. *Bind();*  绑定地址结构

3. *Listen();* 设置监听上限

4. *Accept()*

*while (1) {*

*cfd = Accep();* 接收客户端连接请求

*pthread\_create(&tid, NULL, tfn, NULL);*

*pthread\_detach(tid);*

*}*

5. 子线程：

*void \*tfn(void \*arg) {*

*close(lfd);* 关闭用于建立连接的套接字*lfd*

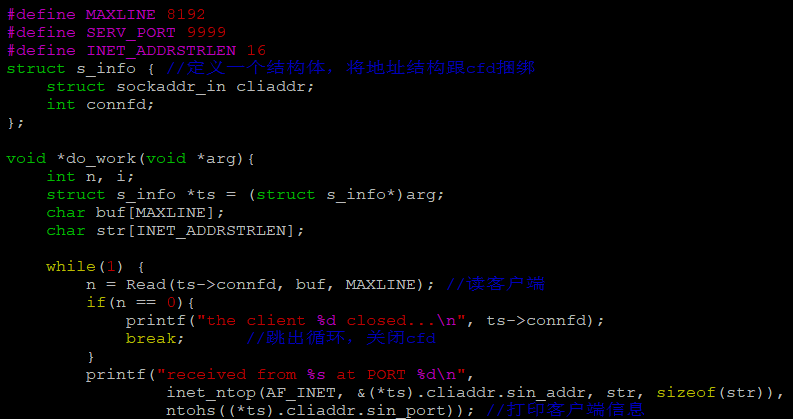
*read(cfd);*

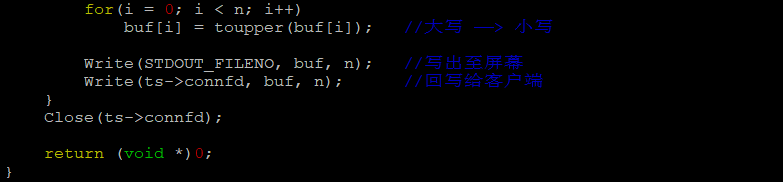
小 ——> 大

*write(cfd);*

*}*

#### 2.2、代码







**【注意】**：如果想将本地文件传到其它外网服务器里，使用scp -r 命令

scp -r ./test/ ssz@101.200.170.171:/home/ssz/socket\_server

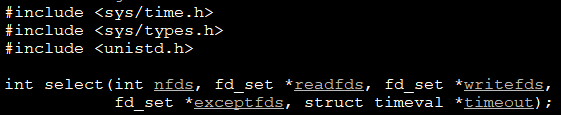
### 3、多路I/O转接服务器

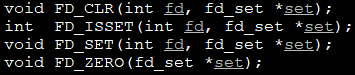
多路I/O转接服务器也叫做多任务I/O服务器。该类服务器实现的主旨思想是，不再由应用程序自己监听客户端连接，取而代之由内核替应用程序监视文件。

#### 3.1、select

1. select能监听的文件描述符受限于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以下客户端使用select是很合适的，但如果连接客户端过多，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。





FD\_CLR：将文件描述符fd从监听集合set中移除

FD\_SET：将待监听的文件描述符fd，添加到集合set中

FD\_ISSET：判断文件描述符fd是否在集合set里

FD\_ZERO：清空一个文件描述符监听集合set

**参数：**

**nfds**：监控的文件描述符集里最大文件描述符+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**：监控异常发生的文件描述符集合，如带外数据到达异常，传入传出参数

【注意】：上面3个传入传出参数是这样的，比如readfds监控文件描述符{3，4，5，6}，然后有5，6写数据到服务器，那么返回集合就是{5，6}，是以**位图**的形式表现的。

**timeout**：定时阻塞监控时间，3种情况

1. NULL，阻塞监听
2. 设置timeval，设置监听超时时长
3. 设置timeval里时间均为0，检查描述字后立即返回，非阻塞监听，轮询

*struct timeval {*

*long tv\_sec; /\*seconds\*/*

*long tv\_usec; /\*microseconds\*/*

*}*

**返回值：**

> 0：所有监听集合（3个）中，满足对应事件的总数

0： 没有满足监听条件的文件描述符

-1： 失败，设置errno

##### 思路分析

*int maxfd = 0;*

*lfd = socket();* 创建套接字

*maxfd = lfd;* 最大文件描述符

*bind();* 绑定地址结构

*listen();* 设置监听上限

*fd\_set rset, allset;* 创建r监听集合

*FD\_ZERO(&allset);* 将*r*监听集合清空

*FD\_SET(lfd, &allset);* 将*lfd*添加到读集合中

*while(1) {*

*rset = allset;* 保存监听集合

*ret = select(lfd+1, &rset, NULL, NULL, NULL);* 监听文件描述符集合对应事件

*if (ret > 0) {*

*if (FD\_ISSET(lfd, &rset)) { 1*在。*0*不在

*cfd = Accept();* 建立连接，返回用于通信的文件描述符

*maxfd = cfd;*

*FD\_SET(cfd, &allset);* 添加到监听通信描述符集合中

*}*

*for (i = lfd + 1; i <= maxfd; i++) {*

*FD\_ISSET(i, &rset);* 有*read、write*事件

*read();*

小 —> 大

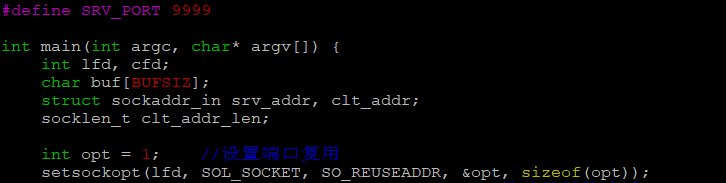
*write();*

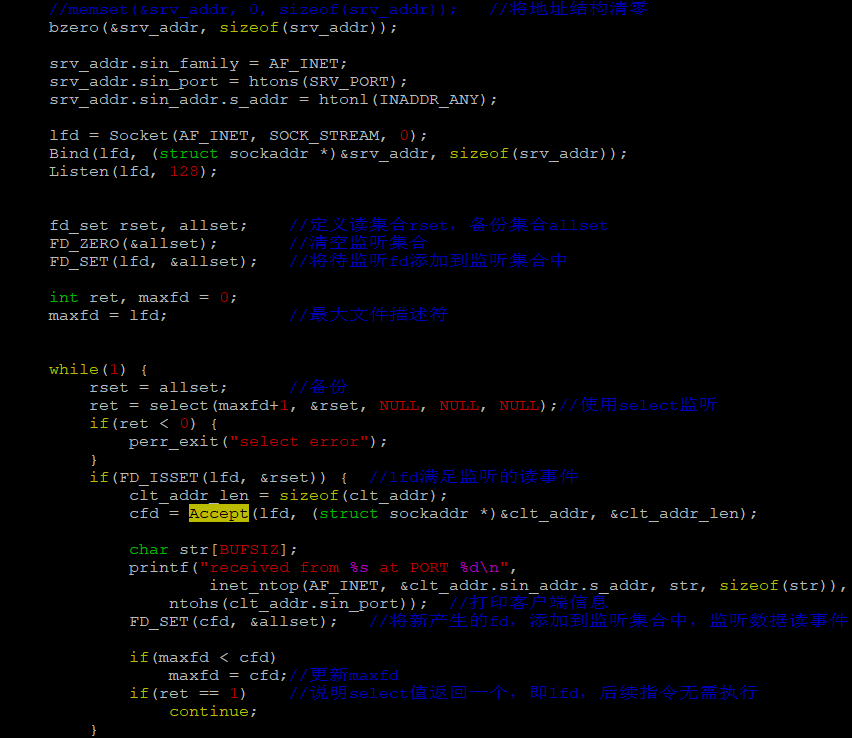
*}*

*}*

*}*

##### 代码







##### select优缺点：

**缺点：**

1. 监听上限受文件描述符限制。最大1024。
2. 检测满足条件的fd，自己添加业务逻辑提高小，提高了编码难度。

**优点：**

跨平台。win、linux、macOS、Unix、类Unix、mips

##### 添加一个自定义数组提高效率

上述代码中，当有客户端发出写事件请求时，因为select返回的只是发出请求的客户端数量，并没有标注是哪个客户端发出的，所以需要进行检测。前面用的方法是对所有的创建的fd进行筛选，找出满足读事件的fd，这样效率不高。

