Linux网路编程基础及并发服务器

主要内容如下:

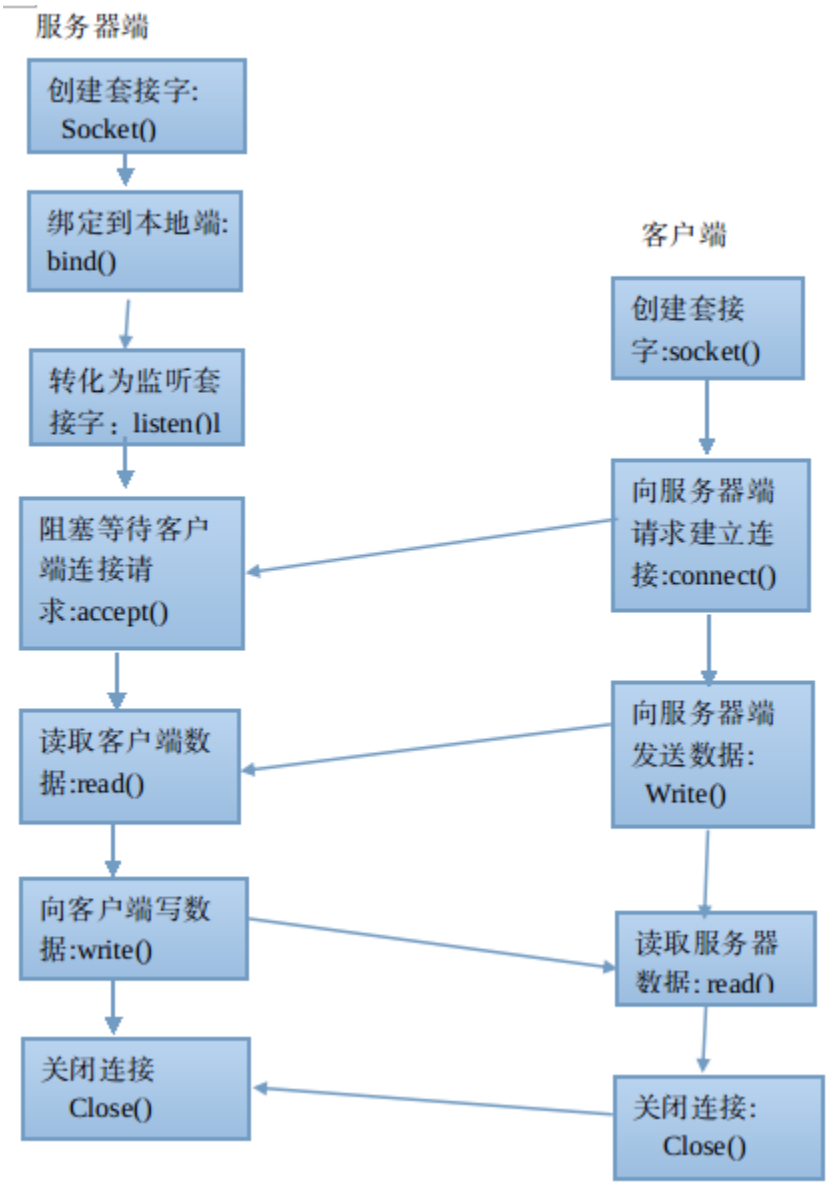
- 简介网络模型以及相关的socket编程
- 简介多进程编程
- 简介多线程编程
- 简介I/O复用
- 通过多进程，多线程以及I/O复用实现服务器并发

引言

- 网络中的实际应用大多都可以归纳为客户机/服务器模型(Clinet/Server模型，C/S模型)，其中客户机是指请求服务的一方，服务器是指提供某种服务的一方。
- 客户机/服务器模型即可以使用TCP协议也可医用UDP协议，或两者混合使用，可根据需要而定。
- 在客户机/服务器模型中，通常服务器端的IP地址和端口号是固定的，客户端程序连接到服务器IP和端口。
- socket,即套接字，是最流行最通用的网络通信应用程序的开发接口。现在不论是Windows还是Linux都使用socket来开发网络应用程序。通常Linux下的网络编程就是指套接字编程。

