Linux操作系统

by [redacted], wszhangxian@126.com，2020年1月20日

1 字符、块、网络设备

背景：Linux中，一切皆文件。设备类型可以分为：字符设备、块设备和网络设备

1. **字符设备**：不支持随机存取，但提供连续数据流，应用程序可以通过字节或字符来读写数据，支持字符寻址。该类设备典型代表是键盘，串口，调制解调器等；
2. **块设备**：不支持字符寻址，但支持随机存取，但读写数据必须以块（通常是`512B`）的倍数进行。典型代表是硬盘、软盘、CD-ROM驱动器和闪存等；
3. **网络设备**：特殊的驱动设备，负责接收和发送帧数据，可能是物理帧，也可能是`ip`数据包。它并不存在于`/dev`下面，而是一个net_device结构，并通过`register_netdev`注册到系统里，可以通过`ifconfig -a`的命令看到。

三者异同点：

- 相同点：支持Linux设备的数据额读写或收发操作；
- 不同点：
  - 字符和块设备挂在在`/dev`下，而网络上设备不是；
  - 字符设备多是外接设备，块设备是系统自带的，网络设备是特殊的`netdriver`设备。

2 `Major`和`Minor Number`

`ls -l`小知识：终端执行`ls -l`

```
drwxr-xr-x  3 xian xian  4096 1月   2 10:50 Desktop
drwxr-xr-x  9 xian xian  4096 12月 25 18:08 Documents
drwxr-xr-x 16 xian xian  4096 1月   2 15:05 Downloads
```

说明：

1. 第一个字段：文件属性字段，例如`drwxr-xr-x`

   1.1 `d`：代表`directory`，即目录，没有则代表普通文件；还有下面

```
l: 链接文件。字母”l”是link(链接)的缩写；
b: 设备文件(block)，置于/dev，是普通文件和程序访问硬件设备的入口，没有大小，只有主设备号和辅设备号；
c: 字符设备文件(character)，置于/dev目录下，一次传输一个字节的设备被称为字符设备；
p: 命令管道文件,与shell编程有关的文件；
s: sock文件,与shell编程有关的文件.
```

1.2 `rwxr-xr-x`：代表文件的权限，`r`是读，`w`是写，`x`是可执行；

2. 第二个字段：占用的节点（硬连接数）

```
-rw-r-r- 1 root root 762 1月 2 18:19 exit
```

例如，`exit`**不是目录**，只有一个；

3. 第三个字段：文件（目录）拥有者
4. 第四个字段：拥有者所在的组
5. 第五个字段：文件占用的空间（以字节为单位）
6. 第六个字段：最近打开查阅的时间
7. 第七个字段：文件或目录名称。

上述和major、minor number有什么联系呢？

答：新打开终端，执行

```
cd /dev
ls -l
crw-r--r--  1 root root  10, 235   1月 2 15:35 autofs
brw-rw----  1 root disk   7,    0  1月 2 15:35 loop0
lrwxrwxrwx  1 root root  11        1月 2 15:35 core -> /proc/kcore
```

问题: major number和minor number是什么？

答：主设备号，10,7,11都是major number；235,0是辅设备号，即minor number，其他略。

major number用途：表示不同的设备类型，用来识别设备的驱动，每增加一个驱动就要添加一个major number，例如

```
crw-rw-rw- 1 root   root    1, 3   Feb 23 1999  null
crw------- 1 root   root   10, 1   Feb 23 1999  psaux
crw------- 1 rubini tty     4, 1   Aug 16 22:22 tty1
crw-rw-rw- 1 root   dialout 4, 64  Jun 30 11:19 ttyS0
crw-rw-rw- 1 root   dialout 4, 65  Aug 16 00:00 ttyS1
crw------- 1 root   sys     7, 1   Feb 23 1999  vcs1
crw------- 1 root   sys     7, 129 Feb 23 1999  vcsa1
crw-rw-rw- 1 root   root    1. 5   Feb 23 1999  zero
```

说明：`/dev/null`和`/dev/zero`使用驱动1，`/dev/psaux`使用驱动10，Linux系统开启的时候就指派好了不同驱动对应不同设备。

minor number用途：表示一个设备的不同分区。根据指定的major number，driver会区分其下面控制的不同辅助设备，它们的设备号即minor number。

3 设备驱动的原理

背景：Linux内核在2.4版本之后引入了设备文件系统`devfs`（device file system），使设备驱动管理很容易。但由于内核版本不兼容问题，下面的知识点建立在没有`devfs`。

`/dev`目录：表示该目录下的设备文件是外设的。

第一步：加载驱动

当Linux添加驱动时，必须分配一个主设备号给它，会调用到`register_chrdev`函数例如

```
#include<linux/fs.h>  //使用到的头文件
int register_chrdev(unsigned int major_number, const char* name,
struct file_operations *fops)
```

问题：如何选择major number呢？

答：如果是已有的major_number，`register_chrdev`函数返回0；如果在函数输入里设置major_number = 0，例如

```
int register_chrdev(0, scull0, &scull_fops);
```

则系统随机分配一个正整数作为major number。如果随机分配的是个负数，则说明代码有问题或分配失败。

```
result = register_chrdev(scull_major, "scull", &scull_fops);
if (result < 0) {
 printk(KERN_WARNING "scull: can't get major %d\n",scull_major);
 return result;
}
/* dynamic */
if (scull_major == 0)
 scull_major = result;
```

- major number是需要提供的；
- name是设备名称，位于`/proc/devices`；
- fops是 a pointer to an array of function pointers，指向函数指针的数组，数组的每个元素存放驱动的进入的entry points.

