网络程序设计

网络数据的内容与形态

整数的长度

- int、short、char和long这些整数类型具有不同的大小，而且大小会因平台而异
- 可以用sizeof()运算符获得数据类型在当前平台的占据内存空间的大小。
- C99语言标准规范了一组可选类型来确定整数的大小。int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t

类型	16位平台	32位平台	64位平台
char	1个字节	1个字节	1个字节
short	2个字节	2个字节	2个字节
int	2个字节	4个字节	4个字节
unsigned int	2个字节	4个字节	4个字节
float	4个字节	4个字节	4个字节
double	8个字节	8个字节	8个字节
long	4个字节	4个字节	8个字节
long long	8个字节	8个字节	8个字节
unsigned long	4个字节	4个字节	8个字节
指针	2个字节	4个字节	8个字节
最大存储空间	2^16	2^32	2^64

在 stdint.h 中 的定义

```
#ifndef __int8_t_defined
# define __int8_t_defined
typedef signed char          int8_t;
typedef short int            int16_t;
typedef int                  int32_t;
# if __WORDSIZE == 64
typedef long int             int64_t;
# else
__extension__
typedef long long int        int64_t;
# endif
#endif
```

整数的符号

- 负数的表示——补码
- 符号扩展
  - 有符号数的扩展与无符号数的扩展
- 自动类型转换
  - 同种数据类型的运算结果，还是该类型(short例外)
  - 不同种数据类型的运算结果，是两种类型中取值范围更大的那种。
  - 任何计算发生之前，会把变量的值加宽到本机(int)大小
- 在发送者和接收者传输数据的过程中，符号性的协商是非常必要的。

字节顺序

- 大端(big-endian)顺序: 高字节数据存放在内存低地址处，低字节数据存放在内存高地址处
- 小端(little-endian)顺序: 低字节数据存放在内存低地址处，高字节数据存放在内存高地址处

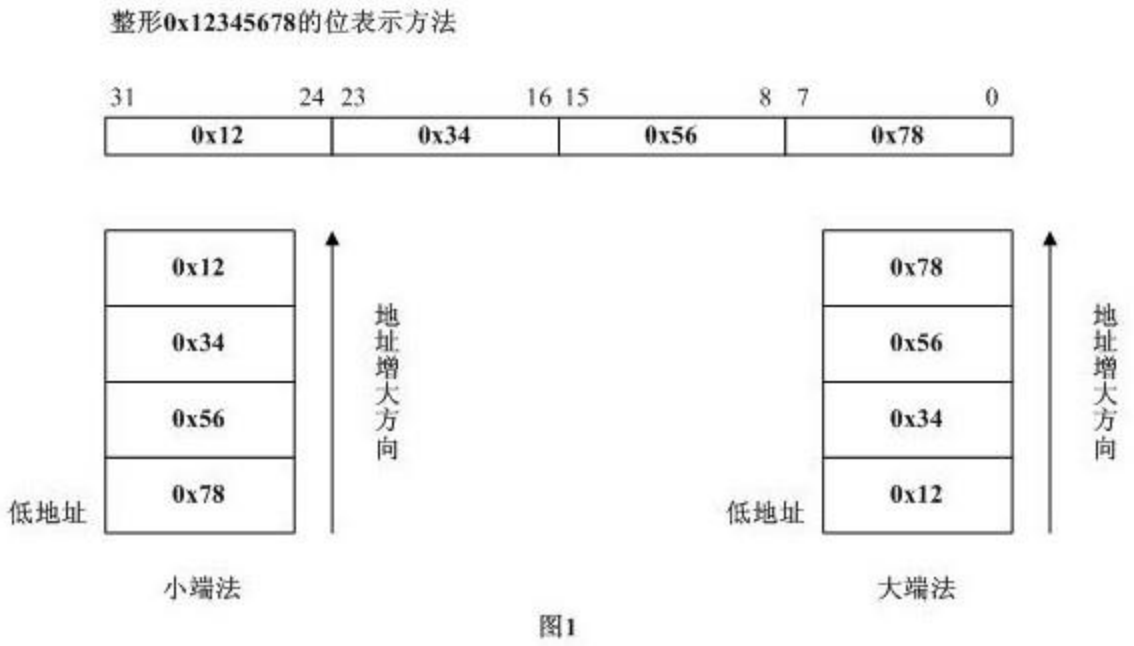


图1

Linux 32（Intel IA32）、Windows（Intel IA32）和Linux 64（Intel x86-64)是小端法；Sun（运行Solaris的Sun Microsystems SPARC处理器）是大端法。