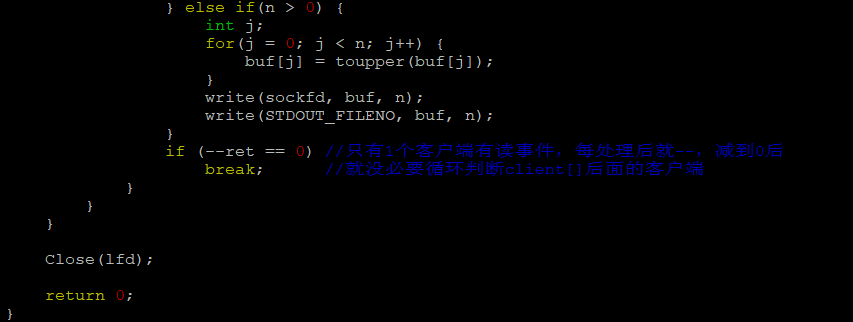
可以使用一个客户端数组client[1024]，元素初始值均为-1，专门用来存连接在服务器上的客户端对应的fd，也就是说，每当lfd发现有客户端进行连接时，就会在client[]中找出未使用的最小下标的区域，用来存放该客户端的cfd；当客户端关闭连接后，不仅关闭该客户端对应的fd，还要将该客户端对应的client数组的元素重新置-1。

当有读事件时，不需要遍历所有的fd了，只需要遍历client数组即可（通过一个maxi来记录连接的最新的客户端，这样方便遍历条件，无需每次都把整个client遍历完），因为凡是连接的客户端，client[i]均不是-1，结合select函数的返回值，可提前结束循环。每当处理完一个事件，就让select的返回值--，当select返回值减到0之后，就无需再遍历完整个client了。

##### 代码

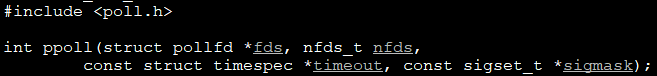


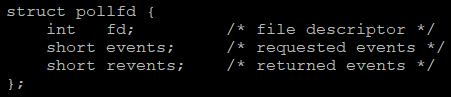




#### 3.2、poll

跟select使用方法基本一致





fds 监控数组（传入传出）

fd：待监听的文件描述符

events：待监听的文件描述符对应的监听事件

取值：POLLIN、POLLOUT、POLLERR

revents：监控事件中满足条件返回的事件，

传入时，给0。如果满足对应事件的话，返回 非0

传出非0值：POLLIN、POLLOUT、POLLERR

**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：阻塞等待，#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

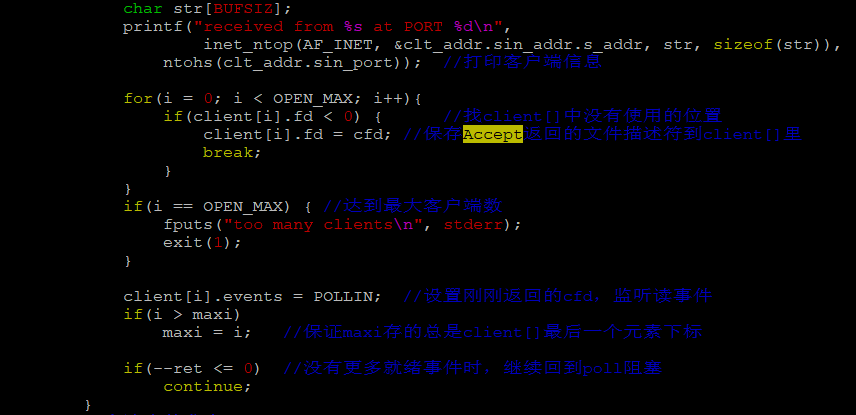
如果不再监控某个文件描述符时，可以把pollfd中，fd设置为-1，poll不再监控此pollfd，下次返回时，把revents设置为0。

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。

##### 代码







##### poll优缺点

**优点：**

自带数组结构。可以将监听事件集合和返回事件集合分离。

拓展监听上限，超出1024限制

**缺点：**

不能跨平台。只能在Linux上使用。

无法直接定位满足监听事件的文件描述符，编码难度较大

#### 3.3、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，它无需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。

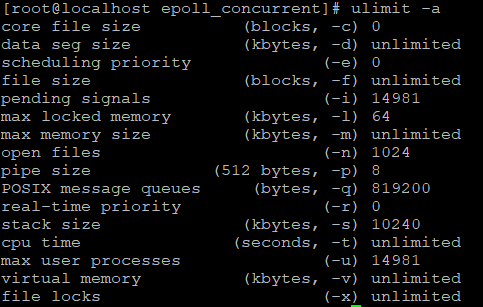
epoll除了提供select/poll那种IO事件的电平触发外，还提供边沿触发，这就使得用户空间程序有可能缓存IO状态，减少epoll\_wait/epoll\_pwait的调用，提高应用程序效率。

可以使用cat命令查看一个进程可以打开的socket描述符上限，受硬件影响。

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



可以使用*ulimit -a*命令查看当前用户下的进程，默认打开文件描述符个数。缺省为1024。（里面的open files）



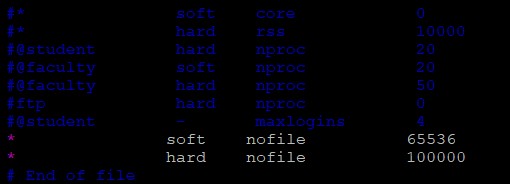
**如有需要，可以通过修改配置文件的方式修改该上限值。**

*sudo vim /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* 设置默认值，可以直接借助命令修改【注销用户，使其生效】

*\* hard nofile 100000* 命令修改上限



可以使用*ulimit -n xxx*来修改open file值，不超过设置的hard nofile

epoll本质是一棵**红黑树**

头文件<sys/epoll.h>

##### epoll\_create函数

创建一棵监听红黑树



**参数：**

**size**：创建的红黑树的监听节点数量。（仅供内核参考）

**返回值：**

成功：指向新创建的红黑树的根节点的fd

失败：-1，设置errno

##### epoll\_ctl函数

操作一棵监听红黑树



**参数：**

**epfd**：epoll\_create函数的返回值。

**op**：对该监听红黑树所做的操作。

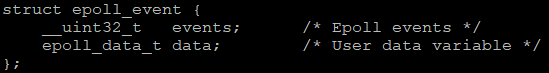
EPOLL\_CTL\_ADD 添加fd到监听红黑树

EPOLL\_CTL\_MOD 修改fd在监听红黑树上的监听事件

EPOLL\_CTL\_DEL 将一个fd从监听红黑树上摘下（取消监听）

**fd**：待监听的fd

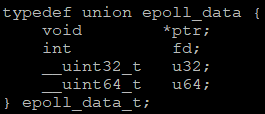
**event**：本质struct epoll\_event结构体 **地址**