一旦驱动在内核表kernel table注册成功，它就会被分配一个主设备号，用户命名的驱动名称会被写入`/dev`目录，并与主设备号关联。例如

```
# 创建设备号
mknode /dev/scull0 c 254 0
# 删除设备号
rm /dev/scull0
```

说明：使用`mknode`创建一个名为`scull0`的节点，位于`/dev`目录，该节点属于字符设备，主设备号是254，辅设备号是0，范围是0~255。

4 数据校验原理

背景：在网络传输数据过程中，发送端的数据第一步要经过服务器编码处理，然后发送给接收端，接收端要根据相应的规则解码，从而得到原始数据。但因为网络干扰，数据损坏等因素，如何保证发送端和接收端的数据一致？因此引入校验原理。

4.1 `CRC`校验

定义：`CRC`是 Cyclic Redundancy Check的缩写[2]，用来检测数据通信传输过程中，数据是否发生变化（错误）。由于它在发送端和接收端都有一个生成多项式Generator Polynomial，因此`CRC`也被称为多项式编码方法，目前最流行的。

`CRC`校验原理：步骤如下[1]

1. 先选择生成多项式Generator Polynomial：
    - 标准多项式：例如IBM公司或ISO使用的规则，生成多项式是

$$g(x) = x^{16} + x^{15} + x^2 + 1$$

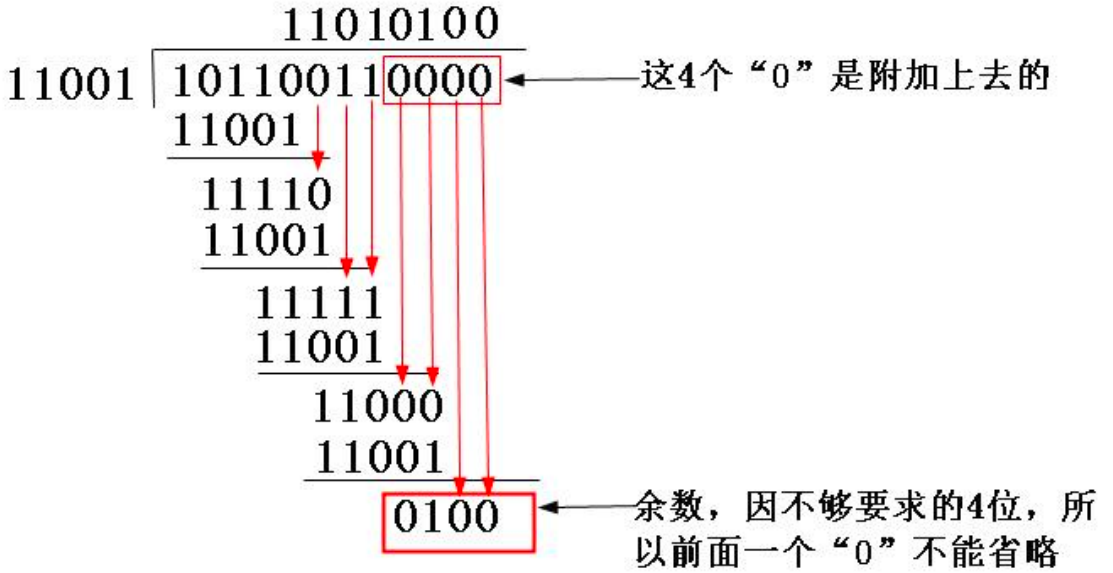
代表的key值是：11000000000000101，就是编解码要用到的除数，其二进制bit串的长度用k来表示，显然k=17。注意**最高位和最低位必须是1**，例如x^{16} 和多项式中的1 。此外多项式必须有加1项；

- 随机多项式：

$$g(x) = x^3 + x + 1$$

代表的key值是：1011（对应与x的指数），k=4

2. 计算校验码：原始数据帧末尾添加（$k-1$）个0。假设数据帧长度为 m，则新的数据帧长度是 $(m+k-1)$，用新数据帧除以key值，所得的余数是帧校验序列Frame Check Sequence。且余数的位数一定要比key值的位数少一位。计算规则见下，相同为0，不同为1（0和1是1,1和1是0）



3. 发送接收数据：将校验码附加在原始数据末端，新生成的数据即编码后要发送的数据。然后接收端按照上述规则，将收到的数据除以相同key值，如果余数为0，说明OK。否则说明数据传输错误。