1.网络模型及socket编程

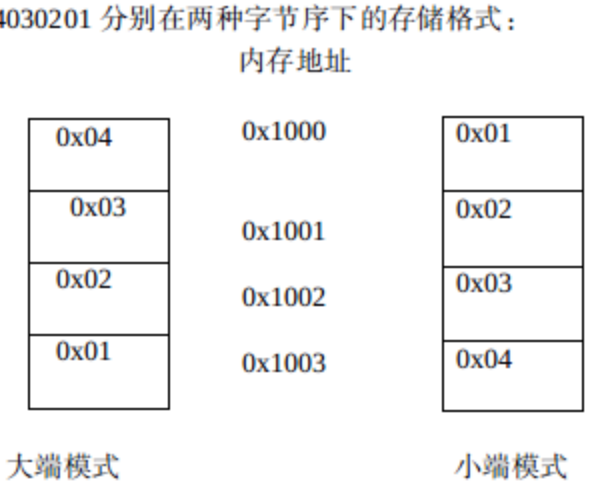
1.1.面向连接的C/S模型框图



1.2.编程之前的准备

- 学习**socket**地址**API**，先要理解主机字节序和网络字节序
  - *字节序*: 现在CPU的累加器一次都能装载（至少）4字节(这里指32位机，下同)，即一个整数。那么这4个字节在内存中的排列的顺序将影响它被累加器装成的整数的值，这就是字节序问题。
  - *大小端*: 字节序分为大端字节序和小端子节序。大端模式是指高字节数据存放在低地址处，低字节数据存放在高地址处;小端模式是指低字节数据存放在内存的低地址处，高字节数据存放在高地址处。具体区别如下图:

0x04030201 分别在两种字节序下的存储格式：



- 如何检查自己电脑的字节序呢?

```
================================================================================
        Filename:  test.c

        Description:  字节序的检查

        Version:  1.0
        Created:  2014年08月02日 17时44分45秒
        Revision:  none
        Compiler:  gcc
        CopyRight: open , free , share
        Author:  yexingkong(zhangbaoqing)
        Email: abqyexingkong@gmail.com
        Company:  Xi'an University of post and Telecommunications
================================================================================

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
int i = 0;

union {
        int value;
        char ch[sizeof(int)];
```

```
    }test;

    test.value = 0x04030201;

    for (i = 0;i < sizeof(int); i++)
    {
        printf("%p --> %#x\n",&test.ch[i],test.ch[i] );
    }
    printf("\n");


    return EXIT_SUCCESS;
}
```

通过打印对比内存中所存的数据的以及数据单元内存地址就可观察出来。。。。

- 在网络上传输数据时,由于数据传输的两端可能对应不同的硬件平台，采用的存储字节顺序也可能不一致，因此TCP/IP协议规定了在网络上必须采用网络字节顺序(也就是大端模式).而现代PC大多采用小端模式，因此小端模式又被称为主机字节序。为了解决这种网络上两台主机使用不同字节序所造成的错误，所有就有了，发送端总是把要发送的数据转化成大端字节序数据后再发送，而接收端知道对方传过来的数据总是采用大端字节序，所以接收端可以根据自身的字节序决定是否对接收的数据进行转换(小端机转换，大端机不转换)。

1.3.套接字的地址结构

> 1.结构struct sockaddr定义了一种通用的套接字地址，它在linux/socket.h中定义代码如下:
>
> ```
> struct sockaddr {
>             unsigned short      sa_family;  //地址类型，AF_XXX
>             char                sa_data[14];  //14字节的协议地址
>         };
> ```

其中，成员sa_family表示套接字的协议族类型，对应于TCP/IP协议该值为AF_INTI;成员sa_data,存储具体的协议地址。sa_data之所以被定义成14个字节，因为有的协议族使用较长的地址格式。一般在编程中并不对该结构体进行操作，而是使用另一个与它等价的数据结构：sockaddr_in。

> 2.每种协议族都有自己的协议地址格式，TCP/IP协议族的地址格式为结构体 struct sockaddr_in,它

> 在netinet/in.h头文件中定义，格式如下:

```
struct sockaddr_in {
    unsigned short      sin_family;  //地址类型
    unsigned short int  sin_port;    //端口号
    struct in_addr      sin_addr;    //IP地址
    unsigned char       sin_zero[8]; //填充字节，一般赋值为0
};
```

其中，成员sin_family表示地址类型，对于使用TCP/IP协议进行的网络编程，该值只能是AF_INET. sin_port是端口号，sin_addr用来存储32为的IP地址，数组sin_zero为填充字段，一般为0.

