作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师

字节顺序的含义

字节顺序，也叫字节序，指一个元素的多个字节的排列顺序。
大端，左边字节是大值。英文叫 Big Endian 。
小端，左边字节是小值。英文叫 Little Endian 。
换个说法，字节顺序是元素的序列化方式。把元素转成多个字节叫序列化，把多个字节转成元素叫反序列化。

字节顺序的使用场景：
编译程序、运行程序。指令读写内存，操作数字、指针等，需要指定字节顺序。由操作系统、编译器管理。
文件操作。把数字写到文件，需要指定字节顺序。如果写文件用小端，读文件用大端，则解析数字不符合预期。
网络操作。网络协议，比如 TCP、UDP 协议。网络协议传递数字(比如 int 型)，需要指定字节顺序，发送方序列化，接收方反序列化。如果发送方用小端，接收方用大端，则解析数字不符合预期。

为什么要区分大端、小端？
数字，可以占用多个字节。把数字转为字节，把字节转为数字，和顺序有关。
举个通俗的例子，有 5 个字符，组成数字。"12345" 与 "54321"，含义不同。

大端的例子：
某个账户的余额为 67891 元。含义为 “6 万 7 千 8 百 9 十 1 个”，左边单位更大，所以是大端。
某个人的体重为 123kg 。含义为”1 百 2 十 3 个”，左边单位更大，所以是大端。

注意，内存本身没有大端、小端的含义。内存是一个很大的字节数组，每个字节相互独立。
形象的表示内存： `byte[] mem = new byte[99999]` ，这里的 99999 只为了表示空间很大。
CPU、编译器、程序，在使用数字和内存字节相互转换时，才有大端、小端的含义。

字符串的字节顺序

这里用单字节字符串，1 个字符占用 1 个字节。

编写代码： string_order.c

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

char *name = "Jack";

int main()
{
    // 查看完整的值。打印时，先打印大值，符合人的视角。
    printf("字符串  地址 = %p  值 = %s \n", name, name);

    char *ptr = name;
    for (size_t i = 0; i < 4; i++)
    {
        // 取一个字节。
```

```
        char *ptrX = (ptr + i);
        char ch = *ptrX;
        printf("当前字节值 = %c    地址 = %p \n", ch, ptrX);
    }

    return 0;
}
```

编译代码：

```
gcc string_order.c -o string_order -std=gnu99
```

运行代码：

```
[root@localhost order]# ./string_order
字符串  地址 = 0x400640  值 = Jack
当前字节值 = J    地址 = 0x400640
当前字节值 = a    地址 = 0x400641
当前字节值 = c    地址 = 0x400642
当前字节值 = k    地址 = 0x400643
```

分析结果：
ASCII 字符集，1 个字符占用 1 个字节。
首个字符的地址和字符串的地址一样。都是 0x400640。
4 个字节的地址，依次排列，每次加 1，0x400640 0x400641 0x400642 0x400643。
字符串”Jack”，不考虑末尾的\0，占用 4 个字节。在内存中，字节依次为 J、a、c、k ，和字符串的顺序一样。
单字节元素，没有大端、小端的区分，因为只占用 1 个字节。

整数的字节顺序

一个 int 整数，由 4 个字节组成。我们看看 4 个字节，在内存中是怎么布局的。

编写代码： int_order.c

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

// int 占用 4 个字节。
int value_int = 0x01020304;

int main()
{
    // 查看完整的值。打印时，先打印大值，符合人的视角。
    printf("int 变量  地址 = %p  16 进制值 = %#x  10 进制值 = %d \n",
           &value_int, value_int, value_int);

    // 取变量的地址。用 char 解析每个字节。
    char *ptr = (char *)&value_int;
```

```
    // 从左往右，遍历内存地址。  int 占用 4 个字节。
    for (size_t i = 0; i < 4; i++)
    {
        // 取一个字节。
        char *ptrX = (ptr + i);
        char ch = *ptrX;
        printf("当前字节值 = %#x   地址 = %p \n", ch, ptrX);
    }
}
```

编译代码：

```
gcc int_order.c -o int_order -std=gnu99
```

运行代码：

```
[root@localhost order]# ./int_order
int 变量  地址 = 0x601034  16 进制值 = 0x1020304  10 进制值 = 16909060
当前字节值 = 0x4   地址 = 0x601034
当前字节值 = 0x3   地址 = 0x601035
当前字节值 = 0x2   地址 = 0x601036
当前字节值 = 0x1   地址 = 0x601037
```

