**Unix环境下C++编程笔记**

**系统篇**

1. 使用man查看POSIX和ANSI C接口

|  |  |  |
| --- | --- | --- |
| 部分 | 说明 | 举例 |
| 1 | 系统指令（shell命令）或可执行文件 | ls,chmod,mkdir |
| 2 | 系统调用（内核提供的函数）[POSIX C] | write,open,chmod,fork |
| 3 | 库函数[ANSI C] | printf,fopen,fprintf,system |
| 4 | 特别的文件（通常在/dev） | tty,ttyS |
| 5 | 文件格式 | /etc/passwd |
| 6 | 游戏 | 略 |
| 7 | 杂项 | 略 |

1. errno和perror：errno在系统函数执行出错后会被置为非0值，这时候使用strerror(errno)可返回对应errno的出错信息(例如：No such file or directory)。也可使用perror(const char\* msg)附加打印自己的说明信息，例如perror(“Open a.txt”)会打印Open a.txt: No such file or directory，相当于cout<<”Open a.txt:”<<strerror(errno)<<endl。还有一点，任何函数都不会将errno的值置为0（即使执行成功），下面例子证明了这一点：

int fd = open("a.txt", O\_RDONLY);

if(fd == -1){

perror("Open a.txt");

}

fd = open("a.cpp", O\_RDONLY);

if(fd != -1){

perror("Open success test");

}

结果输出：

Open a.txt: No such file or directory

Open success test: No such file or directory

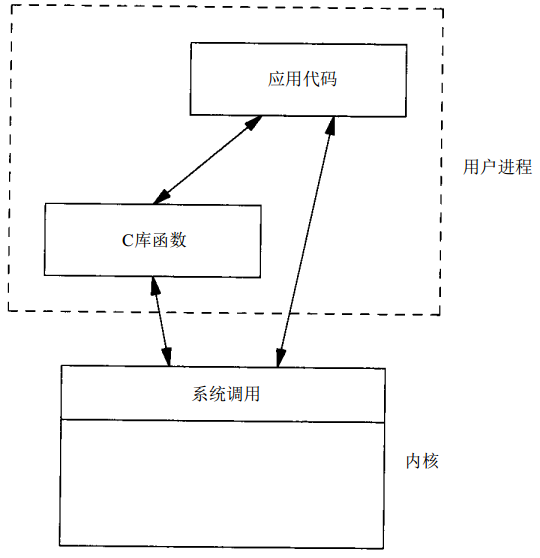
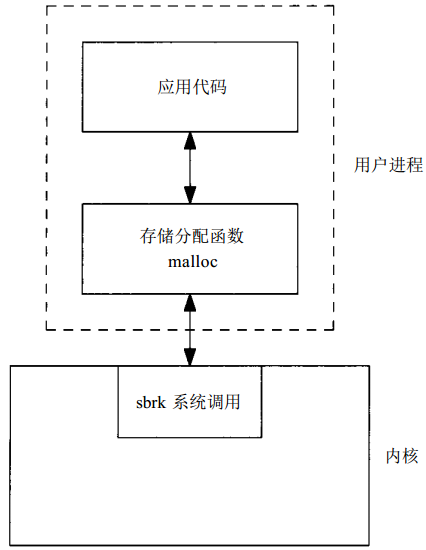
即使a.cpp存在这时候errno还是没有被置为0。所以请在出错处理的模块中使用strerror(errno)和perror(msg)。正常不可能出错的程序不要随便使用perror打印。

1. 系统调用和普通函数调用(包括用户函数、库函数)的区别：
   1. 系统调用是内核提供的开放API(open\write\time)，服务于普通函数调用，当然并不是所有的普通函数调用都依赖于系统调用(例如atoi, strlen)；使用系统调用可以实现库函数，使用库函数和系统调用可以实现应用程序，既然有些库函数是基于系统调用的封装，说明每次调用这些库函数API的时候都会间接进行系统调用，那为什么不直接使用系统调用编写应用程序？（这样看起来效率会更高）

系统调用是主要负责申请操作系统管理的资源（存储[sbrk]、文件描述符[fd]等）给应用程序，库函数则在应用程序中主要负责通过系统调用来获取资源进行当前进、线程的管理和分配使用，二者职责不同（一个负责资源申请，一个负责[进线程内的]资源管理）。如果直接使用系统调用开发应用程序，那么资源管理则必须由用户编写，正是因为如此才有库函数的存在，库函数就是为了提供应用层的编程接口而存在的。那么什么情况下必须使用系统调用接口？

和操作系统相关的接口，比如进程、线程、网络套接字编程（Linux和Windows不同）。不过好在C++11提出了线程thread，支持跨平台线程编程，相信以后会有更多的跨平台接口出现，系统调用可能会越来越少地出现在应用程序中。

* 1. 我们可以替换普通函数的不同实现，但无法改变系统调用的实现。例如malloc函数申请一块内存，如果我们不喜欢标准库的实现方式完全可以自己写一个，但无论我们写成什么样子，都依赖系统调用sbrk；
  2. 系统调用运行在内核态，内核态下，操作系统对所有硬件拥有完全访问权，可以执行机器能够支持的任何指令；普通调用运行在用户态，用户态下，用户程序只能执行机器支持指令的一个子集（程序状态字PSW中有一个位表示该指令是内核态还是用户态，通过判断PSW决定一条机器指令的类型）。有了用户态和内核态的区分，就限制了用户程序对硬件的访问权限，通过为用户程序提供部分（安全的）系统调用接口就可以避免用户程序（执行不当）对硬件造成的破坏，例如恶意删除硬件寄存器、随意修改操作系统运行内存地址内容（虽然有memset但系统限制了地址0~System的访问）。

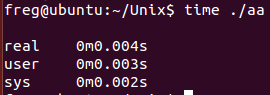


1. Unix系统限制
   1. 由于程序运行在计算机上，而计算机的资源是有限的，所以任何一个程序都不能无限制申请资源，应该规定程序的运行参数数量、每个进程可以同时打开的文件数量、最大文件名字长度等等；
   2. 程序运行时不同类型变量的大小，例如short、int、float是多少字节，与机器无关为了可移植。
   3. Ubuntu 14.04（x86\_64）的一些系统限制

|  |  |
| --- | --- |
| 限制项目 | 限制内容 |
| 文件名最大长度 | 255 |
| 路径最大长度 | 4096 |
| 进程打开的最大文件数 | 1024 |

1. 时间函数

time ./execProg获取程序执行时间



real: 程序实际执行时间=CPU时间+IO时间

user: 用户CPU时间（用户态CPU执行时间）

sys: 系统CPU时间（内核态CPU执行时间）

* 1. 获取日期(时间)：

time\_t t = time(NULL);

tm\* date = localtime(&t);

char buf[100];

strftime(buf, 100, "%Y-%m-%d %H:%M:%S %Z", date);

cout<<buf<<endl;

tm\* date2 = new tm;

sscanf(buf, "%d-%d-%d %d:%d:%d %s", &date2->tm\_year, &date2->tm\_mon, te2->tm\_mday, &date2->tm\_hour, &date2->tm\_min, &date2->tm\_sec, buf);

date2->tm\_year -= 1900;

date2->tm\_mon--;

time\_t t2 = mktime(date2);

cout<<difftime(t2,t)<<endl;

delete date2;

* 1. 计时：

timeval begin, end;

gettimeofday(&begin, NULL);

sleep(2);

gettimeofday(&end, NULL);

cout<<1.0\*(end.tv\_sec-begin.tv\_sec)+1.0\*(end.tv\_usec-begin.tv\_usec)/1000000<<endl;

1. 文件
2. 文件类型：
   1. 普通文件（包括二进制文件）
   2. 目录文件
   3. 字符特殊文件（设备驱动）
   4. 块特殊文件（磁盘驱动）
   5. FIFO
   6. 符号链接
   7. 套接字

使用lstat返回文件信息描述。lstat(const char\* filename, struct stat\* buf)

struct stat{

mode\_t st\_mode; // 文件类型

uid\_t st\_uid; // 文件所有者的ID

gid\_t st\_gid; // 文件所有者所属组的ID

time\_t st\_atime; // 上次访问时间

// …

};

1. 目录遍历

DIR\* dp;

struct dirent\* dirp;

if((dp = opendir("./")) == NULL){

perror("Open ./");

exit(0);

}

while((dirp = readdir(dp)) != NULL){

cout<<dirp->d\_name<<endl;

}

closedir(dp);

1. 文件系统的读、写、执行规范

|  |  |
| --- | --- |
| Rule 1 | 对某个文件进行访问（包括读写和ls），路径中的所有目录必须有可执行权限 |
| Rule 2 | 在目录中创建、删除文件，必须保证目录有写和可执行权限 |
| Rule 3 | 目录的权限是可执行，但仍可以读写里面某个文件的内容，但不可以添加或者删除文件 |

1. I/O缓存机制

IO缓存的本质是将需要往文件/终端设备写入或从文件/终端设备读出的内容暂时放置于内存里的一段缓冲区，等buffer满了或者调用flush后，才将内存中的缓冲区写入文件/终端设备或者指定区域。如果没有缓存，来一个字符就进行一次文件写入，相当于每个字符都会进行一次磁盘IO，速度会慢很多。

缓存的规则：

1. 标准出错不带缓存（为了及时看到错误信息）
2. 涉及终端设备的流是行缓存的；否则是全缓存的
3. 程序退出

case 1：

FILE\* fp = fopen("qqq", "w");

fwrite("123456", 1, 3, fp);

exit(0);

文件qqq有内容，exit在程序推出前清空流缓存，将流中数据写入磁盘

case 2：

FILE\* fp = fopen("qqq", "w");

fwrite("123456", 1, 3, fp);

\_exit(0);