> 3.IP数据结构，struct in_addr的定义如下：

```
struct in_addr {
    unsigned long   s_addr;  //IP地址，要使用网络字节序表示
    }
```

结构体sockaddr的长度为16字节，结构体sockaddr_in的长度也为16字节，通常在编写基于TCP/IP协议的网络程序时，使用结构体sockaddr_in来设置地址，然后通过强制类型转换成sockaddr类型。

> 以下是设置地址信息的示例代码：

```
struct sockaddr_in sock;
sock.sin_family = AF_INET;   //设置使用IPV4 TCP/IP协议
sock.sin_port = htons(80);   //设置端口号
sock.sin_addr.s_addr = inet_addr("192.168.1.132");  //设置地址
memset(sock.sin_zero,0,sizeof(sock.sin_zero)); //将数据sin_zero清0
```

*memset()*函数原型为:
memset(void *s, int c, size_t n);
它将s指向的内存区域的前n个字节赋值为C指定的值。

1.4.转换函数

1.网络字节序转换函数

```
#include <netinet/in.h>
uint16_t  htons(uint16_t hosts);
uint32_t  htonl(uint32_t hostl);
uint16_t  ntohs(uint16_t nets);
uint32_t  ntohl(uint32_t netl);
```

- **htons**: 将16位的短整型数从主机字节序---->网络字节序
- **htonl**: 将32位的长整型数从主机字节序---->网络字节序
- **ntohs**: 将16位的短整形数从网络字节序---->主机字节序
- **ntohl**: 将32位的长整型数从网络字节序---->主机字节序

2.IP地址转换函数

人们习惯用可读性好的点分十进制来表示IP地址，但编程中我们需要先把它们转化为整数(二进制)方便使用，而记录日志时则相反，我们要把整数表示的IP地址转化为可读的字符串。

```
#include <arpa/inet.h>
in_addr_t  inet_addr(const char *str);
int inet_aton(const char *str, struct in_addr  *numstr);
char  *inet_ntoa(struct in_addr  inaddr);
```

函数中a代表ASCII串: n 代表数值(numeric)格式，是存在与套接字地址结构中的二进制值。

- **inet_addr**:将字符串形成的IP地址转换成32位二进制值的IP地址。str指向字符串形式的IP地址。函数调用成功，返回值为32为网络字节序的二进制值的IP地址。这个函数不对IP地址的有效性进行验证所有2^{32}个(0.0.0.0 ~ 255.255.255.255)可能的二进制值都认为是有效的IP地址。现在人们常用inet_aton函数代替inet_addr函数.
- **inet_aton**: 进行相同的转换。str指向字符串形式的Ip地址。numstr指向转换后的32位网络字节序的IP地址。如果成功返回1,否则，返回0.
- **inet_ntoa**: 将32为网络字节序的二进制IP地址准换成相应的点分十进制的IP地址。这个函数的参数是一个结构，而不是指向结构的指针。该函数的返回值所指向的串留在静态内存中嗯，所以函数是*不可重入*的。

1.5.套接字API

1.创建套接字

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocal);
```

- 参数 domain用于指定创建套接字所使用的协议族，它们在头文件linux/socket.h中定义。常用的协议族如下：
  - AF_UNIX: 创建只在本机内进行通信的套接字。
  - AF_INET: 使用IPV4 TCP/IP协议。
  - AF_INET6: 使用IPV4 TCP/IP协议。
- 参数type指定套接字的类型，可以取入下值:
  - SOCK_STREAM: 创建TCP套接字。
  - SOCK_DGRAM: 创建UDP数据报套接字。
  - SOCK_RAW: 创建原始套接字。
- 参数protocal通常设置为0,表示通过参数domain指定的协议族和参数type指定的套接字类型来确定使用的协议。
- 返回值： 执行成功返回一个新创建的套接字;若有错误发生则返回-1,错误代码存放在error中。

2.建立链接**(**客户端会用到**)**

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *sery_addr, socklen_t ad
dr_len);
```