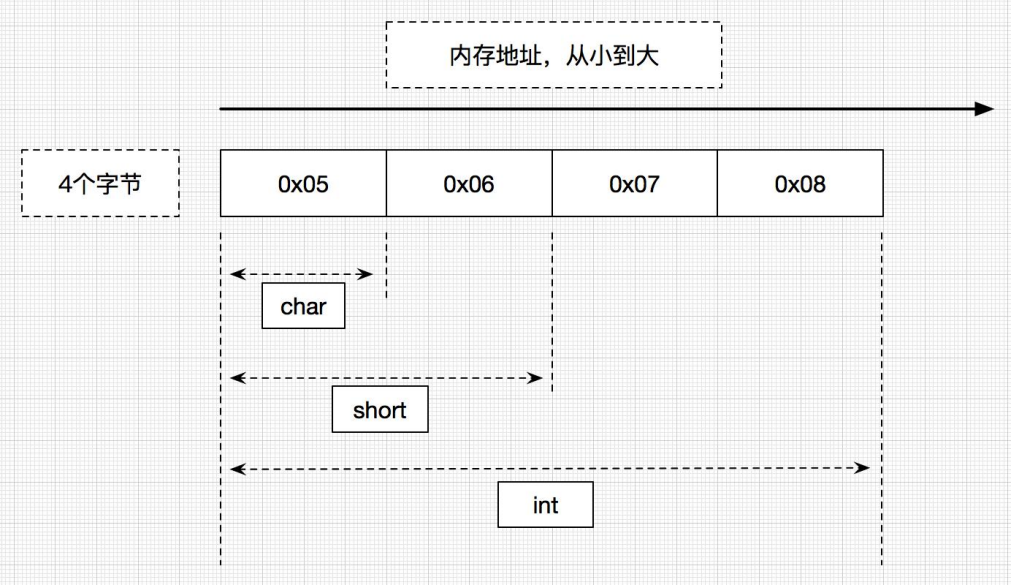
分析结果：
变量 value_int， 16 进制值 = 0x1020304 10 进制值 = 16909060 ,从人的视角看，大值在左侧。
int 变量占用内存 4 个字节，从左往右分别为 0x4、0x3、0x2、0x1。高位 0x01，在内存的右侧。低位 0x04，在内存的左侧。
int 变量在内存的布局，左边是小值，右边是大值。说明，x86 的 int 是小端。

小端顺序的优点

所有的 CPU 架构都使用小端吗？
不一定。有些 CPU 架构是大端。

x86 为什么用小端表示数字？
小端有许多好处。比如，强制类型转换，用字节数组组装数字，更加简洁。
强制类型转换的过程为，指定一个地址，取多个字节，放入新的地址。
变量、函数、对象等元素，都以所占用字节数组的第一个字节的地址表示元素的地址。

测试，用字节数组转数字。



编写代码： byte_transfer.c

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

// 内存中连续的多个字节。
char chars[] = {0x05, 0x06, 0x07, 0x08};

// 设为全局变量，方便查看符号。
char *char_p = NULL;
short *short_p = NULL;
int *int_p = NULL;

int main()
{
    // 用首地址，转为多种数字。
    char_p = (char *)chars;
    short_p = (short *)chars;
    int_p = (int *)chars;

    printf("字节数组的地址 = %p \n", &chars);
    printf("char  地址 = %p  值 = %#9x \n", char_p, *char_p);
    printf("short 地址 = %p  值 = %#9x \n", short_p, *short_p);
    printf("int   地址 = %p  值 = %#9x \n", int_p, *int_p);
}
```

编译代码：

```
gcc byte_transfer.c -o byte_transfer
gcc byte_transfer.c -S -o byte_transfer.s
```

运行代码：

```
[root@localhost order]# ./byte_transfer
字节数组的地址 = 0x601034
```

```
char  地址 = 0x601034  值 =       0x5
short 地址 = 0x601034  值 =     0x605
int   地址 = 0x601034  值 = 0x8070605
```

查看汇编文件：byte_transfer.s
数组，每个字节依次排列。

```
chars:
    .byte   5
    .byte   6
    .byte   7
    .byte   8
```

数字指针重置，直接把数组的地址，赋给数字指针。

```
    movq    $chars, char_p(%rip)
    movq    $chars, short_p(%rip)
    movq    $chars, int_p(%rip)
```

分析结果：
字节数组的地址，和 char/short/int 变量的地址相同，都是首个字节的地址 0x601034 。
用指针生成数字变量，直接又简洁。比如 `int *int_p = (int *)chars` ，对应的汇编为 `movq  $chars, int_p(%rip)` 。
直接把符号的值(即内存地址)赋给另一个符号，只用 1 个 movq 指令，没有其他操作，性能很高。

问题：为什么直接用指针赋值，就能生成对应的数字？
数字由连续的字节组成。取出连续的多个字节，按照顺序就能组装成一个数字。
这就意味着，没有数字含义的一组连续字节，也能用于组装数字。

用字符串转数字

编写代码： string_transfer.c

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

// 一个字符串。
char *ptr = "night moon";

// 一个 int 指针。
int *int_p = NULL;

int main()
{
    // 查看连续的 4 个字节
    printf("字符串 = %s \n", ptr);
    printf("单个字节 地址 = %p  值 = %#x \n", ptr, (*ptr));
    printf("单个字节 地址 = %p  值 = %#x \n", (ptr + 1), *(ptr + 1));
    printf("单个字节 地址 = %p  值 = %#x \n", (ptr + 2), *(ptr + 2));
    printf("单个字节 地址 = %p  值 = %#x \n", (ptr + 3), *(ptr + 3));
```

```
    // char 指针转为 int 指针
    int_p = (int *)ptr;
    printf("int   地址 = %p  值 = %#9x \n", int_p, *int_p);

    return 0;
}
```