文件qqq无内容，\_exit退出直接到内核，内核直接关闭程序

case 3：

ofstream out("qqq", ofstream::out | ofstream::trunc);

out<<"123";

exit(0);

文件out无内容，这也是C和C++的区别之处，exit本是ANSI C的标准，C++对C虽然兼容，但exit对流的缓冲处理也仅限于C的流处理，至于C++的流处理exit并不操心。

case 4：

void f(){

cout<<"f()"<<endl;

}

int main(int argc, char\*\* argv){

atexit(f);

cout<<"main exit"<<endl;

return 0;

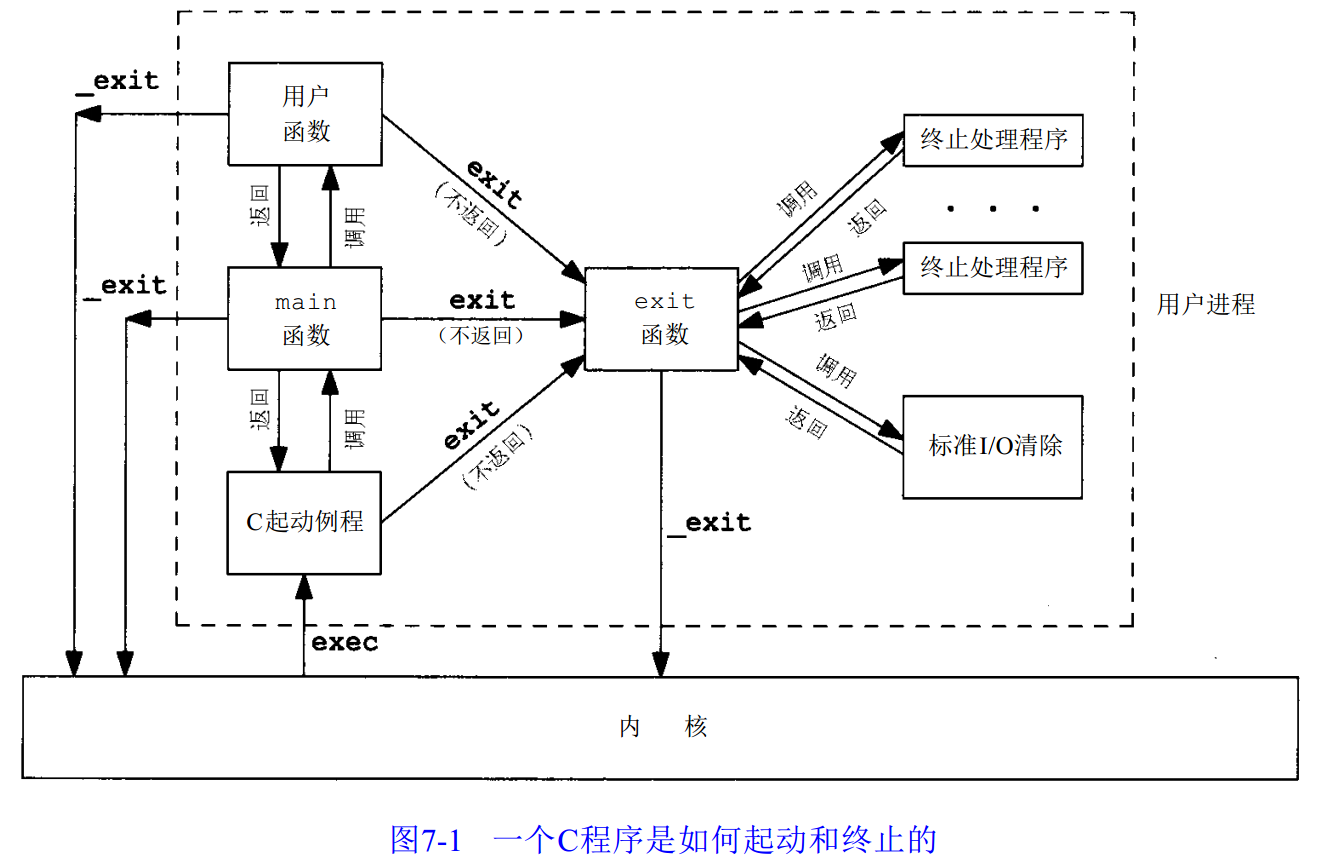
}

输出：

main exit

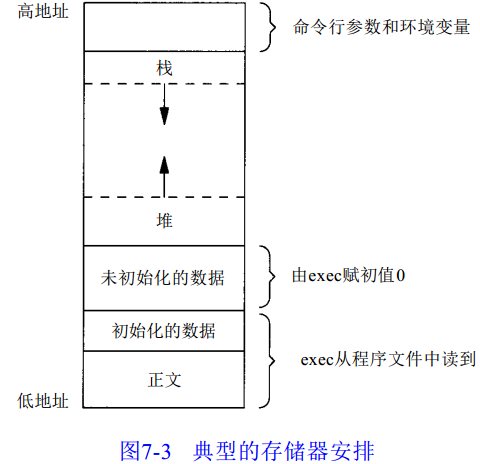
f()

可以用atexit()做一些程序退出前的用户处理



1. 程序内存分配
2. 逻辑上(指的是程序逻辑上的内存数据区域划分，对于任何程序[Java/C++/C]而言都一样)：
   1. 栈区：系统自动动态分配，例如局部变量和函数调用；
   2. 堆区：程序员手动动态分配，例如malloc, new；
   3. 全局静态区：包括全局变量（初始化和未初始化）、static数据；
   4. 常量区：常量；
   5. 代码区：程序执行代码。
3. 物理上(指的是使用C++编译器完成后用size myProgram查看可执行程序的实际物理空间由哪几部分构成，每个部分分别对应到逻辑上的哪些数据部分)：注：对于32位系统4GB的内存区域，Linux会将高地址0xC0000000-0xFFFFFFFF供操作系统使用，用户程序地址空间为0x00000000-0x0BFFFFFFF

|  |  |  |
| --- | --- | --- |
| 内存类型 | 物理内存区域 | 对应逻辑分区 |
| 动态区域（运行时动态分配的空数据间） | 栈(系统自动分配释放) | 栈区 |
| 堆(程序员手动分配释放) | 堆区 |
| 静态区域（编译时分配的数据空间） | data段 | 全局静态区的初始化变量+已初始化的static数据(全局和局部都算。而且初始化只包括初始化数值不为0的) |
| bss段 | 全局静态区的未初始化变量+未初始化的static数据(全局和局部都算。而且未初始化还包括初始化数值为0的) |
| text段 | 代码区+常量区 |



1. IO调用
2. 非阻塞：

void setfl(int fd, int flags){

int val;

if((val = fcntl(fd, F\_GETFL, 0)) < 0){

cout<<"Fcntl error"<<endl;

perror(strerror(errno));

exit(0);

}

val |= flags;

if(fcntl(fd, F\_SETFL, val) < 0){

cout<<"Fcntl error"<<endl;

perror(strerror(errno));

exit(0);

}

}

int main(int argc, char\*\* argv){

char buf[10000];

int fd = open("qqq.txt", O\_WRONLY | O\_CREAT | O\_TRUNC); // 如果直接在open加入文件打开状态O\_NONBLOCK就不需要用fcntl去改变文件状态

if(fd < 0){

cout<<"Open fail"<<endl;

perror(strerror(errno));

return 0;

}

setfl(fd, O\_NONBLOCK);

int number;

if((number = read(STDIN\_FILENO, buf, sizeof(buf))) == 0){

cout<<"No input"<<endl;

return 0;

}

char\* ptr = buf;

while(number){

cout<<"Write file ..."<<endl;

int n = write(fd, ptr, number);

if(n>0){

number -= n;

ptr += n;

}else{

perror(“Write error”);

}

}

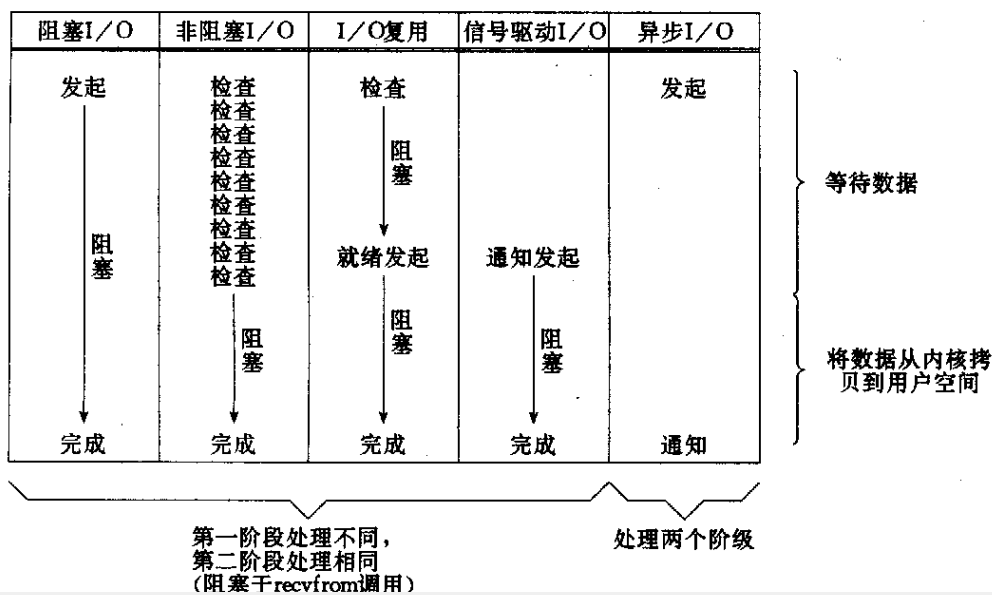
setfl(fd, O\_SYNC);

close(fd);

return 0;

