

python中TCP协议中的粘包问题

转载

weixin_34290390

于 2019-03-22 14:17:03 发布 60 收藏

版权

文章标签:

网络

python

json

1.粘包现象

基于TCP实现一个简易远程cmd功能

复制代码

服务端

```
import socket
import subprocess
sever = socket.socket()
sever.bind(('127.0.0.1', 33521))
sever.listen()
while True:

    1 client, address = sever.accept()
    2 while True:
    3     try:
    4         cmd = client.recv(1024).decode('utf-8')
    5         p1 = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr= subprocess.PIPE)
    6         data = p1.stdout.read()
    7         err_data = p1.stderr.read()
    8         client.send(data)
    9         client.send(err_data)
    10    except ConnectionResetError:
    11        print('connect broken')
    12        client.close()
    13        break

sever.close()
```

客户端

```
import socket
client = socket.socket()
client.connect(('127.0.0.1', 33521))
while True:

    1 cmd = input('请输入指令(Q\q退出)>>:').strip().lower()
    2 if cmd == 'q':
    3     break
    4 client.send(cmd.encode('utf-8'))
    5 data = client.recv(1024)
    6 print(data.decode('gbk'))

client.close()
```

复制代码

上述是基于TCP协议的远程cmd简单功能，在运行时会发生粘包。

2、什么是粘包？

只有TCP会发生粘包现象，UDP协议永远不会发生粘包；

TCP：（transport control protocol，传输控制协议）流式协议。在socket中TCP协议是按照字节数进行数据的收发，数据的发送方发出的数据往往接收方不知道数据到底长度是多长，而TCP协议由于本身为了提高传输的效率，发送方往往需要收集到足够的数据才会进行发送。使用了优化方法（Nagle算法），将多次间隔较小且数据量小的数据，合并成一个大的数据包，然后进行打包，这样接收端就能正确收到来了，必须提供科学的拆包机制。即面向流的通信



weixin_34290390

关注

0

UDP: (user datagram protocol, 用户数据报协议) 数据报协议。在socket中udp协议收发数据是以数据报为单位, 服务端和客户端收发数据是以一个单位, 所以不会使用块的合并优化算法, 由于UDP支持的是一对多的模式, 所以接收端的skbuff(套接字缓冲区) 采用了链式结构来记录每一个到达的UDP包, 在每个UDP包中就有了消息头(消息来源地址, 端口等信息), 这样, 对于接收端来说, 就容易进行区分处理了。即面向消息的通信是有消息保护边界的。

TCP协议不会丢失数据, UDP协议会丢失数据。

udp的recvfrom是阻塞的, 一个recvfrom(x)必须对唯一的一个sendto(y), 收完了x个字节的数据就算完成, 若是 $y > x$ 数据就丢失, 这意味着udp根本不会粘包, 但是会丢数据, 不可靠。

tcp的协议数据不会丢, 没有收完包, 下次接收, 会继续上次继续接收, 己端总是在收到ack时才会清除缓冲区内容。数据是可靠的, 但是会粘包。

3、什么情况下会发生粘包?

1. 由于TCP协议的优化算法, 当单个数据包较小的时候, 会等到缓冲区满才会发生数据包前后数据叠加在一起的情况。然后取的时候就分不清了到底是哪段数据, 这是第一种粘包。

2. 当发送的单个数据包较大超过缓冲区时, 收数据方一次就只能取一部分的数据, 下次再收数据方再收数据将会延续上次为接收数据。这是第二种粘包。

粘包的本质问题就是接收方不知道发送数据方一次到底发送了多少数据, 解决问题的方向也是从控制数据长度着手, 也就是如何设置缓冲区的问题

4、如何解决粘包问题?

解决问题思路: 上述已经明确粘包的产生是因为接收数据时不知道数据的具体长度。所以我们应该先发送一段数据表明我们发送的数据长度, 那么就不会产生数据没有发送或者没有收取完全的情况。

1.struct 模块 (结构体)

struct模块的功能可以将python中的数据类型转换成C语言中的结构体 (bytes类型)

复制代码

```
import struct
s = 123456789
res = struct.pack('i', s)
print(res)
```

```
res2 = struct.unpack('i', res)
print(res2)
print(res2[0])
```

复制代码

2.粘包的解决方案基本版

既然我们拿到了一个可以固定长度的办法, 那么应用struct模块, 可以固定长度了。

为字节流加上自定义固定长度报头, 报头中包含字节流长度, 然后一次send到对端, 对端在接收时, 先从缓存中取出定长的报头, 然后再取真实数据

复制代码

服务器端

```
import socket
import subprocess
import struct
sever = socket.socket()
sever.bind(('127.0.0.1', 33520))
sever.listen()
while True:
```

```
1 client, address = sever.accept()
2 while True:
3     try:
4         cmd = client.recv(1024).decode('utf-8')
5         #利用子进程模块启动程序
6         p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
7         #管道输出的信息有正确和错误的
8         data = p.stdout.read()
```



weixin_34290390

关注

0

```

9 |         err_data = p.stderr.read()
10 |         #先将数据的长度发送给客户端
11 |         length = len(data)+len(err_data)
12 |         #利用struct模块将数据的长度信息转化成固定的字节
13 |         len_data = struct.pack('i', length)
14 |         #以下将信息传输给客户端
15 |         #1.数据的长度
16 |         client.send(len_data)
17 |         #2.正确的数据
18 |         client.send(data)
19 |         #2.错误管道的数据
20 |         client.send(err_data)
21 |     except Exception as e:
22 |         client.close()
23 |         print('连接中断。。。')
24 |         break