events：

EPOLLIN、EPOLLOUT、EPOLLERR

data：联合体



void \*ptr：

fd：对应监听事件的fd

u32：不用

u64：不用

**返回值：**

成功：0

失败：-1，设置errno

events可选值：

**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

EPOLLPRT：表示对应的文件描述符有紧急的数据可读（这里是有带外数据到来）

**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，这是相当于水平触发（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

##### epoll\_wait函数

阻塞监听



**参数：**

**epfd**：epoll\_create函数的返回值。

**events**：传出参数，【数组】，满足监听条件的那些fd结构体

**maxevents**：数组元素的总个数，直接写设定的数组大小，不用sizeof

**timeout**：

-1：阻塞等待

0：不阻塞

>0：等待指定毫秒数

**返回值：**

>0：满足监听的总个数，可以用作循环上限。

0：没有fd满足监听事件

-1：失败，设置errno

##### 思路分析

*lfd = socket();* 创建套接字

*bind();* 绑定地址结构

*listen();* 设置监听上限

*int epfd = epoll\_create(1024); epfd：*监听红黑树的树根

*struct epoll\_event tep, ep[1024]; tep*是*epoll\_ctl*参数*，ep[]是epoll\_wait*参数

*tep.events = EPOLLIN;*

*tep.data.fd = lfd;*

*epoll\_ctl(epfd, EPOLL\_CTL\_ADD, lfd, &tep)* 将*lfd*添加到监听红黑树上

*while(1) {*

*ret = epoll\_wait(epfd, ep, 1024, -1);* 实施监听

*for(i = 0; i < ret; i++) {*

*if(ep[i].data.fd == lfd) { lfd*满足读事件，有客户端发起连接请求

*cfd = Accept();*

*tep.events = EPOLLIN;* 初始化*cfd*监听属性

*tep.data.fd = cfd;*

*epoll\_ctl(epfd, EPOLL\_CTL\_ADD, cfd, &tep);*

*} else { cfd*们满足读事件，有客户端写数据来

*n = read(ep[i].data.fd, buf, sizeof(buf))；*客户端读到*buf*

*if(n == 0){*

*close(ep[i].data.fd);* 关闭*cfd，*从监听树上摘下

*epoll\_ctl(epfd, EPOLL\_CTL\_DEL, ep[i].data.fd, NULL);*

*} else if (n > 0) {*

*小——>大*

*Write(ep[i].data.fd, buf, n); buf*写回客户端

*}*

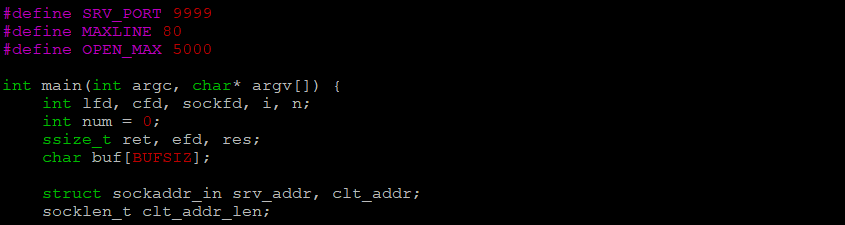
*}*

*}*

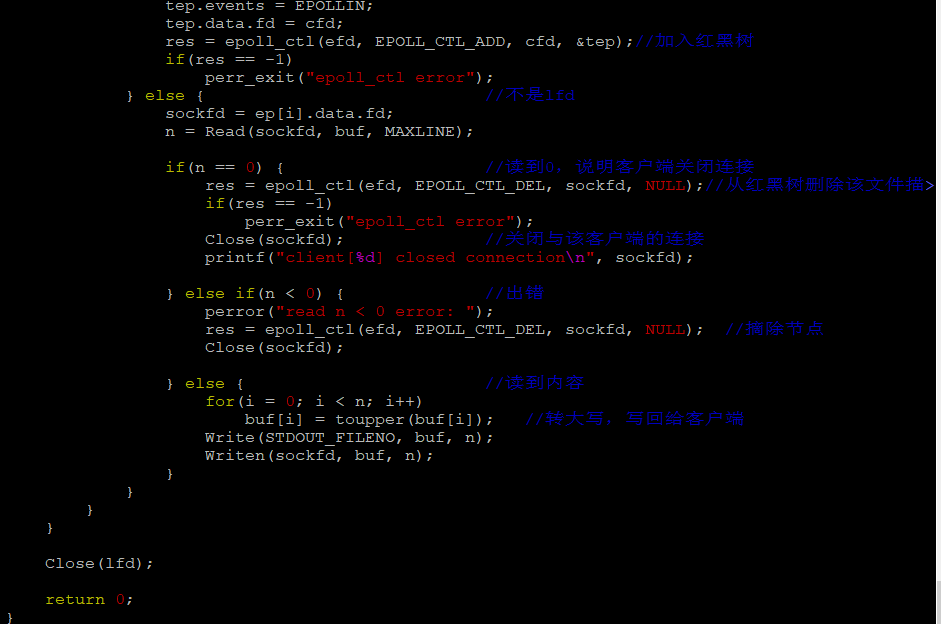
*}*

*Close(lfd);*

##### 代码







##### epoll优缺点

优点：

高效，**突破1024文件描述符上限**。

缺点：

不能跨平台，只支持Linux。

### 4、epoll进阶

#### 4.1、事件模型

EPOLL事件有两种模型：

Edge Triggered(ET) **边沿触发**只有数据到来才触发，不管缓冲区中是否还有数据

struct epoll\_event event;

event.events = EPOLLIN | EPOLLET

Level Triggered(LT) **水平触发**只要有数据都会触发（默认采用模式）

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区中已没有数据了，也就可以认为此次读事件已处理完成。

##### 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。如果你不做任何操作，内核还是会继续通知你的，所以，这种模式编程出错的可能性要低一些。**传统的select/poll都是这种典型的代表。**

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进行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）。

##### 结论

**epoll的ET模式是高效模式，但是只支持非阻塞模式。**

*struct epoll\_event event;*

*event.events = EPOLLIN | EPOLLET;*

*epoll\_ctl(epfd, EPOLL\_CTL\_ADD, cfd, &event);*

*int flg = fcntl(cfd, F\_GETFL);//设置非阻塞*

*flg |= O\_NONBLOCK;*

*fcntl(cfd, F\_SETFL, flg);*

epoll反应堆模型：

epoll ET模式 + 非阻塞轮询 + void\* ptr

**原来**：socket、bind、listen – epoll\_create创建监听红黑树 -- 返回epfd -- epoll\_ctl()向树上添加一个监听fd -- while(1) -- epoll\_wait监听 -- 对应监听fd有事件发生 -- 返回监听满足数组 – lfd满足 -- accep -- cfd满足 -- read -- 小—>大 -- write回去。

**反应堆**：不但要监听cfd的读事件，还要监听cfd的写事件

socket、bind、listen – epoll\_create创建监听红黑树 -- 返回epfd -- epoll\_ctl()向树上添加一个监听fd -- while(1) -- epoll\_wait监听 -- 对应监听fd有事件发生 -- 返回监听满足数组 – lfd满足 -- accep -- cfd满足 -- read -- 小—>大 – cfd从监听红黑树上摘下 – EPOLLOUT – 回调函数 – epoll\_ctl() – EPOLL\_CTL\_ADD重新放到红黑树上监听写事件 – 等待epoll\_wait返回 – 说明cfd可写 -- write回去 – cfd从红黑树上摘下 – EPOLLIN – epoll\_ctl() – EPOLL\_CTL\_ADD重新放到红黑树上监听读事件 – epoll\_wait监听。

eventset函数：

设置回调函数：lfd ——> acceptconn()

cfd ——> recvdata()

eventadd函数：

将一个fd添加到红黑树上，设置监听读事件还是写事件

### 5、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，一种是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输”。

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**，**数据传输速率快**。多用于对实时性要求比较高的通信场合，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。