}

1. 不同IO模型的比较
   1. 阻塞IO模型：进程从一个打开的文件描述符（可能是网络套接字，也可能是设备驱动文件）去读或者写数据，但是由于某种原因（数据还没有准备好或者缓冲区已满）而一直阻塞在IO调用处，适合对单个文件操作的情况。
   2. 非阻塞IO模型：进程从一个打开的文件描述符（可能是网络套接字，也可能是设备驱动文件）去读或者写数据，但是由于某种原因（数据还没有准备好或者缓冲区已满）在IO调用处不阻塞但立即返回IO失败（EAGAIN），然后用轮询的方式不断调用read/write操作直至可以对文件操作，但这种重复执行的方式会浪费CPU时间，适合对单个文件操作的情况。
   3. 异步IO模型：进程从一个打开的文件描述符（可能是网络套接字，也可能是设备驱动文件）去读或者写数据，但是由于某种原因（数据还没有准备好或者缓冲区已满）在IO调用处不阻塞，进程继续执行下面的程序，如果read/write就绪再进行从内核缓冲区到用户地址空间的拷贝，但目前Linux并没有对异步IO的实现。
   4. 多路复用：如果一个进程同时对多个文件描述符进行IO操作，但可能有些会阻塞有些可能可以立即进行IO操作，需要对所有的文件描述符进行轮询监听，哪个就绪就对哪个进行IO操作。

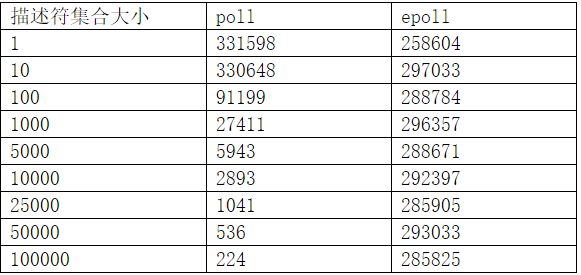


1. 多路复用

|  |  |  |
| --- | --- | --- |
| 类型 | 接口 | 说明 |
| select | int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout) | select监听的描述符序最大号+1[nfds]，最大为1024；超时设置timeout作为timeval结构可精确到微妙，如果timeout设置为NULL表示select设置为阻塞；可以选择监听的描述符集合是读就绪[readfds]、写就绪[writefds]还是异常[exceptfds]，如果readfds为NULL代表不监听读事件的文件描述符（以此类推）；返回就绪的文件描述符数量 |
| pselect | int pselect(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, const struct timespec \*timeout, const sigset\_t \*sigmask) | 和select一样，区别在于：   * 1. 使用timespec结构作为阻塞时间可精确到纳秒，但遗憾的是一般Linux系统时间精度到微妙为止了；   2. 多了信号屏蔽参数 |
| poll | int poll(struct pollfd \*fds, nfds\_t nfds, int timeout) | 与select一样，区别在于：   1. 描述描述符等待事件的结构体换成了pollfd   struct pollfd {  int fd; /\* file descriptor \*/  short events; /\* requested events \*/  short revents; /\* returned events \*/  }   1. timeout精度为毫秒，-1表示阻塞 2. 没有监听描述符的最大数量限制 |
| epoll | int epoll\_create(int size)  int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)  int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout) | 创建：返回一个epoll实例，size是epoll初始建议监听数量，但不能为0  管理：增、删、改[op]epoll实例[epfd]监听的描述符[fd]，修改为event  监听：监听epoll实例[epfd]，返回最多maxevents个待处理事件，事件内容为events，设置超时timeout个毫秒，timeout设置-1表示阻塞；如果真是就绪的文件描述符数量大于maxevents，剩余没有处理的事件仍会在下次调用epoll\_wait时返回，这是epoll默认的LT模式 |

区别：

1. epoll和poll没有监听的描述符数量限制；
2. epoll和poll的时间精度不如select和pselect，所以select的另一个用途是Timer，设置监听的描述符全部为NULL，设置Timer的时间间隔为timeout，循环调用select即可实现一个Timer。
3. epoll与上述三个不同之处在于：每次调用select(pselect/poll)都会将监听的描述符和描述符事件拷贝至内核，epoll则只需拷贝一次，需要修改时调用epoll\_ctl进行一次内核拷贝即可；select(pselect/poll)对文件描述符的监听机制都一样，每次调用会轮询所有被监听的描述符是否就绪，一旦描述符队列变大而且idle-connection比较多时效率明显低于epoll（epoll的轮询机制是异步，epoll有一个描述符就绪链表，如果有描述符就绪就将描述符挂到链表上，这样epoll只要检测链表是否为空就可判断有没有描述符就绪）。下面是随着监听描述符集合增大每秒的多路复用调用次数，epoll很稳。



1. 进程
2. 创建进程

cout<<"123\n";

pid\_t p = fork();

if(p < 0){

perror("Create subprocess");

}else if(p == 0){

cout<<"456\n";

}else{

sleep(1);

cout<<"789"<<endl;

}

运行./aa>ccc

重定向文件ccc输出：

123

456

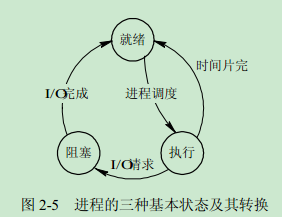
123

789

说明创建一个进程除了会复制地址空间之外还会复制文件描述符，而文件描述符指向的文件却是共享的。终端是行缓存的，当使用文件重定向输出时，缓存没有被\n刷新，所以子进程的STDOUT中含有123的缓存，因此输出两次123。（注：C++中的endl会换行并刷新缓存）。在fork一个进程之前，需要关闭父子进程不必要共享的文件描述符。或者直接使用exec创建新的进程环境(system的实现就是先fork然后在子进程中调用exec)。虽然终端输出、错误输出也会共享，但在程序中添加endl可以刷新缓存（假设cout<<…<<endl是原子操作）。

1. 状态

进程的存在状态，需要说明的是阻塞态的进程不占用CPU的原因是它并不会存在于CPU的调度队列中（连轮询的机会都没有）。



1. 工作目录

每个进程都有一个工作目录 (有时称为当前工作目录。所有相对路径名都从工作目录开始解释。进程可以用chdir(const char\* path)函数更改其工作目录。返回0表示成功，其他则是出错。

1. 守护进程

创建过程：

* 1. fork，创建子进程并让父进程终止，子进程会被init进程托管，处于后台运行状态
  2. setsid，创建新的回话，子进程变成新的会话的会话头和进程组组长，不再有控制终端，和终端无关
  3. fork，子进程的子进程产生，子进程退出，确保将来即使打开终端设备也不会获得控制终端，没有控制终端的会话头打开终端设备会获得控制终端，当子进程退出后，会话头成了子进程的子进程，而他并不是一个会话头，所以不会获得控制终端
  4. 关闭所有文件描述符，关闭所有文件权限掩码

代码：

void init\_daemon(void){

int pid;

int i;

if(pid=fork())

exit(0);//是父进程，结束父进程

else if(pid< 0)

exit(1);//fork失败，退出

//是第一子进程，后台继续执行

setsid();//第一子进程成为新的会话组长和进程组长

//并与控制终端分离

if(pid=fork())

exit(0);//是第一子进程，结束第一子进程

else if(pid< 0)

exit(1);//fork失败，退出

//是第二子进程，继续

//第二子进程不再是会话组长

for(i=0;i< MAXFD;++i)//关闭打开的文件描述符

close(i);

chdir("/");//改变工作目录到/

umask(0);//重设文件创建掩模

return;