4.2 奇偶校验

定义：采用奇校验[3]，需要在原始数据末尾加一个0，然后计算二进制数 中的1的个数m_1；采用偶数校验n_1，末尾加一个1，计算二进制数中的1的个数。接收方接收到数据后，如果是奇校验，计算1的个数m_2；偶校验计算1的个数n_2，对比$m_1 = m_2$或 $n_1 = n_2$即可。如果不相等，则说明数据传输错误。

优点：原理和实现简单；

缺点：准确率低，准确率50%，不适合高速通信。因为每发送一次，都要附加一次，操作过于频繁。

示例：原始数据是 $0\text{x}1\text{a}$，转为二进制为00011010

1. 奇校验：

    - 末尾加0：000110100
    - 计算奇数个数：3
2. 偶校验：
    - 末尾加1：000110101
    - 计算偶数个数：4

4.3 累加和校验

定义：在原始数据添加一个校验码，该校验码是原始数据中几个数据的累加和[3]。

特点：对于简单数据一次性传输校验方便，准确率高。但对复杂类型数据检错能力一般。

示例：

- 原始数据：6，23，4
- 校验码：$6+23+4=33$
- 新数据包：6，23，4，33

接收方收到全部数据后对前三个数据进行同样的累加计算，如果累加和与最后一个字节相同就认为传输的数据没有错误。

5 `uint8_t,uint16_t`数据类型

背景：C语言有int, float, double, char, long, short等6中类型，归纳为

1. 整型：int, short int, long int
2. 浮点型: float, double
3. 字符型：char

而`uint8_t, uint16_t, uint32_t, uint64_t`都有`_t`，表示它们是typedef定义的[4]。typedef的功能就是定义关键字或别名。

使用原因：方便代码的维护。例如C中没有`bool`型，一个软件中一个程序员使用int，另一个使用short，会混乱。最好用一个typedef来定义一个统一的`bool`。

问题1：数据原型是什么？

答：去掉__即原型。

```
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;
```

问题2：如何打印这些数据呢？

答：`printf`

```
1) uint16_t %hu
2) uint32_t %u
3) uint64_t %llu
```

对于`uint8_t`，用unsigned转换，然后`cout`

```
uint8_t data = 0x11;
cout << (unsigned)data << endl
```

问题3：他们的范围是多少？

答：**整型范围**

```
- Int8 - [-128 : 127]
- Int16 - [-32768 : 32767]
- Int32 - [-2147483648 : 2147483647]
- Int64 - [-9223372036854775808 : 9223372036854775807]
```

无符号整型范围

```
- UInt8 - [0 : 255]
- UInt16 - [0 : 65535]
- UInt32 - [0 : 4294967295]
- UInt64 - [0 : 18446744073709551615]
```

6 `8/16`进制和`10`进制转换

【1】十六进制

- 数字：**0-9，A-F**，
- 与10进制的关系是：0-9 对应 0-9；A-F对应10-15；
- 字母不区分大小写；

【2】十进制

- 数字：是0-9，逢10进1；

【3】八进制

- 由 0-7组成，逢8进1；

【4】二进制

- 由 0-1组成，逢2进1；

二进制转十进制：略；

八进制转十进制：假设八进制数是1507

$$7 \cdot 8^0 + 0 \cdot 8^1 + 5 \cdot 8^2 + 1 \cdot 8^3 = 839$$

十六进制转十进制：假设十六进制数是$2\mathrm{AF5}$

$$\begin{aligned} 2\mathrm{AF5} &= 5 \cdot 16^0 + \mathrm{A} \cdot 16^1 + \mathrm{F} \cdot 16^2 + 2 \cdot 16^3 \\ &= 5 \cdot 16^0 + 10 \cdot 16^1 + 15 \cdot 16^2 + 2 \cdot 16^3 \\ &= 10997 \end{aligned}$$

假设十六进制数是$0\mathrm{x}11$，$0\mathrm{x}$可忽略，则

$$\begin{aligned} 0\mathrm{x}11 &= 1 \cdot 16^0 + 1 \cdot 16^1 \\ &= 17 \end{aligned}$$

假设是十六进制数是$0\mathrm{x}5\mathrm{b}$

$$\begin{aligned} 0\mathrm{x}5\mathrm{b} &= \mathrm{b} \cdot 16^0 + 5 \cdot 16^1 \\ &= 11 \cdot 16^0 + 5 \cdot 16^1 \\ &= 91 \end{aligned}$$

7 移位和`MSB,LSB`

`MSB`定义：Most Significant Bit，高位；

`LSB`定义：Least Significant Bit，低位；

原理：关键是看进制，一般是二进制数的移位。

定义：

- $<<$：左移
- $>>$：右移

示例1：左移，高位删除2位，低位补齐。即把最左边的0和1删除，最右边补0

```
,# 移动前
107 = 0110 1011 <<2
# 移动后
172 = 1010 1100
```

示例2：右移，低位删除2位，高位补齐。即把最右边的0和1删除，最左边补0

```
# 移动前
107 = 0110 1011 >>2
# 移动后
26 = 0001 1010
```