TCP通信和UDP通信各自的**优缺点**：

TCP：面向连接的，可靠数据包传输。对于不稳定的网络层，采用完全弥补的通信方式。丢包重传。

**优点**：稳定。数据流量稳定、速度稳定、顺序。

**缺点**：传输速度慢，效率低，开销大。

**使用场景**：数据的完整性要求较高，不追求效率。大数据传输、文件传输。

UDP：无连接的，不可靠的数据包传输。对于不稳定的网络层，采取完全不弥补的通信方式。默认还原网络状况。

**优点**：传输速度快，效率高，开销小。

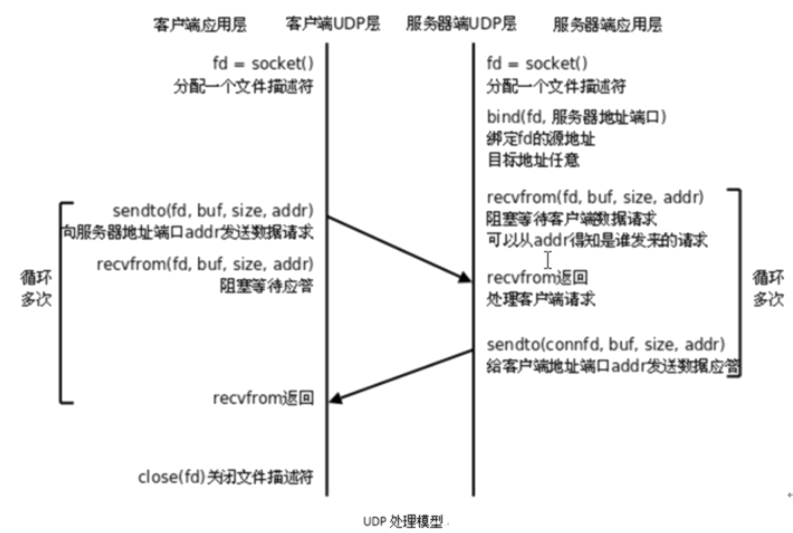
**缺点**：不稳定，数据流量、速度、顺序得不到保证。

**使用场景**：对时效性要求较场合。稳定性其次。游戏、视频会议、视频电话。

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：

#### 5.1、C/S模型-UDP



##### server

1. socket(AF\_INET, SOCK\_DGRAM, 0) 创建socket，SOCK\_DGRAM—报式协议

2. bind() 绑定服务器地址结构

3. listen() 设置监听上限

while (1) {

read(cfd, buf, sizeof) 被替换为 recvfrom() 兼顾accept()功能

小写 ---> 大写

write() 被替换为 sendto() 兼顾connect()功能

}

4. close()

##### client

1. socket(AF\_INET, SOCK\_DGRAM, 0)

2. sendto(“服务器地址结构”, 地址结构大小)

3. recvfrom()

4. 写到屏幕

5. close()

**recvfrom函数和sendto函数：**



参数：sockfd：lfd buf: 缓冲区地址 len：缓冲区大小 flags：0

src\_addr：(struct sockaddr \*)&addr 传出参数，对端地址结构 addrlen：传入传出

返回值：成功：成功接收的数据字节数 失败：-1.设置errno 0：对端关闭



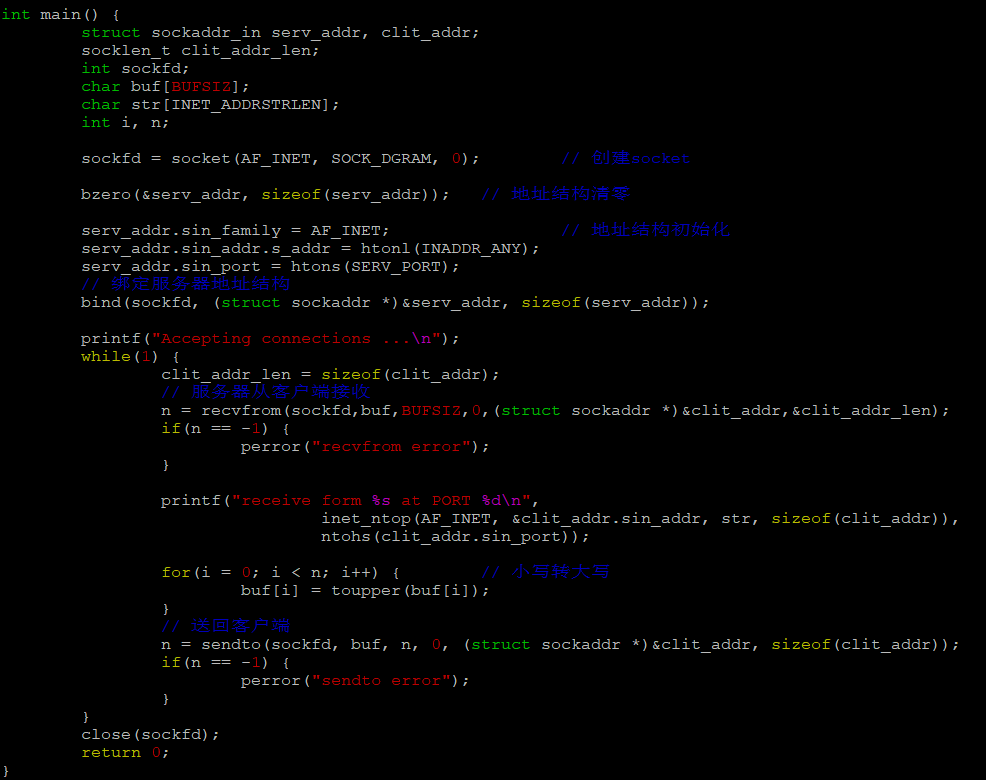
参数：sockfd：套接字 buf：存储数据的缓冲区 len：数据长度 flags：0

dest\_addr：(struct sockaddr \*)&addr 传入参数，对端地址结构 addrlen：地址结构长度

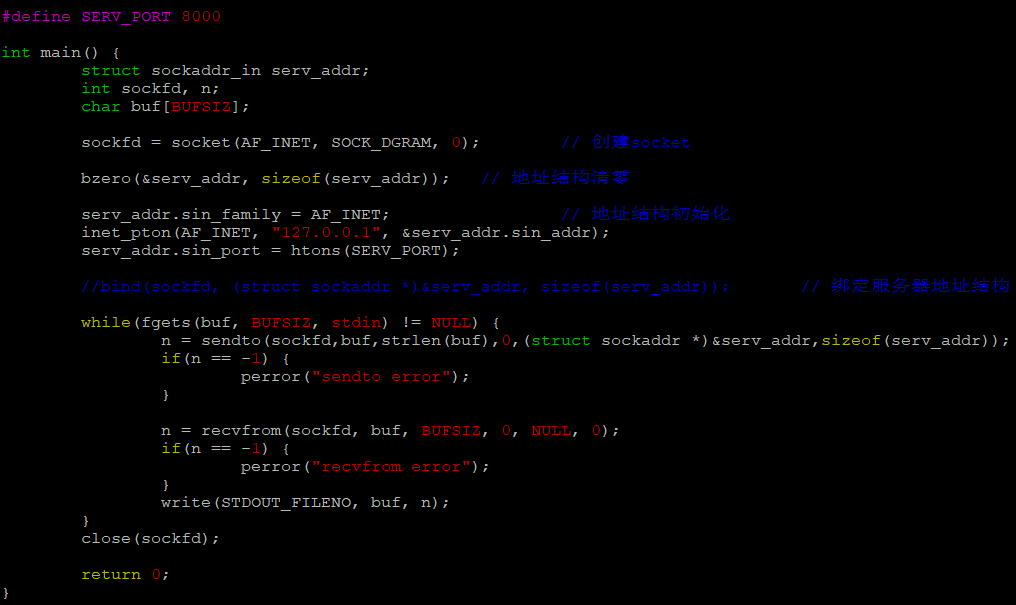
返回值：成功：成功写出的数据字节数 失败：-1.设置errno 0：对端关闭

#### 5.2、代码

##### server代码



##### client代码



### 6、本地套接字

IPC（进程间通信）：pipe、fifo、mmap、信号、本地套接字(domain) ------ C/S模型

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信（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UNIX Domain 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参数仍可指定为0。