- 参数sockfd是一个由函数sockfd创建的套接字.
- 参数serv_addr指定服务器IP地址和端口号。
- 参数addr_len是serv_addr参数的长度。
- 返回值：成功返回0， 错误返回-1.
- 如果该套接字的类型是SOCK_STREAM，则connect函数用于向服务器发出连接请求。
- 如果该套接字的类型是SOCK_DGRAM, 则connect并不建立真正的连接，他只是告诉内核与该套接字进行通行的目的地址（由第二个参数决定），只有该目的地址发来的数据才会被该socket接收。该函数用法：

```
struct sockaddr_in      serv_addr;
memset(&serv_addr, 0 , sizeof(struct sockaddr_in));//<==>bzero(&serv_ad
```

```
dr,sizeof(struct sockaddr_in)); 将sev_addr的各个字段清0
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(80); //转换字节序
if (inet_aton("192.168.1.132", &serv_addr.sin_addr) < 0)
{
  perror("inet_aton");
  exit(1);
}
//sock_fd为socket()函数的第一个参数值.
if (connect(sock_fd,(struct sockaddr *)&serv_addr, sizeof(struct sockad
dr_in)) < 0)
{
  perror("connect");
  exit(1);
}
```

3.绑定套接字

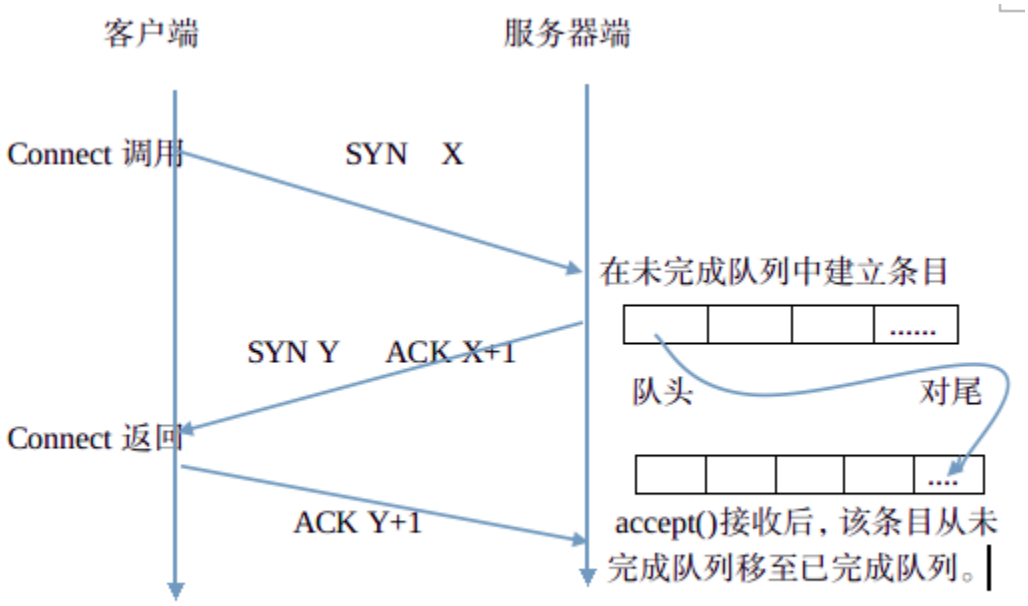
```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

- 参数sockfd为socket()函数的第一参数值。
- 参数my_addr指定服务器的IP和端口号.
- 参数addrlen是my_addr结构的长度。
- 返回值: 成功返回0,失败返回-1.
  - *错误码EACCES*: 被绑定的地址是受保护的地址，仅超级用户能够访问。如普通用户绑定端口(0~1023)上时（关于端口的大小以及普通用户能使用的范围自己可上网查）。
  - *错误码EADDRINUSE*: 被绑定的地址正在使用中。
- bind()函数存在的意义：
  - socket 函数只是创建套接字，但这个套接字将服务在那个端口上，程序并没指定。而我们知道服务器的IP地址和端口一般是固定的，因此在服务器端的程序中，使用bind函数将一个套接字和某个端口号绑定在一起。
- 用法和上面的connect()函数用法一样，就只是把函数名改成bind即可。

4.在套接字上监听

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- 参数sockfd为socket()函数第一个参数值。
- 参数backlog为指定该连接队列的最大长度。
- 返回值：成功返回0，失败返回-1.
- 用法比较简单，但要先定义个队列的长度。
- listen的作用:
  - 由于socket()创建的套接字是主动套接字，这种套接字可用来主动请求连接到某个服务器(connect函数).但作为服务器端程序，我们往往是被动请求连接的，也就是被动等待客户端前来搭讪。
  - 这时listen就会把服务器端的主动套接字转化为被动型，同时还要监听是否有客户端来搭讪。
- 注意：她只是观察是否有客户来连接，但并不能说立即就接受客户的连接。
- *细节知识*： 此函数维护着两个队列，即“未完成链接队列”和"已完成链接队列'.而对于这两个队列。如下图：



图为，tcp 三次握手中的两个队列的位置

- 图的意思就是，当服务器收到客户端发送的建立链接的SYN分节，TCP在未完成队列中创建一个新的条目，然后发送服务器的SYN分节到客户端，并附带对客户SYN的确认ACK。这个条目一直保存在未完成三次握手完毕，该条目将从未完成队列移至已完成队列的队尾。当进程调用accept()函数是，取出已完成链接队列头条目返回给进程，队列为空，进程将睡眠，直到有新的条目到达时才唤醒。

5.接受连接

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr * addr, socklen_t *addrlen);
```