8 异或 ^ 操作

定义：对二进制数，每个位置进行位运算，相同取 0，不相同取 1。

示例1：^操作

```
0100 ^ 0010 = 0110
```

示例2：两个数的交换

```
void Swap(int& a, int& b){
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}

int main(){
    int a = 5;
    int b = 7;
    Swap(a, b);
    printf("a : %d, b : %d\n\n", a, b);
    return 0;
}
```

9 size_t与int区别

区别：size_t在32位上4字节，64位上8字节，不同架构进行编译时需要注意这个问题。而int在不同架构下都是4字节；且int为带符号数，size_t为无符号数；

何时使用size_t？

答：一般多见于Linux底层通信相关的代码，就是用Int既有可能浪费，又有可能范围不够大的情况下。

10 父子进程通信

10.1 `SIGUSR1`和`SIGUSR2`

功能：Linux下利用kill命令，从而使注册的信号句柄运行，实现分支进程的控制。

`SIGUSR1`： 默认处理后进程终止 。
`SIGUSR2`： 同上。

示例：

```
void handler(int signo) {
  switch (signo) {
    case SIGUSR1:  //处理信号 SIGUSR1
      printf("SIGUSR1 received...\n");
      break;
    case SIGUSR2:  //处理信号 SIGUSR2
      printf("SIGUSR2 received...\n");
      break;
    default:
      printf("Other signal...\n");
      break;
  }
}
int main(int argc, char **argv) {
  sigset(SIGUSR1, handler);
  sigset(SIGUSR2, handler);
  printf("Process_pid=[%d]\n", getpid());
  while (1){};
  return 0;
}
```

编译通过后，执行

```
./main &      //少了&符号，会一直卡在main函数的主进程
# 显示
➜  build ./main &
[15] 6833     //[15]代表PID，即进程ID
➜  build Process_pid=[6833]
              //主线程等待用户输入kill命令，激活信号句柄
```

执行

```
kill -SIGUSR1 6833
# 显示
SIGUSR1 received...
```

如果激活另一个case，则执行

```
kill -SIGUSR2 6833
# 显示
SIGUSR1 received...
```

说明：

- [15]以前的`PID`都可以用，哪怕后面有新生的`PID`；
- 一次只能kill掉一个，这样才能实现进程切换；
- 分支进程走完后，父子进程终止。

10.2 Linux的`sigevent`结构

背景：假设系统有一模块要频繁获取系统时间，使用Linux内置的函数开销过大。如果对精度要求不高（如毫秒级），可以用`signal`函数配合`timer_settime`函数来实现个简易的全局时钟。但`SIGALRM`的中断信号会终止sleep（因sleep就是用`SIGALRM`信号量实现的），导致进程控制不可靠会崩溃。

解决：利用`POSIX`内置的定时器：timer_create()(创建)、`timer_settime()`(初始化)以及 timer_delete(销毁)，将自己的时间信号处理函数用timer_create注册为`SIGUSR2`，就不会中断sleep了。

示例：

```
void handler() {
  time_t t;      //初始化t，单位秒
  char p[32];    //初始化数组，用来接收 timeptr的所有信息，例如本例的年-月-日-小时-分钟-秒
  time(&t);      //时间精度限制为秒级
  //strftime功能：用于格式化输出
  //strftime(char* ptr, size_t size_of_buf, const char* format, const struct tm* timeptr);
  strftime(p, sizeof(p), "%Y-%m-%d %H:%M:%S", localtime(&t));
  printf("@zhangxian -----> date: %s \n", p);
}

int main(int argc, char *argv[]) {
```

```
  int ret;               // return code
  timer_t timer;         //声明一个定时器
  struct sigevent evp;   //信号事件，用来设置定时器到期时的通知方式和处理方式
  struct timespec spec;           //时间声明
  struct itimerspec time_value;   //时间间隔

  evp.sigev_value.sival_ptr = &timer;  //指针：指向时钟产生信号的值，即sigev_value
  /*定时器到期时，会产生一个信号*/
  // sigev_notify:定时器到期后通知
  //(1)SIGEV_NONE:什么也不做；
  //(2)SIGEV_SIGNAL:内核将sigev_signo指定的信号传送给进程，si_value会被设定为sigev_value的值
  //(3)SIGEV_THREAD:以sigev_notification_attributes为属性创建线程，
  //地址为sigev_notify_function，  传入sigev_value作为一个参数
  evp.sigev_notify = SIGEV_SIGNAL;       //定时器到期，则发送SIGEV_SIGNAL给内核
  evp.sigev_signo = SIGUSR1;             //当SIGEV_SIGNAL被接收时，将处理该信号码注册的hander函数
  signal(SIGUSR1, (__sighandler_t)handler);

  /*时钟源选CLOCK_MONOTONIC主要是考虑到系统的实时时钟可能会在
  程序运行过程中更改，所以存在一定的不确定性，而CLOCK_MONOTONIC
  则不会，较为稳定*/
  ret = timer_create(CLOCK_MONOTONIC, &evp, &timer);  //返回0，OK；其他错误
  if (ret) perror("timer_create");         //检查时钟是否创建成功
  time_value.it_interval.tv_sec = 2;       //时间间隔单位：秒
  time_value.it_interval.tv_nsec = 0;      //时间间隔单位：nanoseconds
  clock_gettime(CLOCK_MONOTONIC, &spec);  //程序启动到执行到当前的时间
  time_value.it_value.tv_sec = spec.tv_sec + 5; //5秒后启动
  time_value.it_value.tv_nsec = spec.tv_nsec + 0; //默认设置

  ret = timer_settime(timer, CLOCK_MONOTONIC, &time_value, NULL);
//0表示OK; -1表示error
  if (ret) perror("timer_settime");
  while (1) {
    printf("@zhangxian -----> main loop \n");
    sleep(1);
  }
}
```