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+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()创建，如果调用bind()时该文件已存在，则bind()错误返回。

**对比**网络套接字结构和本地套接字结构：

struct sockaddr\_in {

\_\_kernel\_sa\_family\_t **sin\_family**; // 地址结构类型

\_be16 **sin\_port;** // 端口号

struct in\_addr **sin\_addr**; // IP地址

};

struct sockaddr\_un {  
 \_\_kernel\_sa\_family\_t **sun\_family**; // 地址结构类型

char **sun\_path**[UNIX\_PATH\_MAX]; // 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) & ((typr \*)0)—>MEMBER)

**对比网络编程TCP C/S 模型，注意以下几点：**

1. socket



参数 domain：AF\_INET ——> AF\_UNIX/AF\_LOCAL

1. bind



地址结构：sockaddr\_in ——> sockaddr\_un

// 网络通信

struct sockaddr\_in serv\_addr;

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

serv\_addr.sin\_port = htons(SERV\_PORT);

bind(sockfd, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr));

// 本地套接字

struct sockaddr\_un serv\_addr;

serv\_addr.sun\_family = AF\_UNIX;

strcpy(serv\_addr.sun\_path, "serv\_socket");

len = offsetof(struct sockaddr\_un, sun\_path) + strlen("serv\_socket");

bind(sockfd, (struct sockaddr \*)&serv\_addr, len);

【注意】：

求成员变量在结构体中的偏移位置

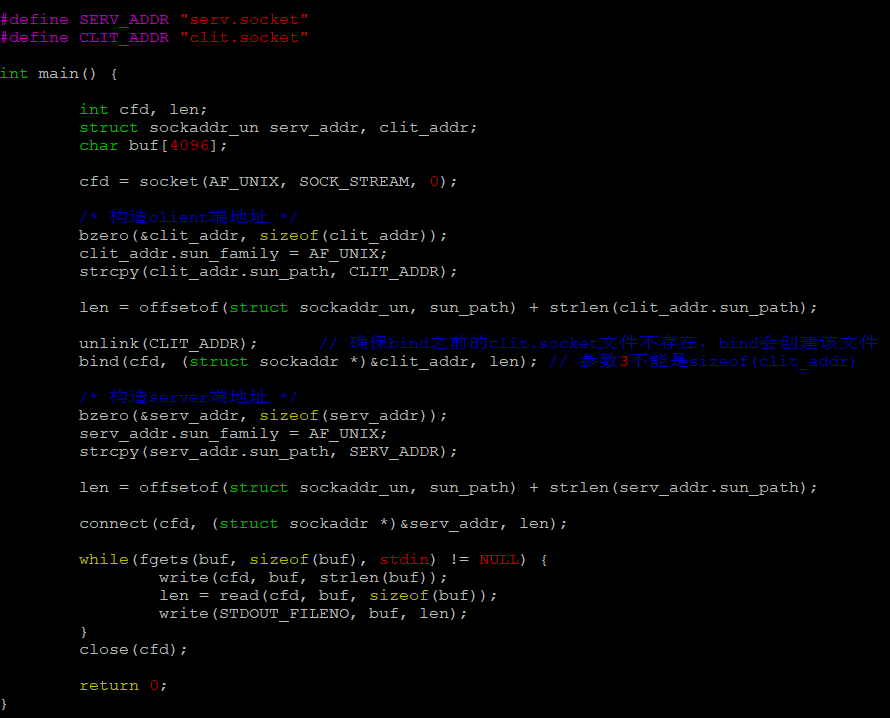
1. bind函数调用成功，会创建一个socket。为保证bind成功，通常在bind之前使用unlink()函数删除原来的socket
2. 客户端不能依赖“隐式绑定”。并且应该在通信建立过程中，创建且初始化2个地址结构。
3. clit\_addr ——> bind()
4. serv\_addr ——> connect()

#### 6.1 代码

##### server代码



##### client代码



#### 6.2 网络套接字对比本地套接字





## 四、Libevent库

### 1、Libevent框架

#### 1.1、创建event\_base（乐高底座）

【头文件】：#include <event2/event.h>

**struct event\_base \*event\_base\_new(void);**

struct event\_base \*base = NULL;

base = event\_base\_new();

#### 1.2、创建事件

**常规事件 event ——> event\_new()**

**带缓冲区的事件 bufferevent ——> bufferevent\_new()**

#### 1.3、将事件添加到base上

**int event\_add(struct event \*ev, const struct timeval \*tv);**

ev：event\_new()函数返回的事件

tv：为NULL，不会超时。意为：一直等待事件被触发，回调函数会被调用

为0，等待期间，检查事件没有被触发，时间到，回调函数依旧会被调用

#### 1.4、循环监听事件满足

**int event\_base\_dispatch(struct event\_base \*base);**

base：event\_base\_new函数的返回值

成功：0，失败：-1

**只有event\_new中指定了EV\_PERSIST才持续触发，否则只触发一次，就跳出循环**

通常这样：EV\_WRITE|EV\_PERSIST、EV\_READ|EV\_PERSIST

##### 其他循环

**int event\_base\_loopexit(struct event\_base \*base, const struct timeval \*tv);**

在指定时间后停止循环

**int event\_base\_loopbreak(struct event\_base \*base);**

立即停止循环

#### 1.5、释放event\_base

**void event\_base\_free(struct event\_base \*base);**

#### 1.6、相关函数了解

##### 查看支持哪些多路IO

const char \*\*event\_get\_supported\_methods(void);

##### 查看当前用的多路IO

const char \* event\_base\_get\_methods(const struct event\_base \*base);

##### 查看fork后子进程使用的event\_base

int event\_reinit(struct event\_base \*base);

成功：0，失败：-1

使用该函数后，父进程创建的base才能在子进程中生效

### 2、常规事件event

**#include<event2/event.h>**

#### 2.1、创建一个事件

struct event \*event\_new(struct event\_base \*base, evutil\_socket\_t fd, short what,

event\_callback\_fn cb, void \*arg);

base：event\_base\_new()返回值

fd：绑定到event上的文件描述符，如fifo的读端

what：EV\_READ

EV\_WRITE

EV\_PERSIST持续触发 可以理解为：while(read())或while(write())，结

合event\_base\_dispatch()函数使用

**cb：回调函数**

typedef void(\*event\_callback\_fn)(evutil\_socket\_t fd, short, void\*);