- 参数sockfd是由函数socket创建。
- 参数addr用来保存发起连接请求的主机的地址和端口。
- 参数addrlen是addr所指向的结构体的大小。
- 返回值： 成功返回一个新的代表客户端的套接字,错误返回-1.
- 作用：就在当服务器监听到客户来连接了，同时没发现错误，好那就接受他的连接请求。并为他开辟一个秘密的服务专线，以后客户和服务器就是通过这个专线来交流的，这个专线ID就是函数的返回值。
- 用法：

```
int client_fd;
int client_len;
struct sockaddr_in  client_addr;  //保存客户端的Ip和端口等信息
.....
client_len = sizeof(struct sockaddr_in);
client_fd = accept(sock_fd, (struct sockaddr *)&client_addr, &clie
nt_len);
if (client_fd < 0)
{
  perror("accept");
  exit(1);
}
```

6.发送数据**(TCP)**

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int client_fd, const void *msg, size_t len, int flags);
```

- 参数client_fd 为已建立好的套接字描述符(accpet的返回值).
- 参数msg 指向存放代发送数据的缓冲区。

- 参数len 为代发送数据的长度。
- 参数flags 为控制选项，一般设置为0.
- 返回值: 成功返回实际发送数据的字节数，错误返回-1.
- *注意: 执行成功只能说明数据从当前应用进程缓冲区向套接字缓冲区写入数据成功，并不表示数据已经成功通过网络发送到目的地。如果要发送的数据长度大于该套接字的缓冲区剩余空间大小时，send()一般会被阻塞。
- 用法:

```
#define BUFFERSIZE      1000  //定义一次发送数据的长度
char    send_buff[BUFFERSIZE];
....
if (send(client_fd, send_fd, len, 0) < 0)
{
  perror("send");
  exit(1);
}
```

7.接收数据**(TCP)**

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int client_fd, void *buf, size_t len, int flags);
```

- 参数client_fd 为已建立好的套接字描述符(accpet的返回值).
- 参数buf 指向存放代接收数据的缓冲区。
- 参数len 为代接收数据的长度。
- 参数flags 为控制选项，一般设置为0.
- 返回值: 成功返回接收到的数据字节数，失败返回-1.
- 如果在指定的套接字上无数据到达是，recv()将被阻塞,如果该套接字设为非阻塞方式(fcntl()可设置)，则此时立即返回-1.