说明：

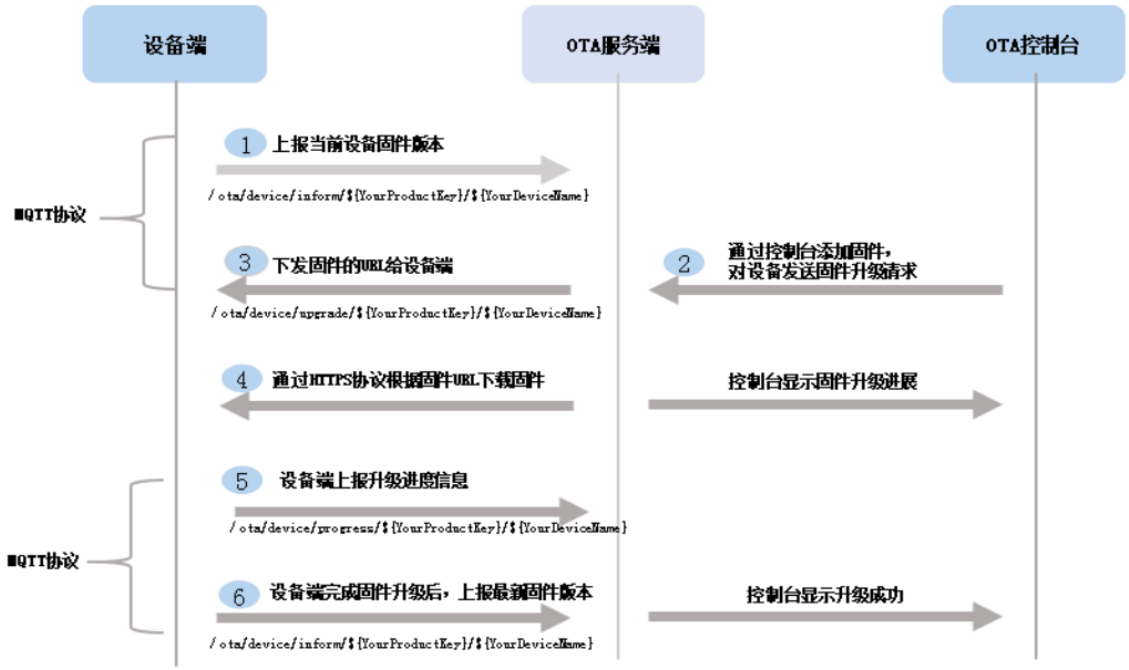
- `timer_settime`：启动/停止或重置定时器[5]
- `timer_gettime`：获得定时器的到期时间和间隔[6]
- `timer_create`：创建定时器，返回的timer id 在调用进程中唯一, 创建后的timer是停止(disarmed)状态
- `strftime`：用于定时器格式输出，参考[7]

11 固件升级流程

`OTA`定义：Over-the-Air Technology即空中下载技术。物联网平台支持`OTA`方式进行设备固件升级。

OTA固件升级流程

MQTT协议下固件升级流程如下图所示。



12 串口通信

定义[9]：数据传输以字节为主，一个字节8个位。拿一个并行通信举例，需要8根线，每根线代表一个位，每根线每次传输一个字节。而串口通信只有一根线传输，一次只传一个位，传一个字节就需要传8次。