#### 2.2、将事件添加到base上

**int event\_add(struct event \*ev, const struct timeval \*tv);**

ev：event\_new()函数返回的事件

tv：为NULL，不会超时。意为：一直等待事件被触发，回调函数会被调用

为0，等待期间，检查事件没有被触发，时间到，回调函数依旧会被调用

#### 2.3、将事件从base上拿下来

**int event\_del(struct event \*ev);**

ev：event\_new()函数返回的事件

#### 2.4、释放事件

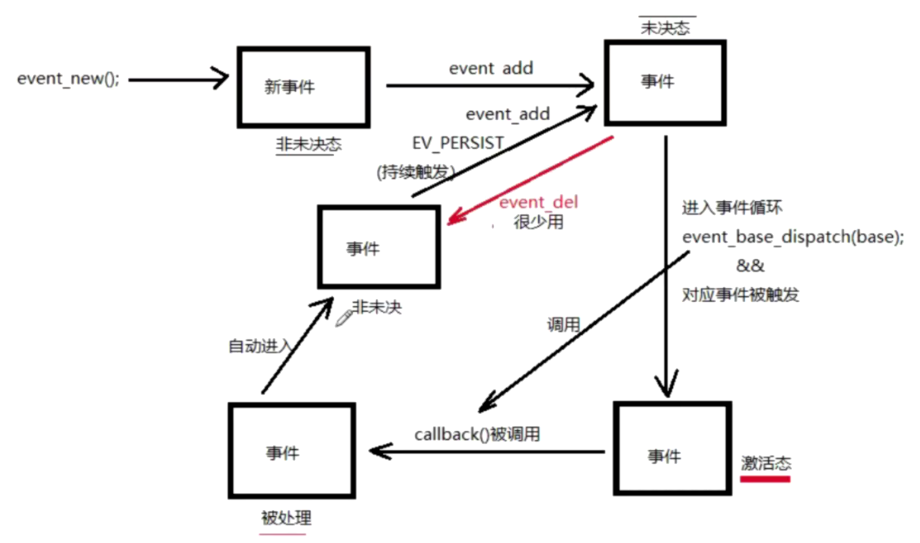
**int event\_free(struct event \*ev);23**

ev：event\_new()函数返回的事件

#### 2.5、未决和非未决

**未决：有资格被处理，但还没有被处理**

**非未决：没有资格被处理**



### 3、缓冲区事件bufferevent

**#include<event2/bufferevent.h>**

**原理**：bufferevent有两个缓冲区；也是队列实现，只能读一次，先进先出

**读**：读缓冲区有数据——>读回调函数被调用——>使用bufferevent\_read()(顶替read()函数)——>从读缓冲区中读数据

**写**：使用bufferevent\_write()——>向缓冲区中写数据——>该缓冲区有数据自动发送给对端——>写完，发送成功，回调函数被调用（通知写数据完成）

#### 3.1、缓冲区事件创建、销毁

##### 创建：

struct bufferevent \*bufferevent\_socket\_new(struct event\_base \*base, evutil\_socket\_t fd,

enum bufferevent\_options options);

**参数：**

base：event\_base\_new函数的返回值

fd：bufferevent绑定的文件描述符lfd。类比event\_new()

options：BEV\_OPT\_CLOSE\_ON\_FREE只用这一个即可。

**返回值：**

成功创建的bufferevent事件对象

##### 释放：

void bufferevent \* bufferevent\_socket\_free(struct bufferevent \*bev)

bev：创建的bufferevent事件对象

#### 3.2、给读写缓冲区设置回调

event事件创建时默认绑定回调函数

void bufferevent\_setcb(struct bufferevent \*bufev,

bufferevent\_data\_cb readcb,

bufferevent\_data\_cb writecb,

bufferevent\_event\_cb eventcb,

void \*cbarg)

**参数：**

bufev：bufferevent\_socket\_new()函数的返回值

readcb：读缓冲对应的回调，自己封装，在其内部读数据。

【注意】：使用bufferevent\_read()读，而不是read()

writecb：不用它，传NULL

eventcb：可传NULL

cbarg：回调函数用的参数

##### readcb对应的回调函数

**typedef void (\*bufferevent\_data\_cb)(struct bufferevent \*bev, void \*ctx);**

如：void read\_cb(struct bufferevent \*bev, void \*arg) {

……

bufferevent\_read(); //读数据，类似于read

}

读数据：从bufferevent输入缓冲区中移除数据

**size\_t bufferevent\_read(struct bufferevent \*bufev, void \*data, size\_t size);**

通常用在readcb中，替代read()

**int bufferevent\_write(struct bufferevent \*bufev, const void \*data, size\_t size);**

通常用在bufferevent\_read之后，替代write()

##### writecb对应的回调函数

不需要，鸡肋！！！

如：void write\_cb(struct bufferevent \*bev, void \*arg) {

……

bufferevent\_read(); //读数据，类似于read

}

##### eventcb对应的回调函数

**typedef void (\*bufferevent\_data\_cb)(struct bufferevent \*bev, short events, void \*ctx);**

如：void event\_cb(struct bufferevent \*bev, short events, void \*arg) {

……

}

**参数：**

events：BEV\_EVENT\_CONNECTED请求的连接过程已经完成，实现客户端时可用。

#### 3.3、禁用、启动缓冲区

**默认**：新建的bufferevent**写缓冲是enable**的，而，**读缓冲是disable**的。

**void bufferevent\_enable(struct bufferevent \*bufev, short events)**; 启用缓冲区

通常用来启用bufferevent的read缓冲

void bufferevent\_disable(struct bufferevent \*bufev, short events); 禁用

events：EV\_READ、EV\_WRITE、EV\_READ|EV\_WRITE

short bufferevent\_get\_enableed(struct bufferevent \*bufev);

获取缓冲区的禁用状态，需要借助&来得到

#### 3.4、客户端连接服务器

**socket()；connect();**

**int bufferevent\_socket\_connect(struct bufferevent \*bev, struct sockaddr \*address,**

**int addrlen)**

参数：

bev：bufferevent事件对象（封装了fd）

address、len：等同于connect()参2/3

#### 3.5、服务器创建监听器

**【头文件】：#include <event2/listener.h>**

struct evconnlistener \* evconnlistener\_new(

struct event\_base \*base,

evconnlistener\_cb cb,

void \*ptr,

unsigned flags,

int backlog,

evutil\_socket\_t fd);

**struct evconnlistener \* evconnlistener\_new\_bind(**

**struct event\_base \*base,**

**evconnlistener\_cb cb,**

**void \*ptr,**

**unsigned flags,**

**int backlog,**

**const struct sockaddr \*sa,**

**int socklen);**

参数：

**cb：监听回调函数**。接收连接之后，用户要做的操作，进行通信。

ptr：回调函数的参数

flags：“可识别的标志”

LEV\_OPT\_CLOSE\_ON\_FREE：释放bufferevent时关闭底层传输端口。这将关

闭底层套接字，释放底层bufferevent等

LEV\_OPT\_REUSEABLE端口复用。可以“|”位或连接

backlog：listen的2参。传-1：表使用默认最大值

sa：服务器的IP+Port

socklen：sa的大小

返回值：成功创建的监听器

**这一个函数，可完成系统调用socket()、bind()、listen()、accept()的作用。**

##### 回调函数类型

typedef void (\*evconnlistener\_cb) (

struct evconnlistener \* listener,

evutil\_socket\_t sock,

struct sockaddr \*addr,

int len,

void \*ptr);

