用处

 最近在学习python网络编程这一块，在写简单的socket通信代码时，遇到了struct这个模块的使用，当时不太清楚这到底有和作用，后来查阅了相关资料大概了解了，在这里做一下简单的总结。

    了解c语言的人，一定会知道struct结构体在c语言中的作用，它定义了一种结构，里面包含不同类型的数据(int,char,bool等等)，方便对某一结构对象进行处理。而在网络通信当中，大多传递的数据是以二进制流（binary data）存在的。当传递字符串时，不必担心太多的问题，而当传递诸如int、char之类的基本数据的时候，就需要有一种机制将某些特定的结构体类型打包成二进制流的字符串然后再网络传输，而接收端也应该可以通过某种机制进行解包还原出原始的结构体数据。python中的struct模块就提供了这样的机制，该模块的主要作用就是对python基本类型值与用python字符串格式表示的C struct类型间的转化（This module performs conversions between Python values and C structs represented as Python strings.）。stuct模块提供了很简单的几个函数，下面写几个例子。

1. 按照指定格式将Python数据转换为字符串,该字符串为字节流,如网络传输时,不能传输int,此时先将int转化为字节流,然后再发送;
2. 按照指定格式将字节流转换为Python指定的数据类型;
3. 处理二进制数据,如果用struct来处理文件的话,需要用’wb’,’rb’以二进制(字节流)写,读的方式来处理文件;
4. 处理c语言中的结构体;

struct模块中的函数

| **函数** | **return** | **explain** |
| --- | --- | --- |
| pack(fmt,v1,v2…) | string | 按照给定的格式(fmt),把数据转换成字符串(字节流),并将该字符串返回. |
| pack\_into(fmt,buffer,offset,v1,v2…) | None | 按照给定的格式(fmt),将数据转换成字符串(字节流),并将字节流写入以offset开始的buffer中.(buffer为可写的缓冲区,可用array模块) |
| unpack(fmt,v1,v2…..) | tuple | 按照给定的格式(fmt)解析字节流,并返回解析结果 |
| pack\_from(fmt,buffer,offset) | tuple | 按照给定的格式(fmt)解析以offset开始的缓冲区,并返回解析结果 |
| calcsize(fmt) | size of fmt | 计算给定的格式(fmt)占用多少字节的内存，注意对齐方式 |

格式化字符串

当打包或者解包的时,需要按照特定的方式来打包或者解包.该方式就是格式化字符串,它指定了数据类型,除此之外,还有用于控制字节顺序、大小和对齐方式的特殊字符.

对齐方式

为了同c中的结构体交换数据，还要考虑c或c++编译器使用了字节对齐，通常是以4个字节为单位的32位系统，故而struct根据本地机器字节顺序转换.可以用格式中的第一个字符来改变对齐方式.定义如下

| **Character** | **Byte order** | **Size** | **Alignment** |
| --- | --- | --- | --- |
| @(默认) | 本机 | 本机 | 本机,凑够4字节 |
| = | 本机 | 标准 | none,按原字节数 |
| < | 小端 | 标准 | none,按原字节数 |
| > | 大端 | 标准 | none,按原字节数 |
| ! | network(大端) | 标准 | none,按原字节数 |

如果不懂大小端,见[大小端参考网址](http://blog.csdn.net/lis_12/article/details/52698634).

格式符

| **格式符** | **C语言类型** | **Python类型** | **Standard size** |
| --- | --- | --- | --- |
| x | pad byte(填充字节) | no value |  |
| c | char | string of length 1 | 1 |
| b | signed char | integer | 1 |
| B | unsigned char | integer | 1 |
| ? | \_Bool | bool | 1 |
| h | short | integer | 2 |
| H | unsigned short | integer | 2 |
| i | int | integer | 4 |
| I(大写的i) | unsigned int | integer | 4 |
| l(小写的L) | long | integer | 4 |
| L | unsigned long | long | 4 |
| q | long long | long | 8 |
| Q | unsigned long long | long | 8 |
| f | float | float | 4 |
| d | double | float | 8 |
| s | char[] | string |  |
| p | char[] | string |  |
| P | void \* | long |  |

注- -!