12.1 通信方式

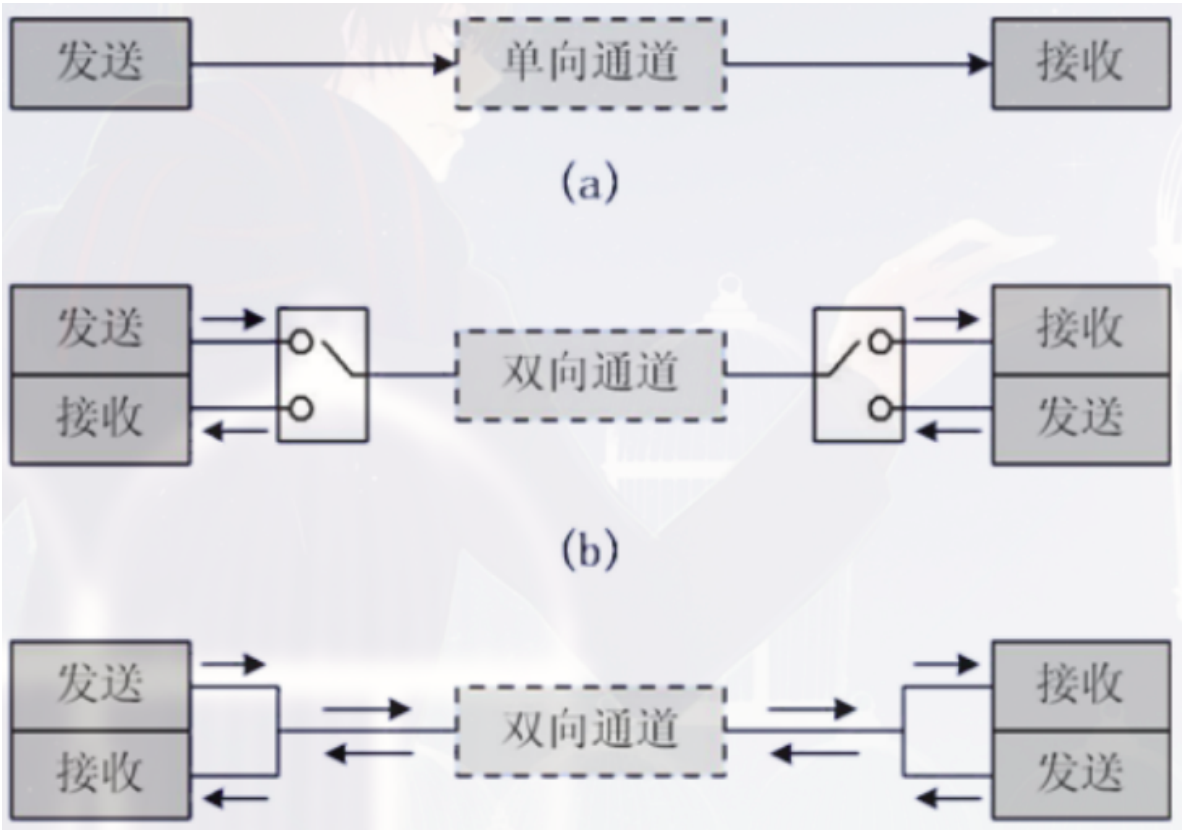
分类	并行通信	串口通信
传输原理	数据各个位同时传输	数据按位顺序传输
优点	速度快	占用引脚资源少
缺点	占用引脚资源多	速度慢

12.2 通信分类

【按照数据传送方向划分】

1. 单工：数据传输只支持数据在一个方向上传输；
2. 半双工：允许数据在两个方向上传输。但某一时刻，只允许数据在一个方向上传输，它实际上是一种切换方向的单工通信；它不需要独立的接收端和发送端，两者可以合并一起使用一个端口。
3. 全双工：允许数据同时在两个方向上传输，需要独立的接收端和发送端。