8.发送数据**(UDP)**

```
#include < sys/types.h >
#include < sys/socket.h >
```

```
int sendto ( int s , const void * msg, int len, unsigned int flags, con
st struct sockaddr * to , int tolen ) ;
```

- 参数s为已建好连线的socket,如果利用UDP协议则不需经过连线操作。
- 参数msg指向代发送数据的缓冲区.
- 参数len指定了代发送数据的长度。
- 参数flags是控制选项，含义和send()一致。
- 参数to用于指定目的地址。
- 参数tolen指定目的地址的长度。
- 返回值： 成功返回实际发送数据的字节数，失败返回-1.
- 常见用法:

```
  struct sockaddr_in addr;
  int addr_len = sizeof(struct sockaddr_in);
  char buffer[256];
//填写sockaddr_in 结构
  bzero ( &addr, sizeof(addr) );
  addr.sin_family=AF_INET;
  addr.sin_port=htons(PORT);
  addr.sin_addr.s_addr=htonl(INADDR_ANY) ; // INADDR_ANY表示0.0.0.0
,所有地址，不确定地址
if (sendto(sockfd,buffer,len,0,(struct sockaddr *)&addr,addr_len)
< 0)
{
  perro("sendto");
  exit(1);
}
```

9.接收数据**(UDP)**

```
#include<sys/types.h>
#include<sys/socket.h>
int recvfrom(int s,void *buf,int len,unsigned int flags ,struct sockadd
r *from ,int *fromlen);
```

- 参数s：标识一个已连接套接口的描述字。
- 参数buf：接收数据缓冲区。
- 参数len：缓冲区长度。
- 参数flags：调用操作方式。
- 参数from：（可选）指针，指向装有源地址的缓冲区。
- 参数 fromlen：（可选）指针，指向from缓冲区长度值。
- 返回值：成功返回实际发送数据的字节数，失败返回-1。
- 常见用法：

```
char                recv_buf[256];
struct sockaddr_in  src_addr;
int                 src_len;
src_len = sizeof(struct sockaddr_in);
if (recvfrom(sock_fd, recv_buf, sizeof(recv_buf), 0 ,(struct socka
ddr *)&src_addr, &ser_len) < 0)
{
  perror("recvfrom");
  exit(1);
}
```

10.关闭套接字

```
#include <unistd.h>
int close(int fd);
```

- 参数fd为一个套接字描述符。
- 返回值：成功返回0,错误返回-1.

示例代码

2.多进程编程

2.1什么是进程?

进程是一个动态的实体，是程序的一次执行过程。进程是操作系统资源分配的基本单位。进程和程序的

区别在于，进程是动态的，程序是静态的。进程是运行中的程序，程序是一些保存在磁盘上的可执行的代码。

2.2进程标识

每个进程都是通过唯一的进程ID来标识的。进程ID是一个非负整数。获得进程ID,可通过如下函数:

函数	功能
pid_t getpid(id)	获得进程ID
pid_t getppid(id)	获得父进程ID

2.3进程控制函数

*进程控制*包括创建进程，执行新进程，退出进程以及改变进程优先级等。

函数	功能
fork	用于创建一个新进程
exit	用于终止进程
exec	用于执行一个应用程序
wait	将父进程挂起，等待子进程终止
nice	改变进程的优先级
getpid	获取当前进程的进程ID