}

1. 僵尸进程和孤儿进程

僵尸进程：子进程终止后，父进程没有调用waitpid释放子进程占用的资源。解决方法包括调用waitpid或者把父进程杀死。

孤儿进程：父进程在子进程之前退出，孤儿进程会被init进程收养，然后退出，init调用waitpid

1. IPC（进程间通信方式）https://github.com/549642238/MyRepository/tree/master/IPC

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| IPC方式 | 优点 | 缺点 | 特点 | 接口 | 共同点 |
| signal(信号) | 操作简单，进程自带，开销不大，适用于一些简单的进程间信息交互 | 类型有限，只能发送有限类型的信号，收到信号的进程根据信号类型做出对应的信号处理函数，无法传递更多信息 | 信号像是一种中断，进程收到信号会执行对应的信号处理函数。命令行的kill -9 pid实际上就是给pid进程发送一个SIGKILL信号，对应的默认处理函数是结束进程  适用场景:  当进程间通信的内容简单，例如仅有几种类别，可以根据不同类别相应不同事件时，适合用信号量 | signal:绑定信号和信号处理函数  kill:给对应pid的进程发送某种信号 | 无 |
| pipe(管道) | 开销不大，操作简单，可以传递复杂的数据完成复杂的进程间通信 | 半双工，必须是具有共同祖先的进程间通信，常用于父子进程间的通信 | popen以及协同进程就是利用pipe实现。  适用场景:  虽然pipe是半双工的，但用两个pipe也是可以实现全双工的，但由于pipe对进程的亲缘关系限制，一般在父子进程间用的较多 | pipe:创建管道，返回一个写端和一个读端，一般会关闭其中一个 | 管道和有名管道都是基于文件的，它们可以利用文件操作的API完成IO等操作，也可以使用多路复用。但依赖于文件的缺点是消息以及管道在创建它们的进程退出后会丢失。虽然是基于文件的，但管道还是内核维护的，即数据通信需要在用户态到内核态间拷贝 |
| fifo  （有名管道） | 可以在没有亲缘关系的进程间通信，操作简单，开销不大，可以传递复杂的数据完成复杂的进程间通信 | 半双工 | fifo可用于实现shell中的管道命令  适用场景:  虽然fifo是半双工的，但用两个fifo也是可以实现全双工的，fifo是基于文件的，可以使用多路复用，当进程间通信不要求消息在创建fifo的进程退出后还存在，一般情况都可以使用fifo，毕竟操作简单也能实现复杂数据双向传输 | mkfifo:创建有名管道，但打开时只能选择只读或只写，所以也是半双工 |
| message queue  (消息队列) | 全双工，消息队列可以做到多播，例如server向队列添加了不同ID的消息，对应不同ID的进程取各自的消息，这是pipe，fifo，socket做不到的。 | 内核维护消息队列开销很大，每次读消息、写消息都要进行用户态到内核态的拷贝 | 消息队列以及队列里的消息创建之后会一直存在，直到有进程读取完消息以及将消息队列删除，创建它们的进程退出不会影响消息队列，消息队列由内核维护  适用场景:  广播通信可以考虑消息队列，要求消息在进程退出后依然存在可以考虑消息队列 | msgget: 获取消息队列  msgsnd:向消息队列添加消息  msgrcv:从消息队列取消息  msgctl:删除消息队列，获取消息队列信息 | System V IPC不以文件为基础，不支持系统的文件操作，系统不得不提供新的API来操作它们的描述符，所以也不能用多路复用，好在新的API针对IPC都很有共性，例如都使用IPC\_RMID删除IPC标识 |
| semaphore  (信号量) | 用于进程间的同步和互斥处理的最优选择 | 进程间无法传递复杂的信息 | 信号量创建之后会一直存在，直到有进程销毁它，创建它们的进程退出不会影响信号量，除非指定UNDO标志，信号量由内核维护  适用场景:  需要进程间同步互斥时使用信号量 | semget:获取一个信号量的集合  semctl:设置/获取集合中各个信号量，删除信号量集合  semop：添加或减少信号量 |
| shared memory  (共享内存) | 直接由内存来作为通信介质，是最快的IPC通信手段。使用更灵活，可以通过结构体作为传输数据，也可以是一对多的进程通信达到广播效果，也可以是多对一的通信 | 操作复杂，需要注意多进程对同一共享内存的读写同步问题，需要和信号量配合使用 | 共享内存创建后地址是一块介于堆和栈之间的地址，所以需要将共享内存的地址映射到进程地址空间  适用场景:  要求进程间通信效率极高的情况使用共享内存，一般和信号量配合使用，广播通信也可以使用共享内存，但考虑到同步操作能用消息队列就不用共享内存 | shmget:获取一块共享内存区域  shmat:将共享内存区域映射到进程地址空间  shmdt:将共享内存从当前进程分离，并不是删除共享内存  shmctl:删除共享内存区域或设置/获取共享内存区域信息 |
| socket(套接字) | 可实现不同主机上进程间的通信，基于设备驱动文件的IO操作，可使用IO多路复用 | 开销较大，内核和用户态交互频繁 | 基于文件操作，是一种特殊的文件描述符  适用场景:  一般用于网络通信 | socket:创建套接字  bind:绑定地址和端口  listen:服务器监听  accept:服务器等候客户机连接  read:读数据  write:写数据  connect:连接服务器 | 无 |

1. 线程
2. 线程概念

轻量级的进程，CPU调度执行的最小单位。

1. 线程和进程的区别

线程共享什么？

进程指令、全局数据、堆区、打开的文件、信号处理方法、当前工作目录、用户ID和组ID

线程不共享什么？

线程ID、栈、errorno、程序计数器、信号掩码、优先级

什么情况下使用线程？什么情况使用进程？

进程用来协作完成某件事情，共享某些资源，例如网易云音乐，可能有界面显示线程监听用户操作、后台下载MP3的线程、推荐广告和消息的线程。进程一般是一个独立的应用，进程之间要保证相互独立，不会相互影响。比如专门邮件收发的进程、浏览器的进程、画图的进程。

线程更轻量级，对于处理多路IO事件很在行，比如多路复用+多线程实现服务器。

1. 线程间通信

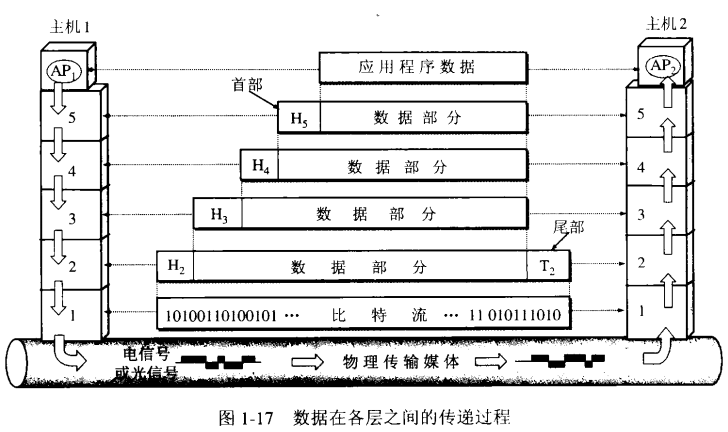
全局变量、堆

互斥：mutex

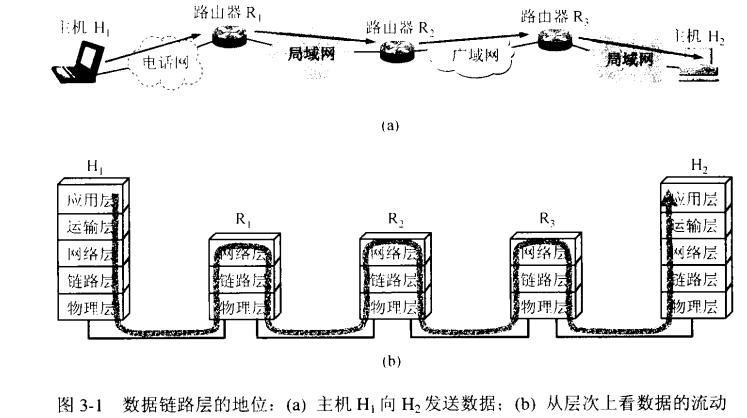
同步：pthread\_cond\_wait/pthread\_cond\_signal

**网络篇**

1. 现实中的网络模型并不是OSI七层模型，而是五层(物理层，数据链路层，网络层，传输层，应用层)，对等层之间的数据传输是抽象的，实际的数据传输链路应该由下面两幅图解释：



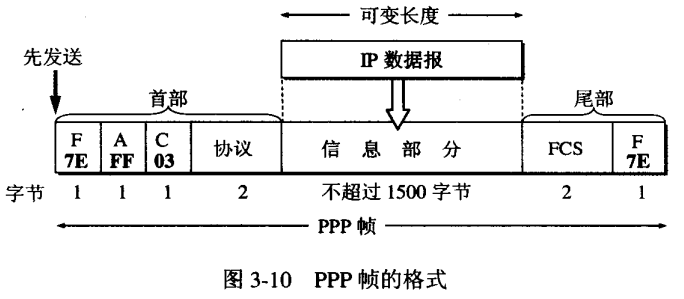
IP数据包 = 传输层数据+传输层首部，到了下一层(数据链路层)，整个IP数据包被认为是上一层的原始数据，所以链路层的数据帧 = 帧首+IP数据包+帧尾。



将网络互联起来的中间设备有：

1. 物理层：集线器、转发器、调制解调器
2. 数据链路层：交换机
3. 网络层：路由器
4. 网络层以上：网关
5. 物理层：将数字信号转换成电信号，通过高低电平+编码（如：差分编码）实现电信号对数据的编码。
6. 数据链路层：封装成帧、差错检验(CRC)、点对点传输(路由器对路由器/主机)

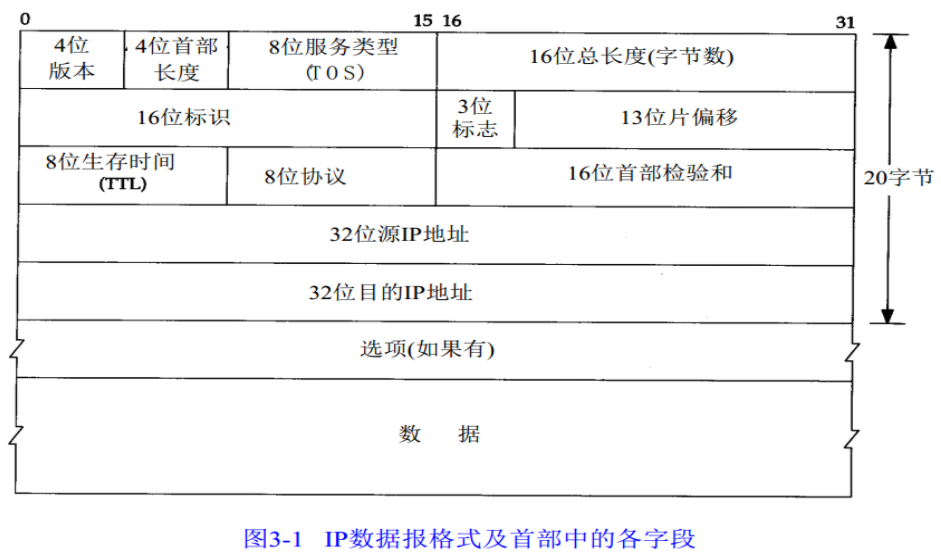
PPP(Point to point protocol)协议中的字节填充：链路层传输的帧有帧首和帧尾一说，如果帧里的数据有和帧首或帧尾重复的数据应当做转义处理。



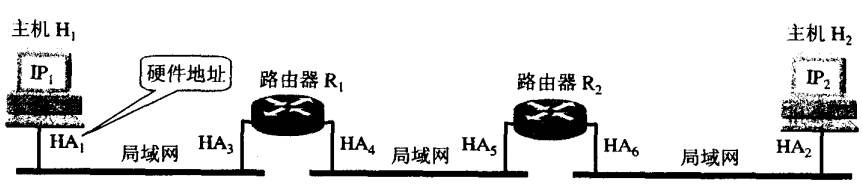
PPP将0x7D定义为转义字符，信息字段中的0x7E转义为两字节(0x7D0x5E)，信息字段中的转义字节(0x7D)转变成(0x7D0x5D)。