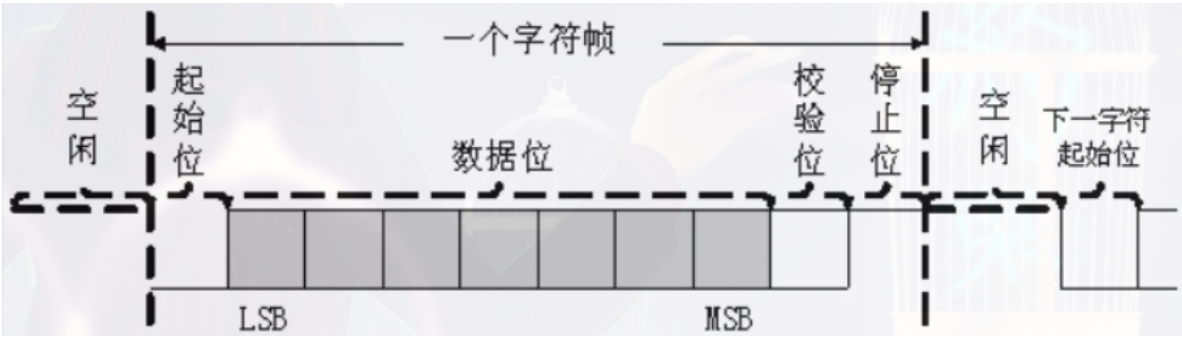
示意图：串口通信按传送方向划分为三种



【按照通信方式划分】

1. 同步通信：带时钟同步信号传输。比如：SPI，IIC通信接口。
2. 异步通信：不带时钟同步信号。比如：UART(通用异步收发器)，单总线。

示意图：异步通信

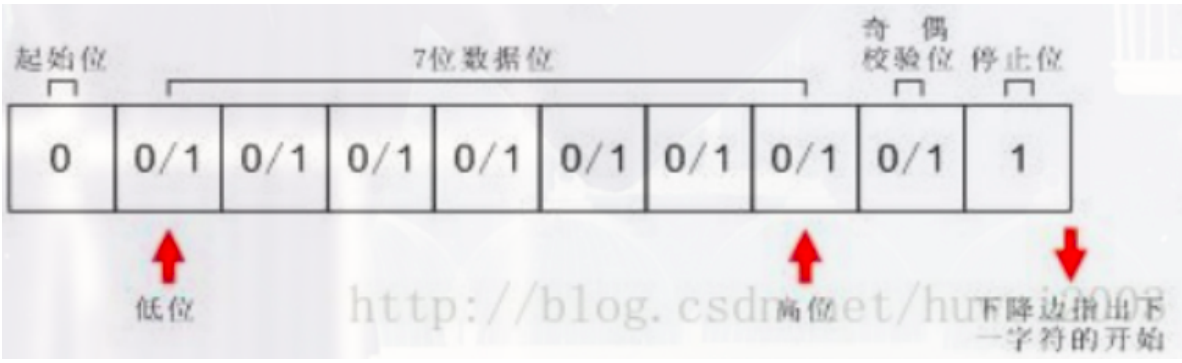


异步通信的两个关键：

- 数据单元——帧，它是双方约定好的数据格式；
- 波特率，它决定了‘帧’里每一位的时间长度。

异步通信的特点：不要求收发双方时钟的严格一致，允许误差大，实现容易，设备开销小，但每个字符要附加2～3位用于起止位，各帧之间还有间隔，因此传输效率不高。

串口通信数据格式：起始位+数据位+校验位+停止位。

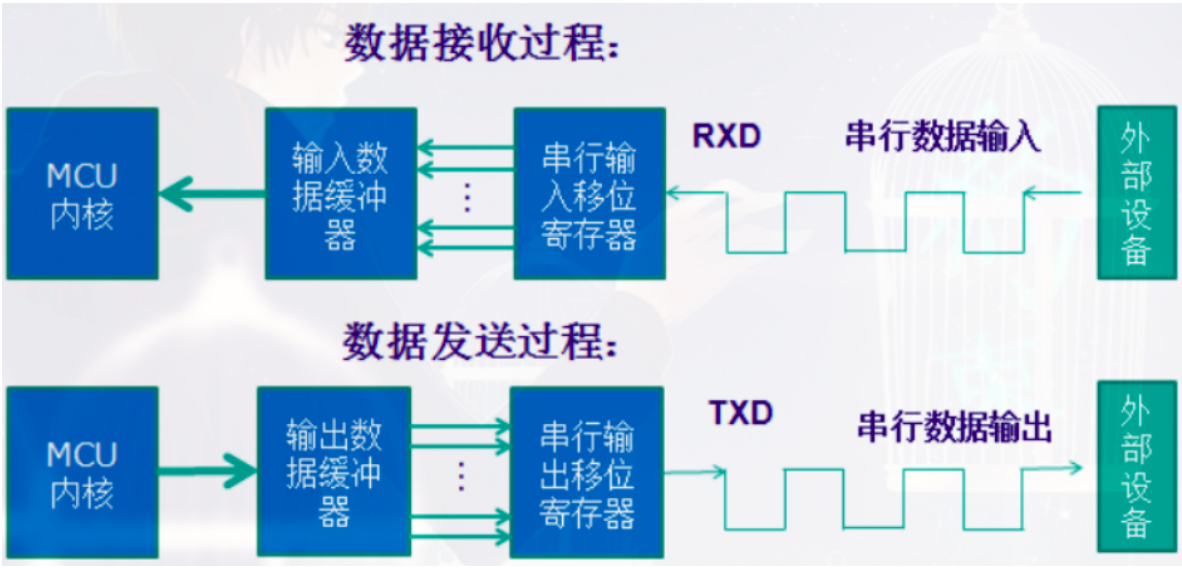


12.3 数据格式参数介绍

1. **波特率**：衡量符号传输速率的参数。指信号被调制以后在单位时间内的变化。如每秒钟传送240个字符，而每个字符格式包含10位（1个起始位，1个停止位，8个数据位），这时的波特率为240Bd；
2. **数据位**：要送的实际数据参数。假设每个数据包使用7位数据。每个包指一个字节，包括开始/停止位，数据位和校验位。**由于实际数据位取决于通信协议的选取，术语“包”指任何通信的情况**；
3. **停止位**：表示单个包的最后一位。典型的值为1，1.5和2位。由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能两台设备间有小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供计算机校正时钟同步时钟同步的机会；
4. **校验位**：检查传输过程中，数据是否舛错，可有可无，但一般都有。

12.4 数据通信过程

见下图



12.5 串口通信协议

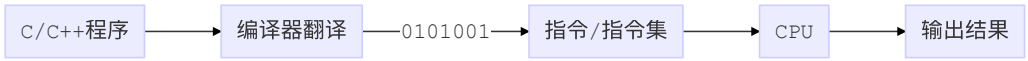
常用的协议包括RS-232、RS-422和RS-485，还有uart协议。232、422、485的数字越大，代表功能越多，对数据线的要求越高，也越贵。