1. \_Bool在C99中定义,如果没有这个类型,则将这个类型视为char,一个字节;
2. q和Q只适用于64位机器;
3. 每个格式前可以有一个数字,表示这个类型的个数,如s格式表示一定长度的字符串,4s表示长度为4的字符串;4i表示四个int;
4. P用来转换一个指针,其长度和计算机相关;
5. f和d的长度和计算机相关;

class struct.Struct(format)

返回一个struct对象（结构体，参考C）。

该对象可以根据格式化字符串的**格式**来读写二进制数据。

第一个参数（格式化字符串）可以指定字节的顺序。

默认是根据系统来确定，也提供自定义的方式，只需要在前面加上特定字符即可：

struct.Struct('>I4sf')

特定字符对照表附件有。

**常见方法和属性：**

**方法**

pack(*v1*, *v2*, *…*)

返回一个字节流对象。

按照fmt（格式化字符串）的格式来打包参数v1,v2,...。

通俗的说就是：

首先将不同类型的数据对象放在一个“组”中（比如元组(1,'good',1.22)），

然后打包（“组”转换为字节流对象），最后再解包（将字节流对象转换为“组”）。

pack\_into(buffer, offset, v1, v2, …)

根据格式字符串fmt包装值v1，v2，...，并将打包的字节写入从位置偏移开始的可写缓冲***buffer***。请注意，offset是必需的参数。

unpack\_from(buffer, offset=0)

根据格式字符串fmt，从位置偏移开始从缓冲区解包。 结果是一个元组，即使它只包含一个项目。 缓冲区的大小（以字节为单位，减去偏移量）必须至少为格式所需的大小，如calcsize()所反映的。

**属性**

format

格式化字符串。

size

结构体的大小。

**实例：**

1.通常的打包和解包

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # -\*- coding: utf-8 -\*-  """  打包和解包  """  import struct  import binascii    values = (1, b'good', 1.22) #查看格式化对照表可知，字符串必须为字节流类型。  s = struct.Struct('I4sf')  packed\_data = s.pack(\*values)  unpacked\_data = s.unpack(packed\_data)    print('Original values:', values)  print('Format string :', s.format)  print('Uses :', s.size, 'bytes')  print('Packed Value :', binascii.hexlify(packed\_data))  print('Unpacked Type :', type(unpacked\_data), ' Value:', unpacked\_data) |

 结果：

Original values: (1, b'good', 1.22)

Format string : b'I4sf'

Uses : 12 bytes

Packed Value : b'01000000676f6f64f6289c3f'

Unpacked Type : <class 'tuple'> Value: (1, b'good', 1.2200000286102295)

[Finished in 0.1s]

说明：

首先将数据对象放在了一个元组中，然后创建一个Struct对象，并使用pack()方法打包该元组；最后解包返回该元组。

这里使用到了*binascii.hexlify(data)*函数。

binascii.hexlify(data)

返回字节流的十六进制字节流。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> a = 'hello'  >>> b = a.encode()  >>> b  b'hello'  >>> c = binascii.hexlify(b)  >>> c  b'68656c6c6f' |

2.使用buffer来进行打包和解包

 使用通常的方式来打包和解包会造成内存的浪费，所以python提供了buffer的方式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # -\*- coding: utf-8 -\*-  """  通过buffer方式打包和解包  """  import struct  import binascii  import ctypes    values = (1, b'good', 1.22) #查看格式化字符串可知，字符串必须为字节流类型。  s = struct.Struct('I4sf')  buff = ctypes.create\_string\_buffer(s.size)  packed\_data = s.pack\_into(buff,0,\*values)  unpacked\_data = s.unpack\_from(buff,0)    print('Original values:', values)  print('Format string :', s.format)  print('buff :', buff)  print('Packed Value :', binascii.hexlify(buff))  print('Unpacked Type :', type(unpacked\_data), ' Value:', unpacked\_data) |