参数：

listener：evconnlistener\_new\_bind函数的返回值

sock：用于通信的文件描述符cfd

addr：客户端的IP+端口

len：addr的len

ptr：外部ptr传递进来值

【注】：该回调函数，不由我们调，是框架自动调用。因此，只需知晓参数含义即可。

#### 3.6、libevent实现TCP服务器流程

1. 创建event\_base

2. 创建服务器连接监听器evconnlistener\_new\_bind()；

3. 在evconnlistener\_new\_bind的回调函数中，处理接收连接后的操作；

4. 回调函数被调用，说明有一个客户端连接上了，会得到一个新的fd，用于跟客户端通信（读、写）；

5. 在监听器回调函数中使用bufferevent\_socket\_new创建一个新bufferevent事件，将fd封装到这个事件对象中；

6. 使用bufferevent\_setcb给这个事件对象的read、write、event设置回调；

7. 设置bufferevent的读写缓冲区enable/disable；

8. 接收、发送数据bufferevent\_read() / bufferevent\_write()；

9. 启动循环监听event\_base\_dispatch；

10. 释放资源。

#### 3.7、libevent实现TCP客户端流程

1. 创建event\_base；

2. 使用bufferevent\_socket\_new()创建一个用跟服务器通信的bufferevent事件对象；

3. 使用bufferevent\_socket\_connect()连接服务器；

4. 使用bufferevent\_setcb()给bufferevent对象的read、write、event设置回调；

5. 设置bufferevent对象的读写缓冲区enable/disable；

6. 接收、发送数据bufferevent\_read() / bufferevent\_write()；

7. 启动循环监听event\_base\_dispatch

8. 释放资源。

## 五、简易Web浏览器

实现一个简单的Web服务器myhttpd。能够给浏览器提供服务，供用户借助浏览器访问服务器主机中的文件。

### 1、超文本标记语言HTML

超文本标记语言（Htyper Text Mark-up Language HTML）是构成网页文档的主要语言。可以说明文字、图形、动画、声音、表格、链接等。在计算机中以.html、.htm作为扩展名，可以被浏览器访问。

#### 1.1、简介

##### HTML特点

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

大多数标签是成对出现的，有开始，有结束 .<html>…</html>

不成对出现的称之为短标签<br/> <hr/>

##### 标签的属性和值

属性 = “属性值”

<font color=”red”>hello, world</font>

属性值建议加引号

##### html组成部分

html文件结构包括头(head)和主体(body)两大部分，头部miaoshudeshiliulanqi，所需信息，而主题则包含所要说明的具体内容。可在Linux下用浏览器打开一个网页，使用Ctrl+u查看该网页的html文件形式，Ctrl+w恢复。

<idoctype html>声明文档类型

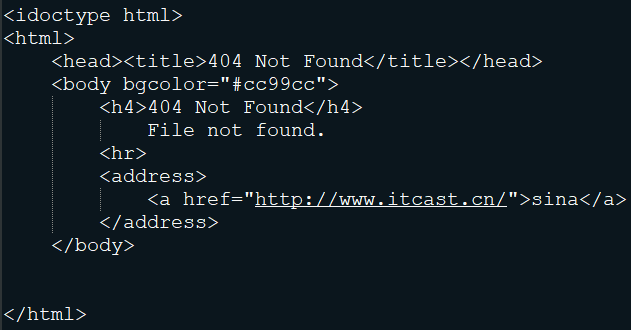
<html>文档的头部和主体内容 </html>跟标签

<head>文档的头部信息 </head>头部标记只能有一对

<title>显示在头部信息 </head>头部标记只能有一对

<body></body>主体标记位于<html>之内，<head>标记之后

简单的错误网页形式，如：



##### 注释

<!-- 我是一个html注释 -->

#### 1.2、常用标签

学习html，主要是标签的使用，能熟练的掌握各种常用标签的功能，特性，可以大大提高开发速度h额效率。初学的同学可以从以下标签学习。

##### 文本和标题标签

* **标题标签**

<h1></h1> //最大 只有一个，搜索引擎优化：seo

<h2></h2>

…

<h6></h6> //最小 1-6依次变小，自动换行

* **文本标签**

<font></font>

属性：

color：文字颜色 “red” “#AAFF00” rgb(165,129,135)

size：文字大小，范围1 – 7，依次增大

* **文本格式化标签**

加粗：

<strong></strong>或<b></b>

工作里尽量使用strong

倾斜：

<em></em>或<i></i>

工作里尽量使用em

删除线

<del></del>或<s></s>

工作里尽量使用del

下划线

<ins></ins>或<u></u>

工作里尽量使用ins

* **段落**

<p>xxx</p>

特点：上下自动生成空白行

* **块容器**

<div></div>

属性：

align：对齐方式 left, center right

* **换行**

<br/>

* **水平线**

<hr/>

属性：

size：任意大小

color：3种表示法

##### 列表标签

* 无序列表

标签

<ul>

<li></li>列表项

</ul>

属性：type

实心圆圈：disc – 默认

空心圆圈：circle

小方块：square

* 有序列表

标签

<ol>

<li></li>列表项

</ol>

属性：

type -- 序号

1 – 默认

a

A

i - 罗马数字（小）

I - 罗马数字（大）

start

从序号的什么位置开始

* 自定义列表

标签

<dl>

<dt></dt> 小标题

<dd></dd> 解释标题

</dl>

##### 图片标签

<img **src**=”bird.gif” **alt**=”图片加载失败” **title**=”小鸟飞翔！” **width**=”300” **height**=”200”/>

**属性**

src：图片的来源，必写属性

alt：替换文本 图片不显示的时候显示的文字

title：提示文本 鼠标放到图片上显示的文字

width：图片宽度

height：图片高度

**注意**

图片没有定义宽高的时候，图片按照百分之百比例显示

如果只更改图片的宽度或高度，图片等比例缩放

##### 超链接标签

<a **href**=”<http://jd.com>” **title**=”a dog” **target**=”\_blank”>超链接</a>

**属性**

href：去往的路径（跳转的界面）必写属性

title：提示文本，鼠标放到链接上显示的文字

target取值  
 \_self：默认值 在自身页面打开（关闭自身页面，打开链接页面）

\_blank：打开新页面

**锚链接（回到顶部）**

实现定义一个锚点标签：<p id=”top”>

超链接到锚点：<a herf=”#top”> 回到顶点 </a>

### 2、HTTP协议基础

#### 2.1、请求消息(Request)

浏览器——>发给——>服务器。主旨包含4部分：

* **请求行**：说明请求类型，要访问的资源，以及使用的http版本
* **请求头**：说明服务器要使用的附加信息
* **空 行**：必须！即使没有请求数据
* **请求数据**：也叫主题，可以添加任意的其他数据

以下是浏览器发给服务器的http协议头内容举例，注意：9行的空行（\r\n）也是协议头的一部分:



1是请求行，中间部分是协议头，9是空行，用于服务器确定协议头结束。

* **请求行**，用来说明请求类型,要访问的资源以及所使用的HTTP版本。

GET说明请求类型为GET，/562f25980001b1b106000338.jpg(URL)为要访问的资源，该行的最后一部分说明使用的是HTTP1.1版本。

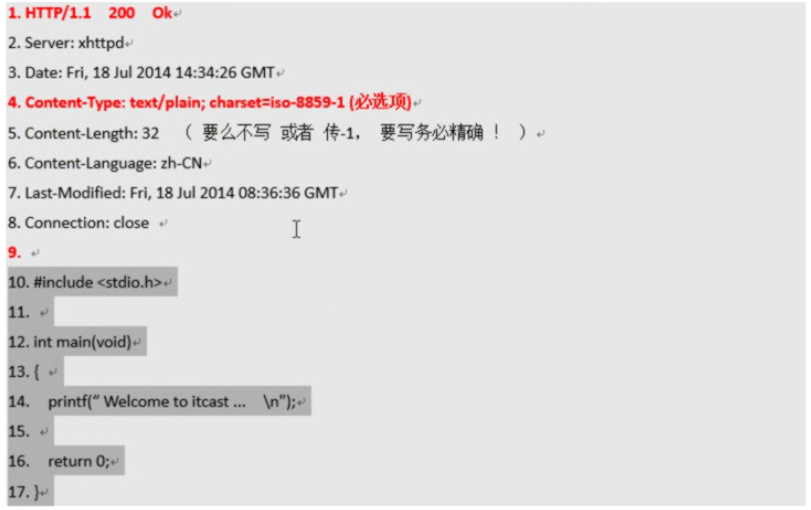
* **请求头部**，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息。
  + HOST，给出请求资源所在服务器的域名。
  + User-Agent，HTTP客户端程序的信息，该信息由你发出请求使用的浏览器来定义,并且在每个请求中自动发送等。
  + Accept，说明用户代理可处理的媒体类型。
  + Accept-Encoding，说明用户代理支持的内容编码。
  + Accept-Language，说明用户代理能够处理的自然语言集。
  + Content-Type，说明实现主体的媒体类型。
  + Content-Length，说明实现主体的大小。
  + Connection，连接管理，可以是Keep-Alive或close。
* **空行**，请求头部后面的空行是必须的即使第四部分的请求数据为空，也必须有空行。
* **请求数据**也叫主体，可以添加任意的其他数据。