2.4创建进程

```
#include <sys/types.h>
#include <unistd.h>
pid_t   fork(void);
```

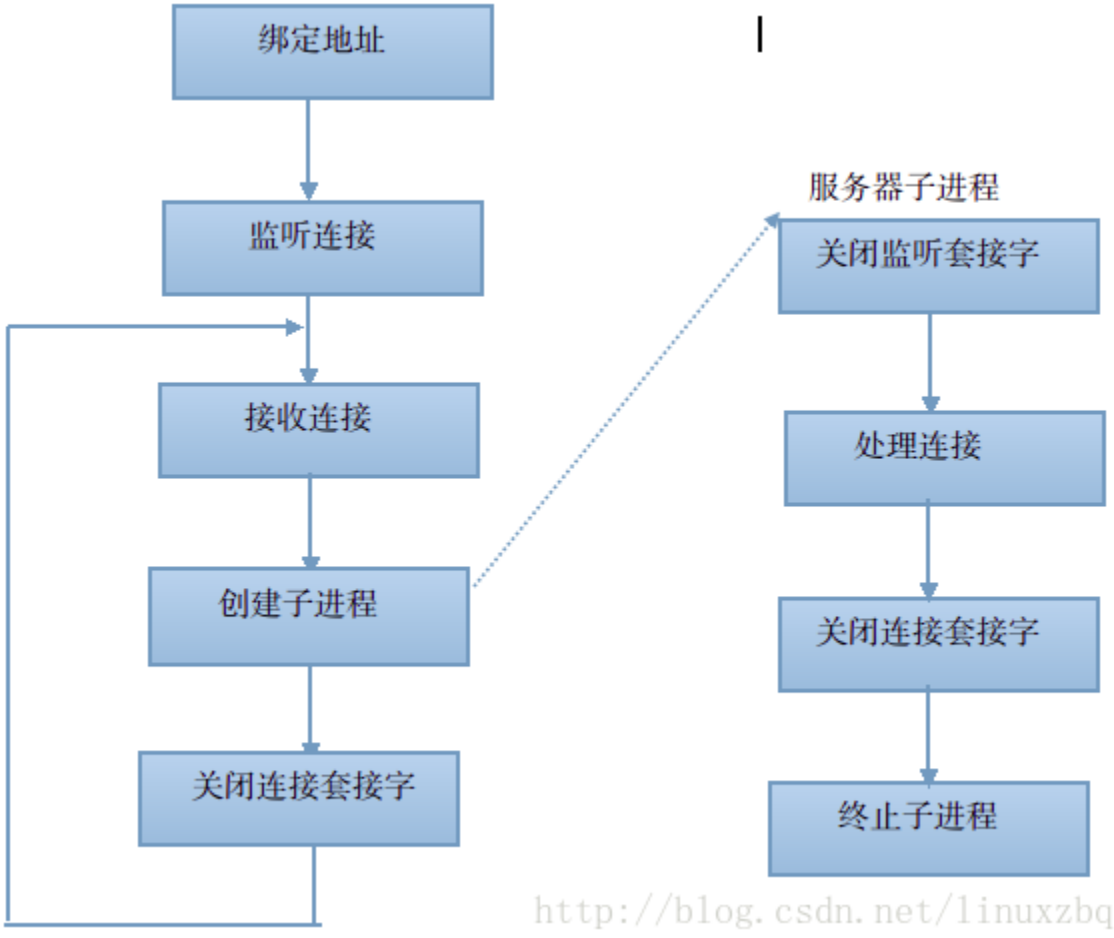
特别说明:

- 一般情况下，函数最多有一个返回值，但fork函数非常特殊，它有两个返回值. 即调用一次，返回两次.

- 成功调用后，当前进程实际上已经分裂为两个进程，一个是原来的父进程，另一个是刚刚创建的子进程,父子进程在调用**fork**函数的地方分开.
- fork函数有两个返回值，一个是父进程调用fork函数后的返回值，该返回值是刚刚创建子进程的ID;另一个是子进程中fork函数的返回值，该返回值是0.
- 两次返回不同的值，子进程返回0,父进程返回值为新创建进程ID，这样可以用来区分父，子进程。
- 函数调用失败，返回-1.
- 父子进程的执行顺序是无序的，由内核调度算法来决定。
- 子进程的代码与父进程完全相同，同时它还会复制(区别于:共享)父进程的数据(堆，栈数据和静态数据).数据的复制采用的所谓写时复制(copy on write),即只用在任一进程(父/子)对数据执行了写操作时，复制才会发生(先是缺页中断，然后操作系统给子进程分配内存并复制父进程数据)。
- 此外，创建子进程后，父进程中打开的文件描述符默认在子进程中也是打开的，且文件描述符的引用计数加1，父进程的用户根目录，当期工作目录等变量的引用计数均会加1.

2.5多进程并发服务器

- 并发过程图如下:



- 在这一过程中，父进程等待客户端请求。当这种请求到达时，父进程调用fork函数，产生一个子进

程，由子进程对该请求做处理。父进程则继续等待下一个客户的服务请求。这种情况下，在fork函数之后，父，子进程需要关闭各自不使用的描述符。

- 具体来讲就是,当父进程产生新的子进程后，父,子进程共享父进程在调用fork之前的所有描述符，一般情况下，接下来父进程只负责接收客户请求，而子进程只负责处理客户请求。关闭不需要的描述符既可以节省系统资源，又可以防止父，子进程同时对共享描述符进程操作，产生不可预计的后果。

- 示例

就到这里。。。。