 结果：

[复制代码](javascript:void(0);)

Original values1: (1, b'good', 1.22)  
Original values2: (b'hello', True)  
buff : <ctypes.c\_char\_Array\_18 object at 0x000000D5A5617348>  
Packed Value : b'01000000676f6f64f6289c3f68656c6c6f01'  
Unpacked Type : <class 'tuple'>  Value: (1, b'good', 1.2200000286102295)  
Unpacked Type : <class 'tuple'>  Value: (b'hello', True)  
[Finished in 0.1s]

[复制代码](javascript:void(0);)

说明：

针对buff对象进行打包和解包，避免了内存的浪费。

这里使用到了函数

ctypes.create\_string\_buffer(init\_or\_size,size = None)

创建可变字符缓冲区。  
返回的对象是c\_char的ctypes数组。

init\_or\_size必须是一个整数,它指定数组的大小,或者用于初始化数组项的字节对象。

3.使用buffer方式来打包多个对象

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | # -\*- coding: utf-8 -\*-  """  buffer方式打包和解包多个对象  """  import struct  import binascii  import ctypes    values1 = (1, b'good', 1.22) #查看格式化字符串可知，字符串必须为字节流类型。  values2 = (b'hello',True)  s1 = struct.Struct('I4sf')  s2 = struct.Struct('5s?')  buff = ctypes.create\_string\_buffer(s1.size+s2.size)  packed\_data\_s1 = s1.pack\_into(buff,0,\*values1)  packed\_data\_s2 = s2.pack\_into(buff,s1.size,\*values2)  unpacked\_data\_s1 = s1.unpack\_from(buff,0)  unpacked\_data\_s2 = s2.unpack\_from(buff,s1.size)    print('Original values1:', values1)  print('Original values2:', values2)  print('buff :', buff)  print('Packed Value :', binascii.hexlify(buff))  print('Unpacked Type :', type(unpacked\_data\_s1), ' Value:', unpacked\_data\_s1)  print('Unpacked Type :', type(unpacked\_data\_s2), ' Value:', unpacked\_data\_s2) |

 结果：

Original values2: (b'hello', True)

buff : <ctypes.c\_char\_Array\_18 object at 0x000000D5A5617348>

Packed Value : b'01000000676f6f64f6289c3f68656c6c6f01'

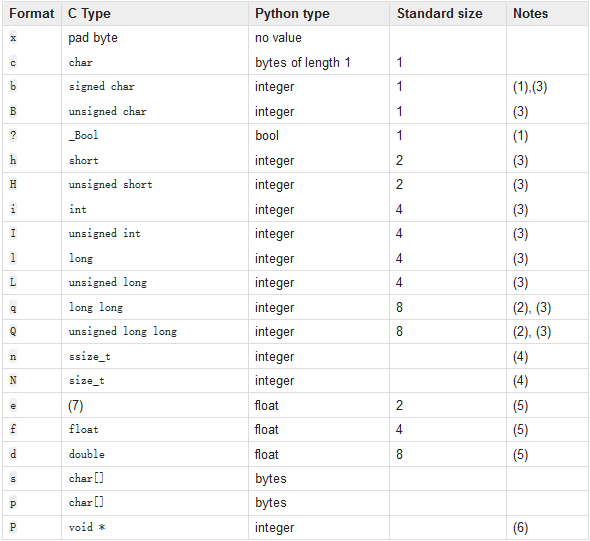
Unpacked Type : <class 'tuple'> Value: (1, b'good', 1.2200000286102295)

Unpacked Type : <class 'tuple'> Value: (b'hello', True)

[Finished in 0.1s]

**附：**

1.格式化对照表



提示：

signed char（有符号位）取值范围是 -128 到 127(有符号位)  
unsigned char （无符号位）取值范围是 0 到 255

[Python参考手册struct模块链接](https://docs.python.org/2/library/struct.html" \l "struct-format-strings" \t "_blank)