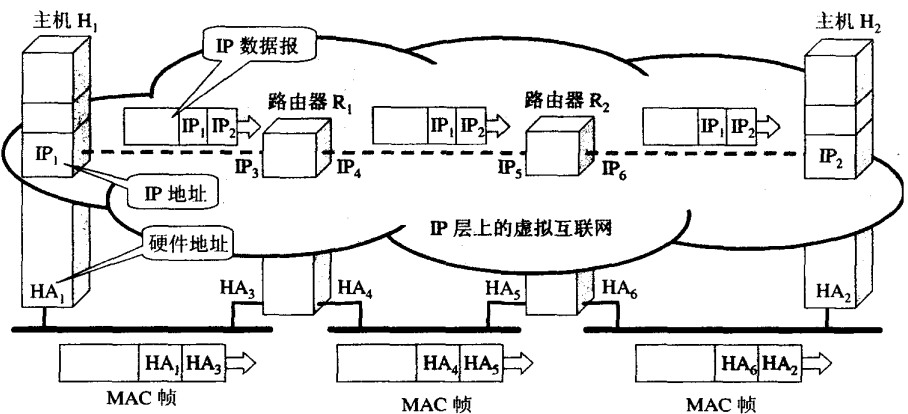
1. 网络层：网络数据包(IP包)的传输，提供路由选择服务

IP数据报



1. 路由选择：





IP地址是抽象的地址，路由器连入不同的网络会获取不同的IP地址。IP地址会放在IP包的首部。

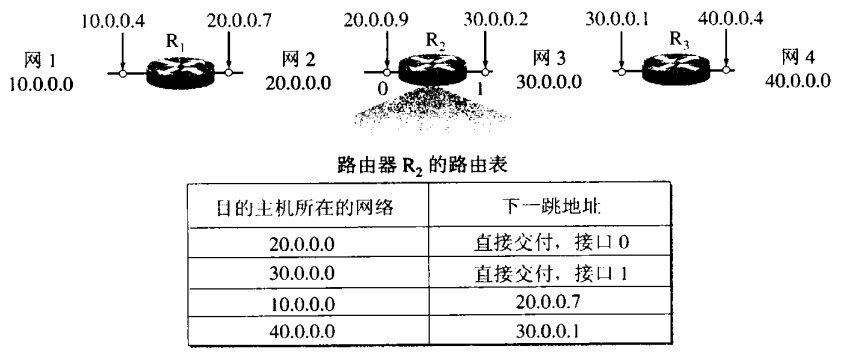
MAC地址是实际的设备ID，对应于一台真实的设备。MAC地址会放在MAC帧的首部，实际的数据转发是根据IP地址对应的实际MAC地址来确定转发对象的，因为在链路层只能看到MAC帧，路由器只知道附近有哪些路由器(以及它们对应的MAC)，路由器必须知道他要将帧转发到哪个路由器上。

路由器如何根据目的IP知道它下一跳要转发给谁（换言之，路由表怎么构建）？

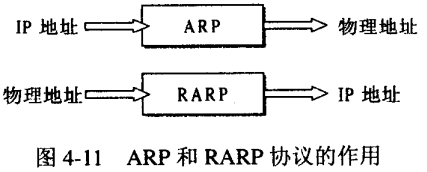
系统启动时，从与本路由器直接相连的各网络地址中，推导出一组初始路由，通过初始路由只能访问相连网上的主机。RIP正是用于路由表的维护和刷新，RIP协议中的路由舒心素散发是距离向量算法。路由选择协议：RIP(内部网关协议)，BGP(外部网关协议)。

分组转发算法进行路由：

* + 1. 从数据报的首部提取目的主机的 IP 地址 D, 得出目的网络地址为 N；
    2. 若网络 N 与此路由器直接相连。则把数据报直接交付目的主机 D。否则是间接交付，运行(c)；
    3. 若路由表中有目的地址为 D 的特定主机路由。则把数据报传送给路由表中所指明的下一跳路由器。否则，运行(d)；
    4. 若路由表中有到达网络 N 的路由，则把数据报传送给路由表指明的下一跳路由器。否则，运行(e)；
    5. 若路由表中有一个默认路由。则把数据报传送给路由表中所指明的默认路由器；否则，运行(f)；
    6. 报告转发分组出错。

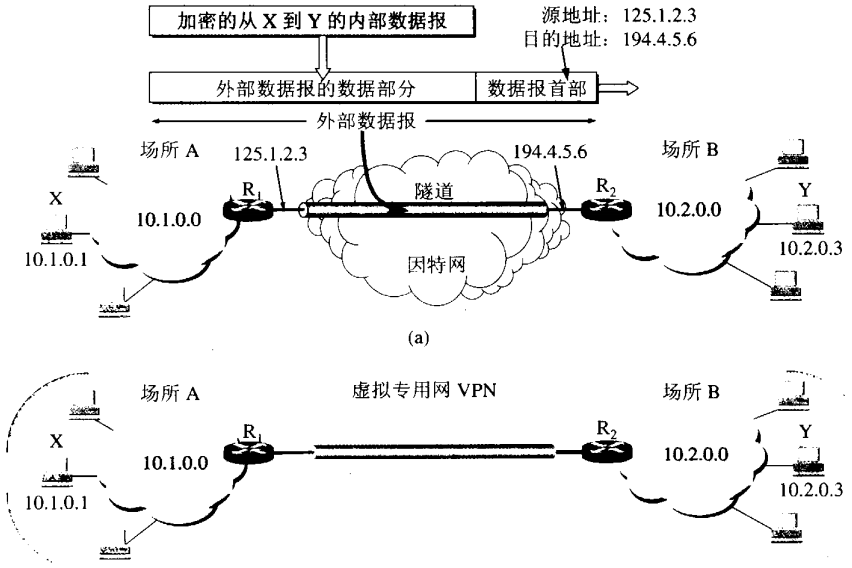


路由器如何知道要转发给下一跳的MAC地址（换言之，他如何根据IP地址转换到对应的MAC地址）？

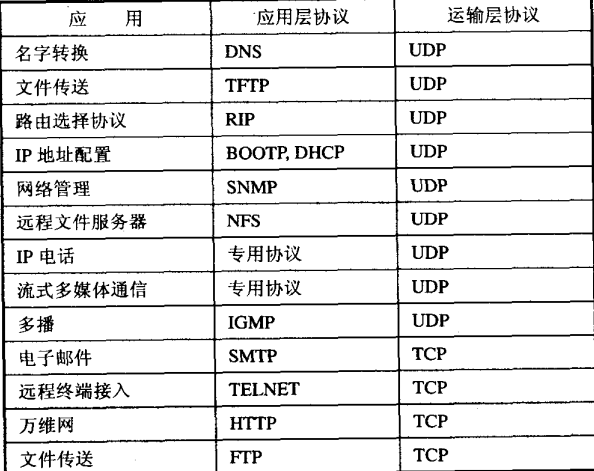


ARP协议：每个主机有一个ARP高速缓存，里面是本局域网各主机和路由器的IP和MAC地址映射表。假设主机A要给某一IP的主机(假如是B)发送数据，A查看高速缓存发现对应IP的MAC是xxx，A将xxx写入MAC帧再转发给对应xxx的设备。如果A在ARP高速缓存无法查到这一IP地址，可能A或B刚加入网络，A运行ARP进程在本地局域网广播“我的IP是192.168.0.80，MAC是00-00-C0-15-AD-18，我想知道IP为192.168.0.66的MAC地址”。其他主机收到此消息查看A的MAC是否在在自己的ARP缓存中存在，如果不存在可能A是新机子，将A的IP和MAC写入缓存；如果存在，但A的IP不是广播里的，说明A可能刚启动换了新的IP，需要在ARP缓存中更新A的IP。B主机收到广播消息发现A要找的IP就是自己，B向A发送回应，A收到B的回应后填写对应IP和MAC地址。

1. ICMP：差错控制（终点不可达、超时、源点抑制、参数问题、重定向），询问(ping是由ICMP实现)。
2. IGMP：多播。
3. VPN：虚拟专用网络。一个机构的两个子网通过Internet互联，每个子网的路由器有真实的IP地址。场所A的主机X要和B的Y通信，R1收到数据发现目的网络要走因特网，需要对数据包加密后传输到R2，因特网作为载体可能包括很多个路由器，但R1和R2好像是一条点对点的通路。VPN需要对应的服务器，路由表应当知道对应的目的地址是我方的并且如何通过因特网达到目的地址。



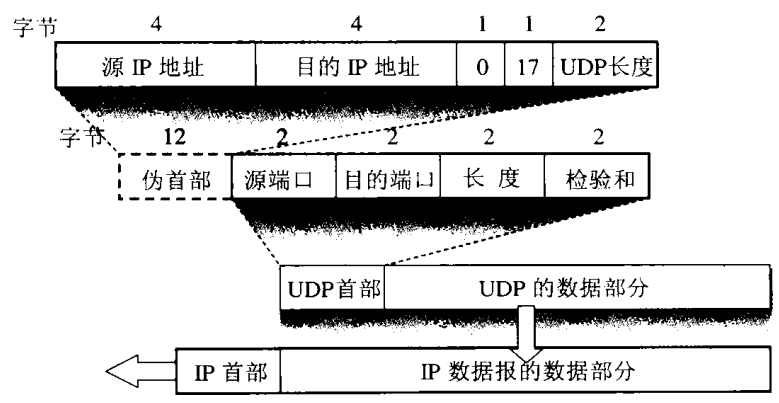
1. 传输层：完成复用和分发功能，网络层保证主机到主机（端到端）的逻辑通信，运输层保证一台主机上的应用程序到另一台主机上的应用程序（端到端）的逻辑通信，任何发往应用进程数据包都要经过传输层（复用），再由传输层分发到各个应用进程（分用）。
   1. UDP传输之前不需要建立连接。TCP传输之前需要建立连接，还要提供确认、流量控制、拥塞控制、差错检验等服务，开销较大。什么时候选用TCP什么时候选用UDP根据网络（或设备）开销和传输可靠性要求来决定。