13 Linux x86/ARM区别

指令：是指挥机器工作的指示和命令，程序就是一系列按一定顺序排列的指令，执行程序的过程就是计算机的工作过程（即编译器将程序解释为机器语言）。

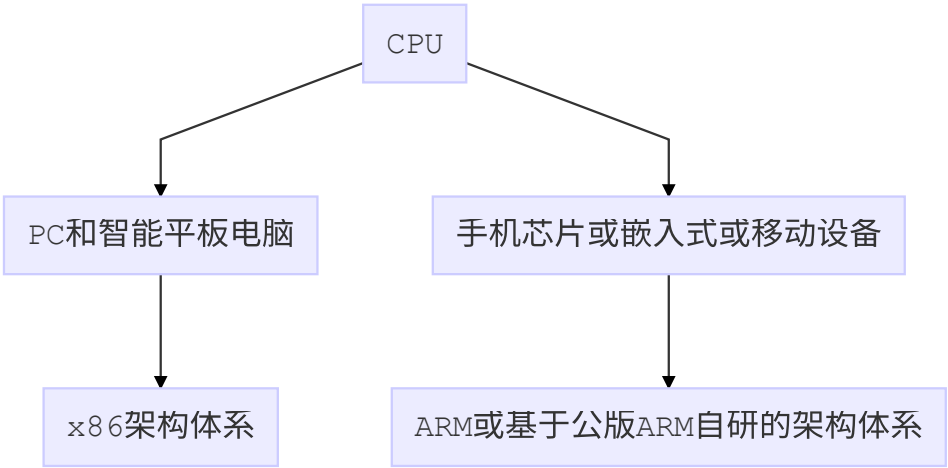
指令集：CPU中用来计算和控制计算机系统的一套指令的集合。

CPU如何工作：早期计算机编程人员将计算机指令转换成1010二进制程序，再将其在纸带上用孔来表示，打孔的纸带输入到计算机之后，计算机才能计算。纸带上的二进制程序就是机器语言，这些孔就是告诉计算机，做读数据、加减乘除、或者逻辑运算，而纸带上的输入则是“孔”和“非孔”，也就“断电”和“通电”，反应在晶体管上的状态就是闭与开。如果晶体管是一个灯泡，表现出来就是一闪一灭。由于机器语言太过于难懂，科学家发明了汇编语言来代替它。这编译器相当于一个翻译机，它根据设定好的规则，将汇编语言一条条转化为二进制语言。那么问题来了，编译器所用的规则，是由设计计算机（现在归结为CPU）的人规定的，用汇编语言的工程师，就必须按照他给的规则来编，否则出来的结果就不知道是什么东西了。因此，设计**CPU的人给出的编写规则就可以说是指令，规则的集合就是指令集**。而设计CPU的厂商有多家，这样就产生了不同的指令集，比如intel阵营的8086指令集，ARM阵营的RISC指令集。不同阵营，其对应的汇编语言也是不一样的。虽然汇编语言接近人类语言，但还是比较难懂，为了解决跨平台运行程序的问题，科学家又发明了高级语言，例如如C/C++/VB/PHP/Java等等，用它们编写的程序要么在相应系统的运行环境中直接运行，要么经过编译打包出对应的程序就可以了。 这样就解决了汇编语言不兼容的问题。



注意：CPU架构设计的不同，例如ARM和X86，将决定指令和指令集（SSE/NEON）的不同。指令集的不同，意味着优化方式也不同。

`x86`和`arm`的由来：指令集可分为复杂指令集（CISC）和精简指令集（RISC），前者属于一种便于编程和提高存储器访问效率的芯片设计体系，需要不同的时钟周期完成多条不同的指令，例如intel的8086和Motorola的68K，也就是通常所说的x86。后者属于一种提高处理器运行速度的芯片设计体系，关键技术是流水线piplining操作，即在一个时钟周期内完成多条指令。由于历史发展原因，因特尔和微软联手，将PC市场的CPU处理器采用x86架构体系，而移动端和嵌入式，特别是手机处理器，则多为arm架构。例如高通的骁龙400系列，600系列等，而700和800系列是基于arm公版自研的，华为的麒麟，公版的arm。



x86和arm的区别：前者高性能，后者低功耗，各自地位不同。打个比喻，前者是一个大学生，懂微积分和加减法，什么都能做；后者是多个小学生，只懂减法，微积分做不了。

CPU和GPU的浮点计算能力为什么不同？

答：见上图，左边是CPU右边是GPU。其中负责计算的单元叫Algorithm Logic Unit ALU. 从CPU和GPU架构模式上可以看出，后者明显面积更大，所以计算能力更强。

参考文献

[1] https://www.geeksforgeeks.org/modulo-2-binary-division/

[2] https://blog.csdn.net/lycb_gz/article/details/8201987

[3] https://blog.csdn.net/liyuanbhu/article/details/7882789

[4] https://blog.csdn.net/weixin_42108484/article/details/82692087

[5] http://man7.org/linux/man-pages/man2/timer_settime.2.html

[6] https://www.jianshu.com/p/aa96876ebabc

[7] http://www.cplusplus.com/reference/ctime/strftime/

[8] https://www.cnblogs.com/cstdio1/p/11175762.html