大端和小端（Big endian and Little endian）

一、大端和小端的问题

对于整型、长整型等数据类型，Big endian 认为第一个字节是最高位字节（按照从低地址到高地址的顺序存放数据的高位字节到低位字节）；而 Little endian 则相反，它认为第一个字节是最低位字节（按照从低地址到高地址的顺序存放据的低位字节到高位字节）。

例如，假设从内存地址 0x0000 开始有以下数据：
0x0000　　0x0001　　0x0002　　0x0003
0x12　　0x34　　0xab　　0xcd
如果我们去读取一个地址为 0x0000 的四个字节变量，若字节序为big-endian，则读出结果为0x1234abcd；若字节序为little-endian，则读出结果为0xcdab3412。

如果我们将0x1234abcd 写入到以 0x0000 开始的内存中，则Little endian 和 Big endian 模式的存放结果如下：
地址　　0x0000　　0x0001　　0x0002　　0x0003
big-endian　0x12　　0x34　　0xab　　0xcd
little-endian　0xcd　　0xab　　0x34　　0x12

一般来说，x86 系列 CPU 都是 little-endian 的字节序，PowerPC 通常是 big-endian，网络字节顺序也是 big-endian还有的CPU 能通过跳线来设置 CPU 工作于 Little endian 还是 Big endian 模式。

对于0x12345678的存储：

小端模式：（从低字节到高字节）
地位地址 0x78 0x56 0x34 0x12 高位地址

大端模式：（从高字节到低字节）
地位地址 0x12 0x34 0x56 0x78 高位地址

二、大端小端转换方法

htonl() htons() 从主机字节顺序转换成网络字节顺序
ntohl() ntohs() 从网络字节顺序转换为主机字节顺序

Big-Endian转换成Little-Endian

```
#define BigtoLittle16(A) ((((uint16)(A) & 0xff00) >> 8) | (((uint16)(A) & 0x00ff) << 8))
#define BigtoLittle32(A) ((((uint32)(A) & 0xff000000) >> 24) | (((uint32)(A) & 0x00ff0000) >> 8) | \
             (((uint32)(A) & 0x0000ff00) << 8) | (((uint32)(A) & 0x000000ff) << 24))
```

三、大端小端检测方法

如何检查处理器是big-endian还是little-endian?

C程序：

```
    int i = 1;
    char *p = (char *)&i;
    if(*p == 1)
          printf("Little Endian");
    else
          printf("Big Endian");
```

大小端存储问题，如果小端方式中（i占至少两个字节的长度）则i所分配的内存最小地址那个字节中就存着1，其他字节是0.大端的话则1在i的最高地址字节处存放，char是一个字节，所以强制将char型量p指向i则p指向的一定是i的最低地址，那么就可以判断p中的值是不是1来确定是不是小端。

联合体union的存放顺序是所有成员都从低地址开始存放，利用该特性就可以轻松地获得了CPU对内存采用Little-endian还是Big-endian模式读写。

```
/*return 1: little-endian, return 0: big-endian*/
int checkCPUendian()
{
  union
  {
    unsigned int a;
    unsigned char b;
  }c;
  c.a = 1;
  return (c.b == 1);
}
```

实现同样的功能，来看看Linux 操作系统中相关的源代码是怎么做的：

```
static union { char c[4]; unsigned long mylong; } endian_test = {{ 'l', '?', '?', 'b' } };

#define ENDIANNESS ((char)endian_test.mylong)
```

Linux 的内核作者们仅仅用一个union 变量和一个简单的宏定义就实现了一大段代码同样的功能！（如果ENDIANNESS='l'表示系统为little endian，为'b'表示big endian）

四、一些笔试题目

```
char *sz = "0123456789";
int *p = (int*)sz;
printf("%x\n",*++p);
```

字符'0'对应的十六进制是0x30，请问在x86环境下程序输出是多少?

假设字符串sz地址从@0开始，那么sz在内存的存储为

@0 @1 @2 @3 @4 @5 @6 @7 @8 @9

0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39

当你把char*强制类型转化成int*后，因为int占四个字节，那么p指向@0，并且*p占有的地址是@0@1@2@3，打印的时候 先进行++p操作，那么p指向@4，此时*p占有的地址是@4@5@6@7，根据上面地地址存地位，高地址存高位的解释，那么*p应该等于0x37363534

```
    int a = 0x12345678;
    char *p = (char*)(&a);
    printf("%x\n",*(p+1));
```

例如对于0x12345678，网络字节顺序是这样0x12,0x34,0x56,0x78存储的，这种方式称为big-endian

intel处理器是0x78 0x56 0x34 0x12这样来存储的，称为小尾little-endian

在x86环境下题目中的p指向0x78，加1后指向0x56

```
#include <stdio.h>
union
{
    int i;
    char x[2];
}a;
int main()
{
    a.x[0] = 10;
    a.x[1] = 1;
    printf("%d",a.i);
    return 0;
}
```

x86下输出答案： 266 （x86下：低位低地址，高位高地址，i内存里存的值是0x010A，十进制为266）

```
int main()
{
    union
    {
```

```
        int i;
        struct
        {
            char first;
            char second;
        }half;
    }number;
    number.i=0x4241;
    printf("%c %c\n", number.half.first, number.half.second);
    number.half.first='a';
    number.half.second='b';
    printf("%x\n", number.i);
    return 0;
}
```

x86下输出答案：

A B （0x41对应'A',是低位；0x42对应'B',是高位）

6261 (number.i和number.half共用一块地址空间0x6261)