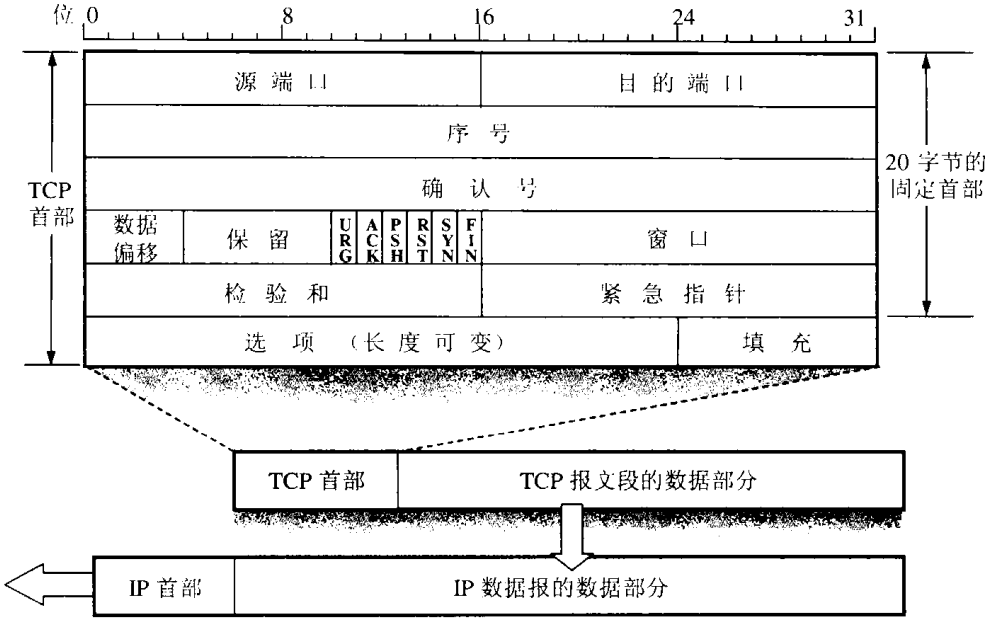
* 1. 端口的由来，我们为什么不直接用进程ID构成IP：ID的地址进行不同主机进程间的通信？
     + 1. 不同操作系统的进程标识符不统一
       2. 一开始通信的时候如何知道要发送给对方主机的进程的对应ID，例如主机A要给B发送电子邮件，A在发送之前如何知道B的电子邮件进程的ID

一个应用进程绑定一个端口号，一般服务器的端口号是public的，客户端向服务器IP：Port发送信息，服务器也会知道客户端的端口号（对应的进程会绑定它），这样从端口IO数据就达到了通信的目的。所以我们一般服务器socket编程会有一个bind的过程，就是把进程和指定的IP：port绑定。

* 1. UDP：伪首部不在网络传输中出现，不会包含在IP报文封装的数据部分，只是用来在传输层临时加上伪首部去做校验和。



* 1. TCP
     + 1. 报文

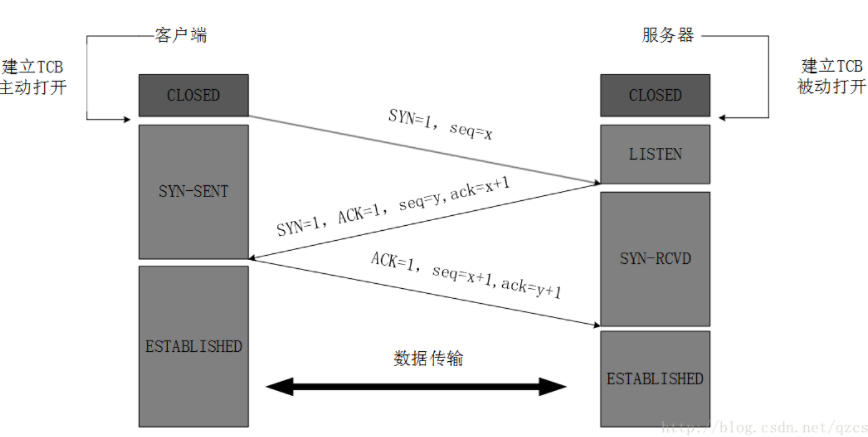


* + - 1. 三次握手（连接）

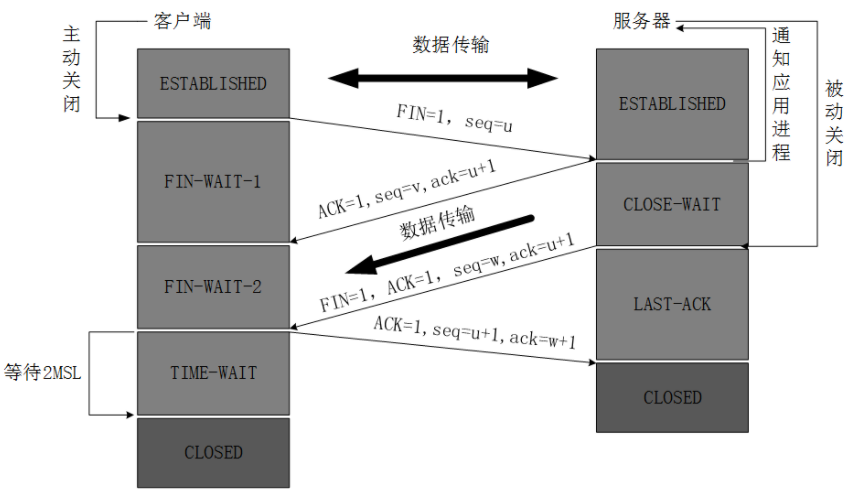
SYN位置1的报文不能携带数据。

三次握手的原因：

如果通过两次握手建立TCP连接，假如第一个Client发向Server的连接请求由于网络延时滞留在网络中，Server没有及时回复ACK给Client，Client会再次发送连接请求给Server，Server收到请求并与Client建立连接，如果Client退出后，Server才收到滞留在网络中的第一次连接请求，两次握手会让Server与一个不存在的Client建立连接，导致不必要的错误和资源浪费。而且这会让DDOS攻击变得更加容易，原本的DDOS攻击需要大量的client向server发送SYN=1的数据包，现在只有两次握手，那么只要一个client循环发送SYN=1的数据包就可以了。



* + - 1. 四次握手（断开）



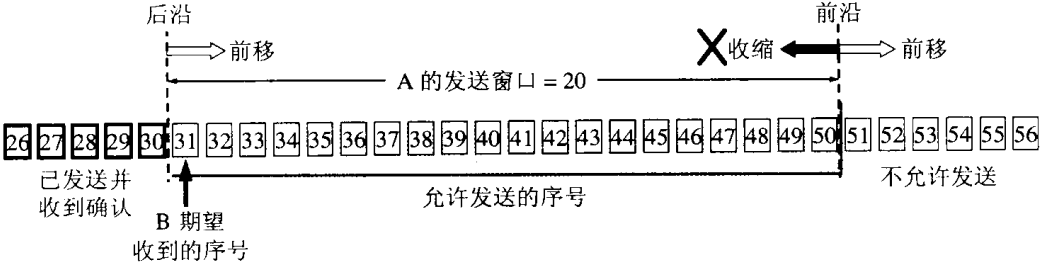
FIN位置1的报文不能携带数据。

客户端在TIME\_WAIT滞留2MS的原因：

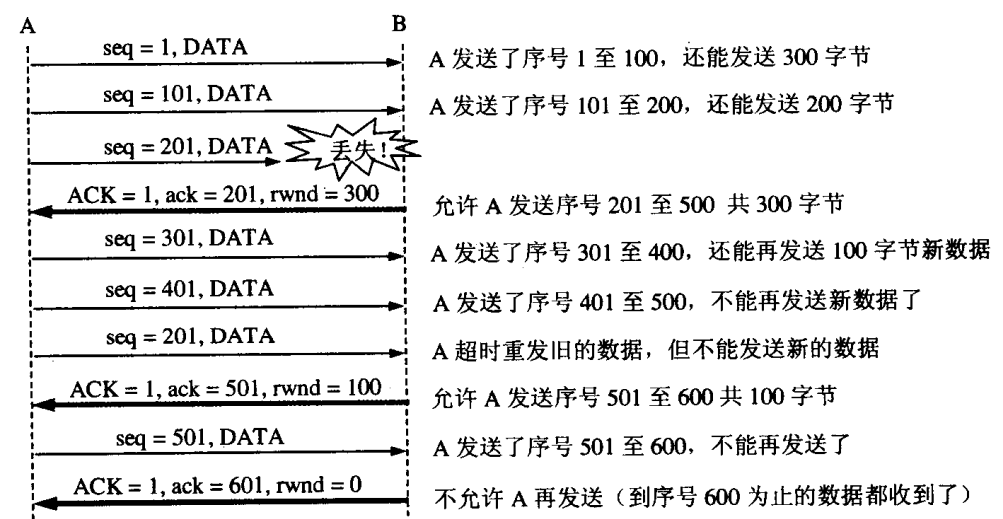
1. 为了保证最后的ACK到达Server，如果Client直接关闭，而ACK没有到达Server，Client已经关闭无法接收Server的重传请求，导致Server无法正常关闭。
2. 保证上次的连接请求报文在网络中全部消失，2MS是一个大于所有数据包在网络层存在的时间，足以让上次连接的数据包全部消失在网络中。如果Client立即关闭，当再次连接到Server后Server可能会收到上次Client发给Server滞留在网络中的数据包。
   * + 1. TCP特点