字节顺序

- 主机字节顺序：整数在内存中保存的顺序
- 网络字节顺序：网络通讯过程中，用于网络中发送和接收的数据都采用大端字节顺序。
- 数据发送前，需要将主机字节顺序的数据转换为网络字节顺序再进行网络传输。
- 接收网络数据时，需要把数据从网络字节顺序转换为主机字节顺序后再进行处理。

字节顺序

- WinSock提供了一些函数来处理主机字节顺序和网络字节顺序的转换

htons 把 unsigned short 类型从主机序转换到网络序

htonl 把 unsigned long 类型从主机序转换到网络序

ntohs 把 unsigned short 类型从网络序转换到主机序

ntohl 把 unsigned long 类型从网络序转换到主机序

字节顺序

- 由于IP、UDP、TCP都把用户数据看作没有结构的字节集合，所以不关心用户数据中整数是否为网络字节顺序。因此用户数据的字节顺序必须由开发者达成一致。

字节顺序

- 什么时候需要使用字节顺序转换？
  - 用户定义的数据类型为大于1字节的整数时，需要字节顺序转换
  - 当使用解析函数如gethostbyname()等，数值是以网络字节顺序方式返回，不需要在发送前进行字节顺序转换。

浮点数的传送


●双精度浮点数（共64位）
1位
11位
52位
符号部分
指数部分
尾数部分
●单精度浮点数（共32位）
1位
8位
23位
符号部分
指数部分
尾数部分
头条@Go语言中文网

浮点数网络传输

- 1、将浮点数a通过内存拷贝，赋值给相同字节的整型数据b；
- 2、将b转换为 络字节序变量c并发送到服务端；
- 3、服务端接收c并将c转换为主机字节序变量d；
- 4、将整型数据d通过内存拷贝，赋值给相同字节的浮点数据e；

结构的对齐与填充

- 内存对齐：计算机系统对于基本数据类型在内存中的存放位置都有限制，要求这些数据存储的首地址是某个数K的倍数。这样各种基本数据类型在内存中就是按照一定的规则排列的，而不是一个紧挨着一个排放。
- 对齐模数：内存对齐中指定的对齐数值K。
- 各硬件平台对存储空间的处理有很大的不同，因此网络数据传输中的结构化定义必须考虑内存对齐会影响到的变量的位置。

结构的对齐与填充

- 微软C编译器的对齐策略是:
  - 1)结构体变量的首地址能够被其最宽基本类型成员的大小所整除。
  - 2)结构体每个成员相对于结构体首地址的偏移量都是成员大小的整数倍，如果有需要，编译器会在成员之间加上填充字节。
  - 3)结构体的总大小为结构体最宽基本类型成员大小的整数倍。

0	a (char)
1	
2	
3	
4	b (int)
5	
6	
7	
8	c (double)
9	
10	
11	
12	
13	
14	
15	

图 1 结构体 S1 存储结构示意图

0	a (char)
1	
2	
3	
4	
5	
6	
7	
8	b (double)
9	
10	
11	
12	
13	
14	
15	
16	c (int)
17	
18	
19	
20	补齐到 8 的整数倍
21	
22	
23	

图 2 结构体 S2 存储结构示意图

结构的对齐与填充

- #pragma pack 能够改变C编译器默认的对齐方式
- #pragma pack([show] | [push|pop] [, identifier], n)
- show:显示当前对齐模数，以警告消息形式显示
- push:将当前指定的对齐模数进行压栈操作，同时设置当前对齐模数为n。
- pop:从内部编译器堆栈中删除最顶端的记录。如果没记录n，则当前栈顶记录为新的对齐模数
- identifier:与push一起使用时赋予当前被压入栈的记录一个名词，同pop一起使用时，出栈直到identifier出栈，如果指定的identifier不在堆栈，则忽略pop操作
- n:对齐模数，缺省值8，合法值为：1,2,4,8,16

结构的对齐与填充

- 为了避免数据构造和成帧在结构和字节对齐上的歧义，传输数据中结构化二进制数据的定义一般会考虑对齐问题，尽量把字段按照对齐策略进行排列，并显式增加填充字段。
- 这一方面可以避免由于对齐处理歧义带来的数据理解错误，另一方面可以为协议将来的扩充预留空间。

网络数据传输形态

- 在通信两端进行数据交互的数据格式有文本串和二进制两种形态。
- 使用文本串进行消息传递时的一般做法是：
  - 1)定义消息命令。定义一些固定含义的文本串作为消息的控制命令。
  - 2)定义消息标识。定义一些固定含义的文本串来标识消息内容。
  - 3)选择文本表示方式。使得数字、布尔值等数据类型可以表示为文本字符串。
  - 4)选择编码方式。ASCII,Unicode等