编译代码：

```
gcc string_transfer.c -o string_transfer
gcc string_transfer.c -S -o string_transfer.s
```

运行代码：

```
[root@localhost order]# ./string_transfer
字符串 = night moon
单个字节 地址 = 0x4006d0  值 = 0x6e
单个字节 地址 = 0x4006d1  值 = 0x69
单个字节 地址 = 0x4006d2  值 = 0x67
单个字节 地址 = 0x4006d3  值 = 0x68
int   地址 = 0x4006d0  值 = 0x6867696e
```

分析结果：
字符串、int 变量，地址相同，都为 0x4006d0。
字符串，打印了连续的 4 个字节，分别为 0x6e 0x69 0x67 0x68。
int 变量，值为 0x6867696e 。包含了上方的 4 个字节。
说明，数字的生成非常灵活，给定一个合法的内存地址，就能生成数字。

用大端实现整数读写

编写代码： big_endian.c

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

// int 转 byte。大端模式。
void int_to_byte_big_endian(int tmp, char *byte_p)
{
    // 从小到大，取出 int 的 4 个字节对应的值。
    char ch1 = (char)((tmp >> 0) & 0xFF);
    char ch2 = (char)((tmp >> 8) & 0xFF);
    char ch3 = (char)((tmp >> 16) & 0xFF);
    char ch4 = (char)((tmp >> 24) & 0xFF);

    // 依次给 4 个字节赋值。
    // 大端。大值在左侧。
    *(byte_p + 0) = ch4;
    *(byte_p + 1) = ch3;
```

```
    *(byte_p + 2) = ch2;
    *(byte_p + 3) = ch1;

    printf("int 转 byte，大端模式： \n");
    printf("int 值 = %#x \n", tmp);
    printf("byte 依次为 %#x  %#x  %#x  %#x \n", *(byte_p + 0),
           *(byte_p + 1), *(byte_p + 2), *(byte_p + 3));
    printf("\n");
}

// byte 转 int。大端模式。
int byte_to_int_big_endian(char *byte_p)
{
    // 依次读取 4 个字节的值。
    char byte1 = *(byte_p + 0);
    char byte2 = *(byte_p + 1);
    char byte3 = *(byte_p + 2);
    char byte4 = *(byte_p + 3);

    // 大端。 大值在左侧。
    // 用 4 个字节，组装一个 int 。
    int tmp = 0;
    tmp += ((int)byte1) << 24;
    tmp += ((int)byte2) << 16;
    tmp += ((int)byte3) << 8;
    tmp += ((int)byte4) << 0;

    printf("byte 转 int，大端模式： \n");
    printf("byte 依次为 %#x  %#x  %#x  %#x \n", byte1, byte2, byte3, byte4);
    printf("int 值 = %#x \n", tmp);
    printf("\n");
}

// int 变量，占用 4 个字节
int int_param = 0x1A1B1C1D;

// 4 个连续的字节
char byte_buf[4];

int main()
{
    // int 转 byte。大端模式。
    int_to_byte_big_endian(int_param, byte_buf);

    // 重置 4 个字节。
    byte_buf[0] = 0x61;
    byte_buf[1] = 0x62;
    byte_buf[2] = 0x63;
    byte_buf[3] = 0x64;
```

```
    // byte 转 int。大端模式。
    byte_to_int_big_endian(byte_buf);
}
```

编译代码：

```
gcc big_endian.c -o big_endian
```

运行代码：

```
[root@localhost order]# ./big_endian
int 转 byte，大端模式：
int 值 = 0x1a1b1c1d
byte 依次为 0x1a  0x1b  0x1c  0x1d

byte 转 int，大端模式：
byte 依次为 0x61  0x62  0x63  0x64
int 值 = 0x61626364
```

分析结果：

int 和 byte 的相互转化，需要处理每个字节。

int 转 byte，int 变量的大值为 0x1a ，在左侧的 byte，说明实现了大端写。

int 包含 4 个字节，用移位运算、位与运算取出每个字节，把字节再写到对应的内存位置。

byte 转 int，左侧的 byte 为 0x61，在 int 变量的左侧，说明实现了大端读。

从内存地址，依次取出 4 个字节，用移位运算、加法运算，组装一个 int。