可靠到达：滑动窗口将整个要传输的数据分为三大段：已经发送并得到ACK确认的数据包、窗口内已经发送还没得到确认和还没来得及发送的数据包、窗口外（还不能发送）未发送的数据包。当发送方每收到一个确认ACK，窗口就向未发送的数据包移动一格。

如果发送方发送了数据，会有一个重传计时器（可能是发送的数据没到达接收方，也可能是接收方的ACK没达到发送方），重传计时器超时会重发没有收到ACK的数据包。重传计时器的超时时间会根据每次ACK的返回时间自适应调整，保证既可以高效传输数据，又不会在网络拥塞时加重网络负担。



流量控制：通知发送方发送数据的速率不要太快，要让接收方来得及接收。接收方通过发送自己的接收窗口大小通知发送方的发送窗口大小不能超过自己。

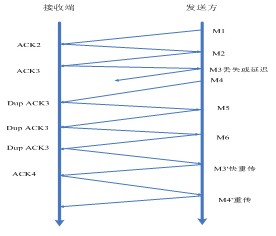


拥塞控制：防止过多数据注入到网络，使网络中的路由器或链路不致过载，是一个全局性问题。慢开始，拥塞窗口大于等于发送窗口，当出现ACK丢失的情况较多时，认为发生拥塞，拥塞窗口调低。

* + - 1. 重传机制

快速重传：

发送端收到来自接收端的N个ACK（附带ack=i），如果有3个或以上的重复ack number，不需要等到TCP重传定时器超时就立即重传相应报文。



重传次数：指数退避型重传时间间隔，发送端没有收到接收端的ACK，或三次ACK的ack的number一样引起了快重传，发送端会间隔1.5s重传数据包，如果还没收到ACK则间隔3s、6s、12s再次重传数据包。每次重传间隔都会基于原来的重传间隔基数（1.5s）呈指数级增长，直到总重传时间达到9min。重传间隔、重传总时间以不同系统实现而区别。

* 1. TCP和UDP的区别
     + 1. TCP通信之前需要建立连接，UDP不需要
       2. TCP是字节流的通信模式，UDP通信是报文模式
       3. TCP保证数据可靠、有序到达，不重复、不漏传。UDP则不保证数据可靠到达
       4. 就效率来讲UDP更高，适合视频流传输，TCP保证可靠性可用做邮件、HTTP
  2. Socket编程

Q1. 多个客户端为什么可以和服务器的同一个端口通信？多个客户端同事发消息为什么在服务端不会受到乱码？

A1. 一个socket描述符是由源地址、源端口、目的地址、目的端口、协议号区分的，只要一个不同socket描述符就是不同的，每个socket描述符对应一个缓冲文件，当服务端多个socket（客户端）同时进行IO时是不会相互影响的，所以服务器不会受到乱码。

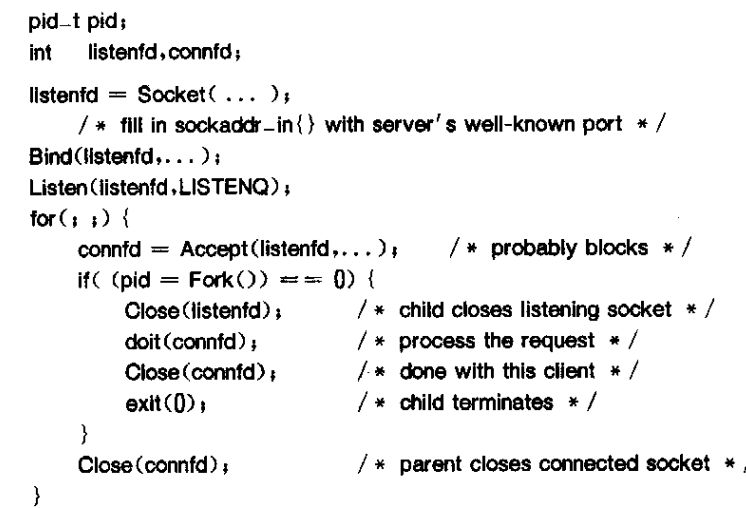
Q2. 为什么要有listen，服务器建立好socket后直接监听执行accept不就可以了吗？

A2. 套接口建立后，系统默认它是一个主动连接状态，默认这个套接口一会儿会connect服务器，函数listen将套接口由主动状态变为被动接受连接请求的状态，参见TCP状态转换图，listen将CLOSED转换成LISTEN态。

listen的第二个参数backlog是等待（就绪）连接的队列最大长度，就是在accept之前（包括正在三次握手连接和三次握手完成但没有被accept调用从队列中拿走）状态的的等候连接服务器的队列中最大的队列长度。

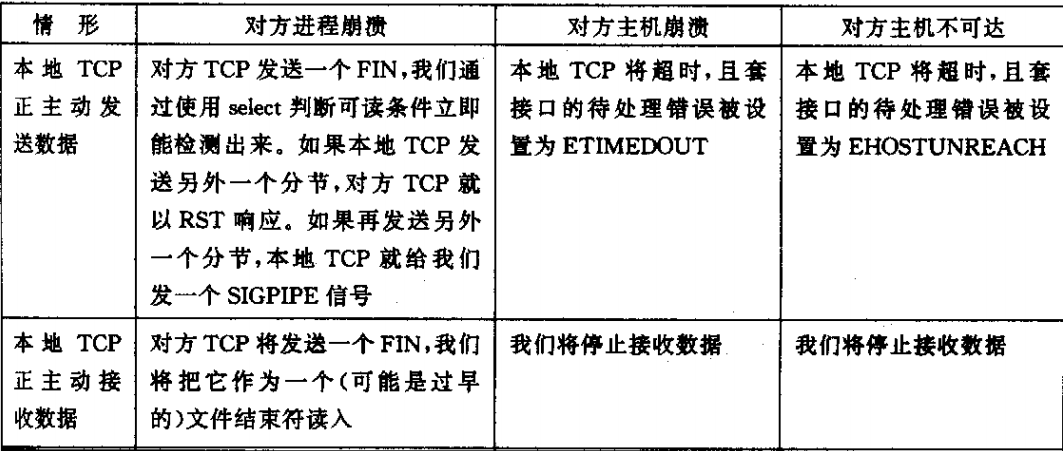
Q3. shutdown和close的区别？

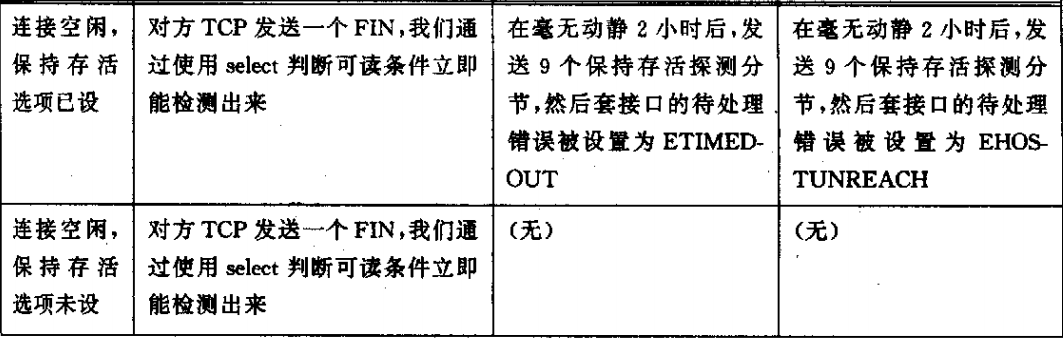
A3. shutdown用于关闭套接口，调用后服务器和客户端的连接断开。close实际上是文件操作，close(sock\_fd)实质上是将sock\_fd访问计数减一，如果用多进程+socket编写的服务器，由于子进程和父进程共享描述符，一个客户端连接对应的sock\_fd会有两次访问计数（子进程和父进程），所以只有子进程关闭描述符并不会真正关闭描述符从而也不会触发关闭的四分节连接终止序列。如果只有子进程或父进程关闭描述符，最终随着程序运行会导致描述符用尽。shutdown还可以控制关闭写通道还是读通道（由于TCP是全双工的），shutdown(socket\_fd, how)，how=SHUT\_RD,SHUT\_WR,SHUT\_RDWR。