网络数据传输形态

- 当传递二进制格式的消息时，需要对传递内容进行定义，规范该内容的字节长度、位置及含义。
- 当使用二进制数据进行消息传递时需要注意以下几个问题：
  - 不同实现以不同的格式存储二进制数据。大端，小端
  - 不同实现在存储相同数据类型时可能也不同。32位，64位
  - 不同实现为协议打包的方式也不同。对齐策略
- 尽量显式做到字节对齐

字符编码

- 字节：计算机中存储数据的基本单元
- 字符：抽象意义的符号，是各种文字、标点符号、图形符号、数字等的总称。
- 字符集：一组抽象的字符集合。ASCII,GB2312,BIG5,JIS
- 字符编码：规定了每个字符分别用一个字节还是多个字节表示，及用哪个字节值来存储。

字符编码

- 字符编码的发展经历了三个阶段
  - 第一阶段：字符编码的产生。ASCII
  - 第二阶段：字符编码的本地化。ANSI编码:GB2312, JIS
  - 第三阶段：字符编码的国际化。Unicode
- Unicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

字符编码

- Unicode编码系统分为编码方式和实现方式两个层次。
- Unicode的编码方式与通用字符集(UCS)概念相对应。目前对应于UCS-2
- Unicode的实现方式称为Unicode转换格式(UTF)。它规定了如何对UCS对应的基本多语种码点(BMP)进行传输的策略。UTF-8, UTF-16, UTF-32

字符编码

- UTF-8编码策略
  - 首先，针对不同语言的字符集，使用不同数目的字节编码序列。例如，汉字的编码采用3个字节长度的编码序列。
  - 其次，第一个字节的换码位序列的特征指明了编码序列的字节数。若编码序列的字节数超过两个，则每个字节都由1个换码位序列开始。
    - 首字节，换码位序列为n个值为1的二进制位加上1个值为0的二进制位
    - 后续字节的换码位序列均为二进制的’10’

字符编码

UTF-8编码字节序列

1字节 0xxxxxxx

2字节 110xxxxx 10xxxxxx

3字节 1110xxxx 10xxxxxx 10xxxxxx

4字节 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

5字节 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

6字节 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

文本化传输编码标准

- Base64编码
- UTF-7编码
- QP(Quoted Printable)编码

Base64编码

- 3字节->4字节
- 6bit编码成8bit
- 每76个字符加一个换行符。
- 分组不足三字节的。两字节则在编码结果后面加1个”=”。一个字节则在编码结果后面加2个”=”

Base64编码

索引	对应字符	索引	对应字符	索引	对应字符	索引	对应字符
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y		

文本	M						a						n					
ASSII编码	77						97						110					
二进制位	0	1	0	0	1	1	0	1	0	1	1	0	0	0	1	0	1	1	0	1	1	1	0
索引	19						22						5					
Base64编码	T						W						F					

文本（1 Byte）	A																						
二进制位	0	1	0	0	0	0	0	1															
二进制位（补0）	0	1	0	0	0	0	0	1	0	0	0	0											
Base64编码	Q						Q						=						=				
文本（2 Byte）	B						C																
二进制位	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	1							
二进制位（补0）	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	1	0	0					
Base64编码	Q						k						M						=				

UTF7

- 一个修改的Base64，目的是传输Unicode数据(UTF-16)
- 对被编码成UTF-16的ASCII字符，可直接用ASCII等价字节表示。
- 对被编码成UTF-16的非ASCII字符，先通过Base64编码输出后,再在前面加上字符”+”， 并在结尾加上字符”-”。表示非ASCII字符编码结果的起始和结束。
- 可使用非字母表中的字符来表示起止符。

QP编码

- 对于ASCII字符，不进行转换，直接用ASCII等价字节表示
- 对于非ASCII字符，一个字节用两个十六进制数表示，前面加”=”
- 原始数据中的等号”=”用”=3D”表示

数据校验

- 二进制反码求和
- 计算校验和步骤
  - 1)将校验和字段置为0；
  - 2)填充校验和覆盖范围内所有的数据内容；
  - 3)将校验和覆盖范围内的数据看成是以16位单位的数字组成
  - 4)计算校验和，并拷贝到校验和字段