#### 2.2、响应消息(Response)

服务器——>发给——>浏览器。主旨包含4部分：

* **状态行**：包括http协议版本号，状态码，状态信息
* **消息报头**：说明客户端要使用的一些附加信息
* **空行**：必须！
* **响应正文**：服务器返回给客户端的文本信息

以下是经服务器按照http协议，写回给浏览器的内容举例，1-9行是协议头部分



4表示响应的数据类型，9行以后是响应的数据本体。

* **状态行**，由HTTP协议版本号， 状态码， 状态消息 三部分组成。

第一行为状态行，（HTTP/1.1）表明HTTP版本为1.1版本，状态码为200，状态消息为OK。

* **消息报头**，用来说明客户端要使用的一些附加信息。

第二行至第八行为消息报头，Date:生成响应的日期和时间；Content-Type:指定了MIME类型的HTML(text/html),编码类型是UTF-8。

* **空行**，消息报头后面的空行是必须的。
* **响应正文**，服务器返回给客户端的文本信息。空行后面的html部分为响应正文。

#### 2.3、HTTP请求方法

* GET

请求指定的页面信息，并返回实体主体。

* 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）。数据被包含在请求主体中。POST请求可能会导致新的资源的建立和（或）已有资源的修改。

* HEAD

类似于GET请求，只不过返回的响应中没有具体内容，用于获取报头。

* PUT

从客户端向服务器传送的数据取代指定的文档的内容。

* DELETE

请求服务器删除指定的页面。

* 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。

* OPTIONS

允许客户端查看服务器的性能。

* TARCE

回显服务器收到的请求，主要用于测试或诊断。

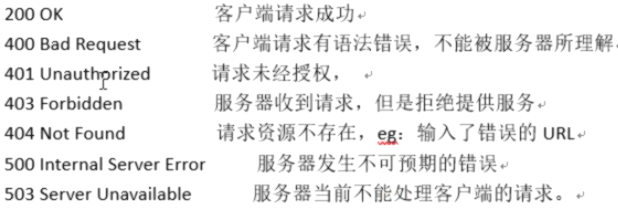
**GET和POST的区别：**

* + 最直观的区别就是GET把参数包含在URL中，POST通过request body传递参数。
  + GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
  + GET请求在URL中传送的参数是有长度限制。（大多数）浏览器通常都会限制url长度在2K个字节，而（大多数）服务器最多处理64K大小的url
  + GET产生一个TCP数据包；POST产生两个TCP数据包。对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100（指示信息—表示请求已接收，继续处理）continue，浏览器再发送data，服务器响应200 ok（返回数据）

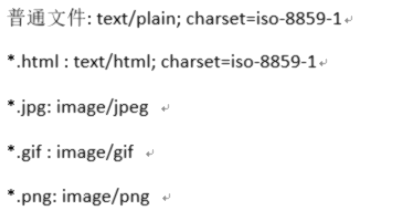
#### 2.4、HTTP常用状态码

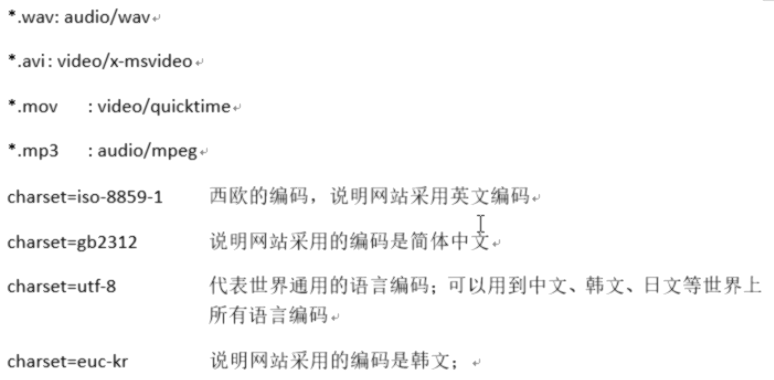
状态码有三位数字组成，第一个数字定义了响应的类别，共分五种类别：

* 1xx：指示信息—表示请求已接受，继续处理
* 2xx：成功—表示请求已被成功接收、理解、接受
  + 200 OK：客户端请求被正常处理
  + 206 Partial connent：客户端进行了范围请求
* 3xx：重定向—要完成请求必须进行更进一步的操作
  + 301 Moved Permanently：永久重定向，该资源已被永久移动到新位置，将来任何对该资源的访问都要使用本响应返回的若干个URL之一
  + 302 Found：临时重定向，请求的资源现在临时从不同的URL中获得
* 4xx：客户端错误—请求有语法错误或请求无法实现
  + 400 Bad Request：请求报文有语法错误
  + 403 Forbidden：请求被服务器拒绝
  + 404 Not Found：请求不存在，服务器上找不到请求的资源
* 5xx：服务器端错误—服务器未能实现合法的请求
  + 500 Internet Server Error：服务器在执行请求时出现错误
* 常见状态码



#### 2.4、常见网络文件类型







### 3、思路

1. get\_line()获取http协议的第一行；
2. 从首行中拆分 GET、文件名、协议版本。获取用户请求的文件名；
3. 判断文件是否存在。stat()函数判断；
4. 判断是文件还是目录；
5. 如果是文件，open —> read —> 写回给浏览器；
6. 先写http应答协议头：http/1.1 200 ok …；
7. 再写文件数据。

### 4、开发注意事项

#### 4.1、浏览器请求ico

准备一个favicon.ico文件放置到服务器提供访问的资源目录中。

浏览器在请求图片的同时，会请求一个ico图标，用于浏览器<title>标签文字部分前端的小图标显示。

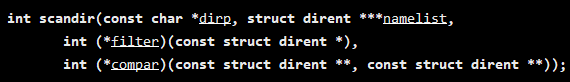
这个ico的文件名固定——favicon.ico。因此，自行准备一个ico文件，放置于服务器提供给浏览器访问的目标目录即可。

#### 4.2、容错处理

**返回值一定要检查！！！**

#### 4.3、快捷遍历目录scandir()

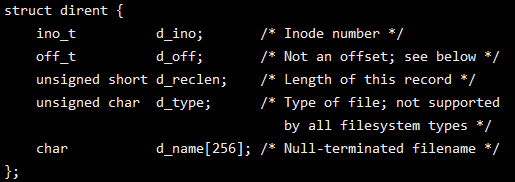
服务器端，可以使用文件操作时“递归遍历目录”的源码，实现遍历目录内文件名，回显给浏览器。另外标准C库中，提供了scandir函数，可以快捷地实现该功能。函数原型如下：



**scandir(待操作目录，&子目录项列表数组，过滤器(通常NULL)，alphasort)**

dirp：待访问的目录名称

struct dirent类型：(参考readdir()函数)



compar：参数取如下函数即可

int alphasort(const void \*a, const void \*b)；默认是

**调用：**

struct dirent\*\* namelist;

int num = sacndir(dirname, &namelist, NULL, alphasort);

for (int i = 0; i < num; i++)

char \*name = namelist[i] -> d\_name;