* 1. TCP socket服务器客户端在各种情况下的异常断开：

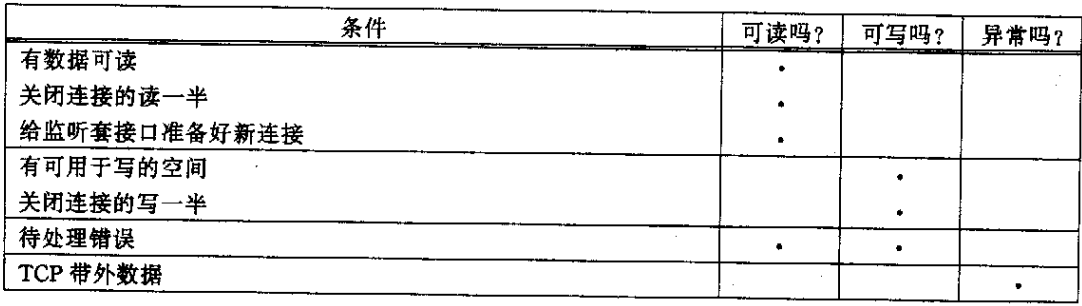
|  |  |
| --- | --- |
| 连接断开情况 | 对应的现象 |
| 客户端连接服务器（完成三次握手后），客户端在服务器accept之前断开连接（可以是close，也可以是SIG\_TERM） | 服务器出于CLOSE\_WAIT状态，服务器accept返回后，如果对返回的套接字描述符操作会返回客户端退出的信息 |
| 客户端和服务器建立连接后，关闭服务器程序 | 客户端通过read可以知道服务器关闭，然后关闭套接字 |
| 客户端和服务器建立连接后，拔掉服务器网线，模拟服务器奔溃 | 如果客户端在服务器奔溃期间没有进行数据IO，服务器重连后仍能进行正常通信；如果客户端在服务器奔溃期间向服务器请求数据，客户端read阻塞，直到超时返回ETIMEDOUT错误, 关闭程序后，客户端处于FIN\_WAIT2，查看服务器状态为FIN\_WAIT1。重启服务器发现Socket bind error:Address already in use |
| 客户端和服务器建立连接后，拔掉客户端网线，模拟客户端奔溃 | 如果客户端在服务器奔溃期间没有进行数据IO，服务器重连后仍能进行正常通信；如果客户端在服务器奔溃期间向服务器请求数据，客户端能在1min内连接网络，仍能进行正常通信，超过1min后，无法再建立连接。关闭程序后，客户端处于FIN\_WAIT2，服务器状态为FIN\_WAIT1。重启服务器发现Socket bind error:Address already in use |
| 服务器关机 | 关机前系统会清理所有进程，关闭描述符，客户端可以知道服务器关闭 |





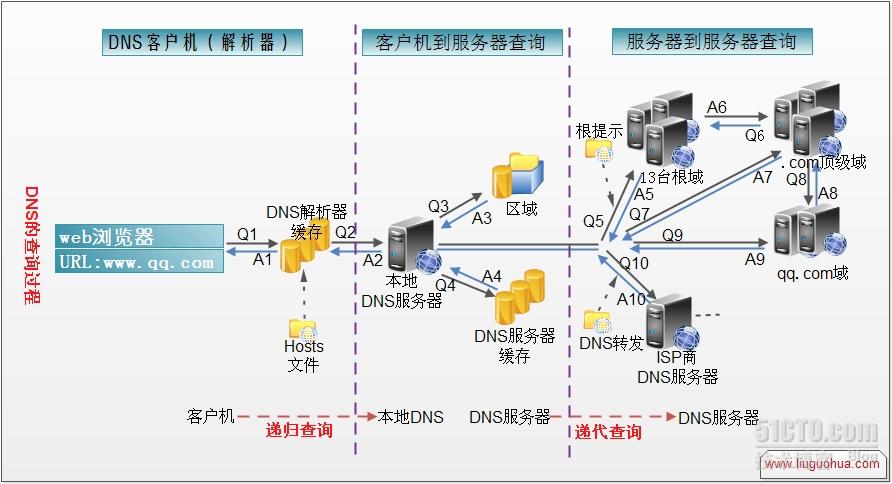
客户机断网重连后还可以继续和服务器通信说明服务器一般都阻塞在read状态，在客户机奔溃时服务器是不知情的，也可能客户机意外断开导致服务器一致保持与已断开的客户机的socket，但这个其实已经是半开连接的状态了。因此SO\_KEEPALIVE在高并发的服务器情况下可以有效地回收客户机奔溃的socket。

* 1. 多路复用就绪的情况

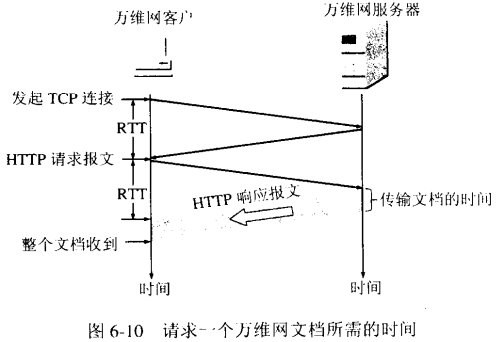


1. 应用层：每个应用层协议都为了解决一类应用问题
   1. HTTP：
      1. 请求过程：①浏览器根据URL（Unique resource locator）(http://tinylink.emnets.org/view/index.html)分析得到对应网站的域名tinylink.emnets.org；②域名服务器返回对应域名的IP地址120.26.201.44；③浏览器与服务器的80端口建立TCP连接；④浏览器发送GET请求获取view/index.html文件；⑤服务器返回index.html；⑥释放TCP连接，浏览器显示index.html文本。

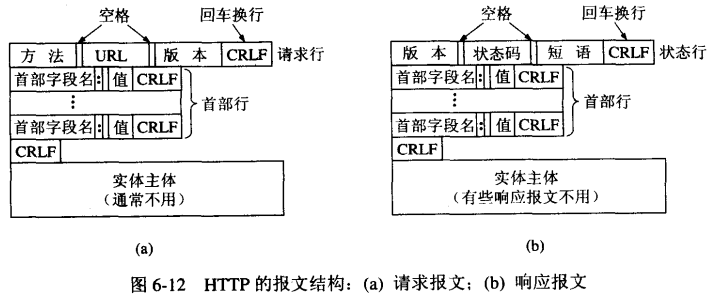
DNS解析过程：



* + 1. 1.1和1.0：HTTP/1.0是短连接，每次HTTP请求建立TCP连接，页面返回释放TCP连接，每次请求都要花费2\*RTT+页面传输时间。HTTP/1.1建立长连接，非流水方式每次请求都花费RTT+页面传输时间，也就是上一次页面确认后才能请求下一个页面；流水方式开始花费RTT时间建立连接，之后可以连续请求（无需等候返回确认）。



* + 1. HTTP报文：



HTTP报文分为请求报文和响应报文：

请求报文包括：请求行、首部行、实体

GET http://10.214.149.119/tinylink/view/index.php?type=view HTTP/1.1 \r\n

Host:10.214.149.119 \r\n

User-Agent:Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36 \r\n // 请求主机系统信息、浏览器版本

Connection:keep-alive \r\n

Accept-Encoding:gzip, deflate \r\n // 客户端接受的编码

Accept-Language:zh-CN,zh;q=0.8 \r\n // 客户端接受的语言

Cookie:Hm\_lvt\_80476b9c0514ea05afdb96912c580c40=1510206249 \r\n

\r\n

GET请求实体没有内容，请求参数都在url中包含

POST http://10.214.149.119/tinylink/tinylink\_module/tinylink\_control.php HTTP/1.1 \r\n

Accept:application/json, text/javascript, \*/\*; q=0.01 \r\n

Accept-Encoding:gzip, deflate \r\n

Accept-Language:zh-CN,zh;q=0.8 \r\n

Connection:keep-alive \r\n

Content-Length:716 \r\n // 请求实体内容长度

Content-Type:multipart/form-data;boundary=----WebKitFormBoundaryAlNBaf04fagMTy7C \r\n

Cookie:Hm\_lvt\_80476b9c0514ea05afdb96912c580c40=1510206249; PHPSESSID=6qmaoss7n6cqs0iibmrq7befg0 \r\n

Host:10.214.149.119 \r\n

User-Agent:Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36 \r\n

\r\n

-----------------------------178337587380872764801987301

Content-Disposition: form-data; name="upload"; filename="test.cpp"

Content-Type: text/x-c++src

TL\_HTTP http;

void setup(){

}

void loop(){

TL\_Humidity.read();

http = TL\_WiFi.fetchHTTP();

http.get("http://10.214.149.119/tinylink/receiveData.php?userid=21621191&nodeid=2&time=xxx&co2=xxx");

TL\_Time.delayMillis(1000);

}

-----------------------------178337587380872764801987301

Content-Disposition: form-data; name="MAX\_FILE\_SIZE"

2097152

-----------------------------178337587380872764801987301

Content-Disposition: form-data; name="type"

execute

-----------------------------178337587380872764801987301--

POST请求实体包含了请求参数，例如上传的文件、字符串参数等

响应报文包括：状态行、首部行、实体

HTTP/1.1 200 OK \r\n

Cache-Control:no-store, no-cache, must-revalidate, post-check=0, pre-check=0 \r\n

Connection:Keep-Alive \r\n

Content-Type:text/html; charset=UTF-8 \r\n// 实体内容格式

Date:Sun, 03 Dec 2017 07:41:02 GMT \r\n

Expires:Thu, 19 Nov 1981 08:52:00 GMT \r\n

Keep-Alive:timeout=5, max=100 \r\n

Pragma:no-cache \r\n

Server:Apache/2.4.23 (Unix) PHP/5.6.24 \r\n // 服务器信息

Transfer-Encoding:chunked \r\n

X-Powered-By:PHP/5.6.24 \r\n

\r\n

<html lang="zh-CN" xmlns="http://www.w3.org/1999/xhtml">

...

响应报文的实体一般返回html或者文档、字符串、图片、视频等

缓存控制header：https://www.cnblogs.com/chenqf/p/6386163.html

* + 1. Cookie和Session

为了解决HTTP每次访问都是无状态而带来的一些局限问题，比如记录用户A每次购买操作引起的购物车状态变化，需要为每位用户维系一段对话。Session位于服务端（文件），Cookie位于浏览器存有session\_id，浏览器发送session\_id到服务器就可以区别出每一位用户的购物车状态。除此，Cookie还可以存放用户名和密码，这样下次用户登录的时候浏览器可以帮你自动填写上次记住的用户名和密码。

* + 1. GET和POST区别

GET是请求数据，POST上传数据。但其实两者都可以上传数据。GET上传的数据长度有限，毕竟数据全附带在URL后面的参数里。POST没有长度限制，将数据放在报文的实体部分里边。

* + 1. HTTP和HTTPS的区别

HTTP明文传输，HTTPS加密传输。HTTPS多了一个SSL层。HTTP默认端口80，HTTPS默认端口443。HTTPS用到CA证书。