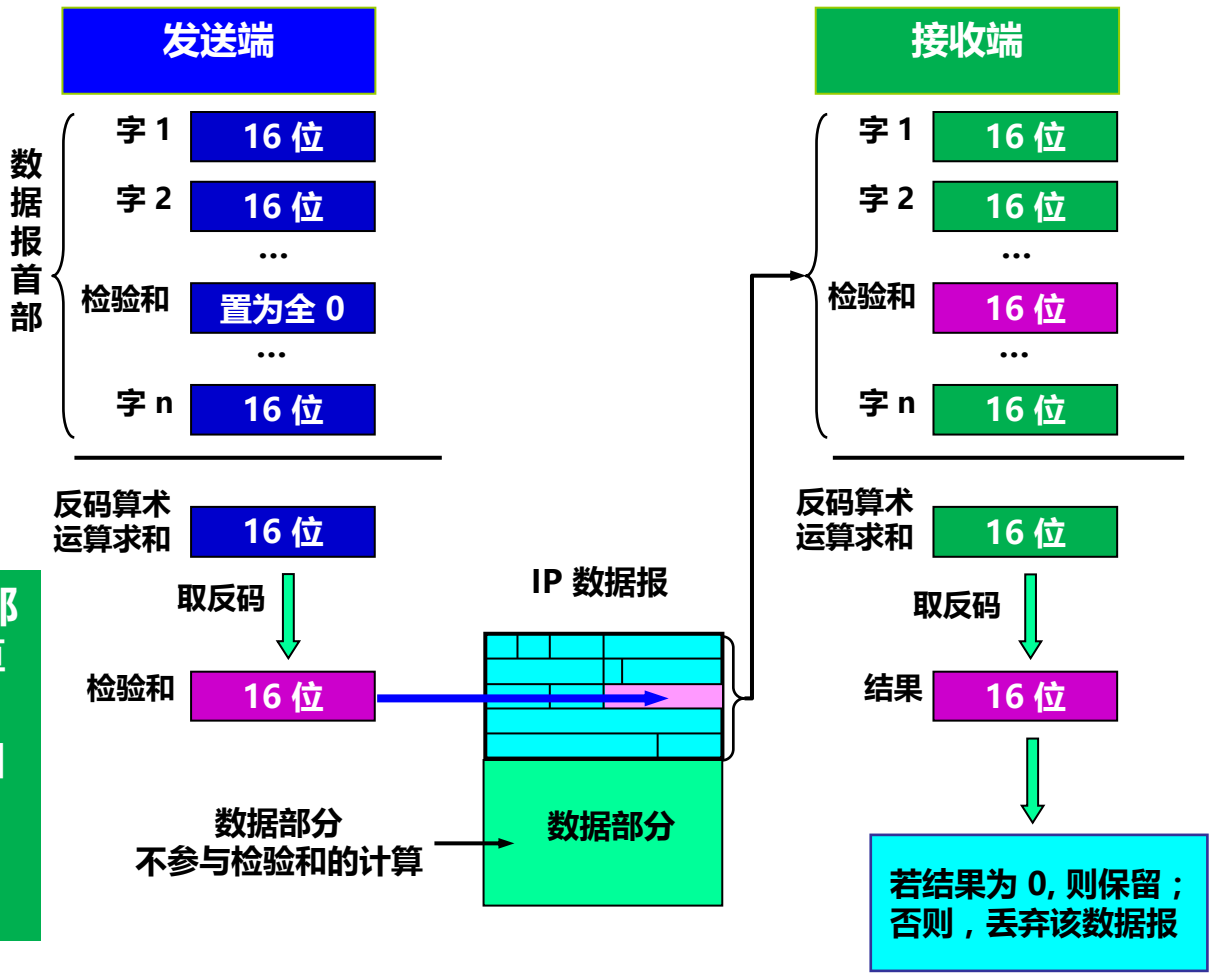
数据校验

- 接收方差错检验的步骤
  - 1)将校验和覆盖范围内的数据(包括校验和在内)看成是以16位为单位的数字组成
  - 2)计算校验和
  - 3)检查计算出的校验和结果是否等于0，接收报文或丢弃报文

数据校验

- 反码运算求和
  - 从低位到高位，按位相加，有进位则向高位进1(和一般的二进制法则一样),**若最高位有进位，则向最低位进1.**
  - 最后的运算结果取反码

IP 数据报首部检验和的计算采用 16 位二进制反码求和算法



```
#include <windef.h>

USHORT CheckSum(USHORT *pchBuffer, int iSize) { //USHORT:16bit
    unsigned long ulCksum = 0; //32bit
    //对16位字求和
    while (iSize>1){
        ulCksum += *pchBuffer++;
        iSize -= sizeof(USHORT);
    }
    if (iSize) {
        ulCksum += *(UCHAR *)pchBuffer;
    }
    //进位求和
    ulCksum = (ulCksum >> 16) + (ulCksum & 0xffff);
    ulCksum += (ulCksum >> 16);
    //取反
    return (USHORT)(~ulCksum);
}
```