```

客户端

```

import socket
import struct

```

```

client = socket.socket()
client.connect(('127.0.0.1', 33520))
while True:

```

```

1 | cmd = input('请输入指令>>:').strip().encode('utf-8')
2 | client.send(cmd)
3 | #1.先接收传过来数据的长度是多少，我们通过struct模块固定了字节长度为4
4 | length = client.recv(4)
5 | #将struct的字节再转回去整型数字
6 | len_data = struct.unpack('i', length)
7 | print(len_data)
8 | len_data = len_data[0]
9 | print('数据长度为%s:' % len_data)

```

```

1 | all_data = b''
2 | recv_size = 0
3 | #2.接收真实的数据
4 | #循环接收直到接收到数据的长度等于数据的真实长度（总长度）
5 | while recv_size < len_data:
6 |     data = client.recv(1024)
7 |     recv_size += len(data)
8 |     all_data += data

```

```

1 | print('接收长度%s' % recv_size)
2 | print(all_data.decode('gbk'))

```

复制代码

#总结:

```

1 | 服务器端:
2 | 1.在服务器端先收到命令，打开子进程，然后计算返回的数据的长度
3 | 2.先利用struct模块将数据长度转成固定4个字节传给客户端
4 | 3.再向客户端发送真实的数据。
5 | 客户端（两次接收）:
6 | 1.第一次只接受4个字节，因为长度数据就是4个字节。这样防止了数据粘包。解码得到长度数据
7 | 2.第二次循环接收真实数据，拼接真实数据完成

```



weixin_34290390

关注

0

很显然，如果仅仅只是这样肯定无法满足在实际生产中一些需求。那么该怎么修改？

我们可以把报头做成字典，字典里包含将要发送的真实数据的详细信息，然后json序列化，然后用struct将序列化后的数据长度打包成4个字节（4个字节足够用了）

我们可以将自定义的报头设置成这种这种格式。

发送时：

1先发报头长度

2再编码报头内容然后发送

3最后发真实内容

接收时：

1先收报头长度，用struct取出来

2根据取出的长度收取报头内容，然后解码，反序列化

3从反序列化的结果中取出待取数据的详细信息，然后去取真实的数据内容

复制代码

服务器端

```
import socket
import subprocess
import datetime
import json
import struct

sever = socket.socket()
sever.bind(('127.0.0.1', 33520))
sever.listen()

while True:

    1 client, address = sever.accept()
    2 while True:
    3     try:
    4         cmd = client.recv(1024).decode('utf-8')
    5         #启动子进程
    6         p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    7         #得到子进程运行的数据
    8         data = p.stdout.read() #子进程运行正确的输出管道数据，数据读出来是字节
    9         err_data = p.stderr.read() #子进程运行错误的输出管道数据
    10        #计算数据的总长度
    11        length = len(data) + len(err_data)
    12        print('数据总长度: %s' % length)
```

#先需要发送报头信息，以下为创建报头信息（至第一次发送）

```
1 #需要添加时间信息
2 time_info = datetime.datetime.now()
3 #设置一个字典将一些额外的信息和长度信息放进去然后json序列化，报头字典
4 masthead = {}
5 #将时间数据放入报头字典中
6 masthead['time'] = str(time_info) #时间格式不能被json序列化，所以将其转化为字符串形式
7 masthead['length'] = length
```



weixin_34290390

关注

0

```

1 | #将报头字典json序列化 2 | json_masthead = json.dumps(masthead) #得到json格式的报头
3 | # 将json格式的报头编码成字节形式
4 | masthead_data = json_masthead.encode('utf-8')
5 | #利用struct将报头编码的字节的长度转成固定的字节(4个字节)
6 | masthead_length = struct.pack('i', len(masthead_data))

```

```

1 | #1.发送报头的长度（第一次发送）
2 | client.send(masthead_length)
3 | #2.发送报头信息(第二次发送)
4 | client.send(masthead_data)
5 | #3.发送真实数据（第三次发送）
6 | client.send(data)
7 | client.send(err_data)
8 | except ConnectionResetError:
9 |     print('客户端断开连接。。。')
10 | client.close()
11 | break
12 |
13 |

```

客户端

```

import socket
import struct
import json
client = socket.socket()
client.connect(('127.0.0.1', 33520))
while True:

```

```

1 | cmd = input('请输入cmd指令(Q\q退出)>>:').strip()
2 | if cmd == 'q':
3 |     break

```

```

1 | #发送CMD指令至服务器
2 | client.send(cmd.encode('utf-8'))

```

```

1 | #1.第一次接收，接收报头信息的长度，由于struct模块固定长度为4字节，括号内直接填4
2 | len_masthead = client.recv(4)
3 | #利用struct反解报头长度，由于是元组形式，取值得到整型数字masthead_length
4 | masthead_length = struct.unpack('i', len_masthead)[0]

```

```

1 | #2.第二次接收，接收报头信息，接收长度为报头长度masthead_Length 被编码成字节形式的json格式的字典，
2 | # 解字符串编码得到json格式的字典masthead_data
3 | masthead_data = client.recv(masthead_length).decode('utf-8')
4 | #得到报头字典masthead
5 | masthead = json.loads(masthead_data)
6 | print('执行时间%s' % masthead['time'])
7 | #通过报头字典得到数据长度
8 | data_length = masthead['length']

```



weixin_34290390

关注

0

```
1 #3.第三次接收，接收真实数据，真实数据长度为data_length
2 # data = client.recv(data_length) #有可能真实数据长度太大会撑爆内存。
3 #所以循环读取数据
4 all_data = b''
5 length = 0
6 #循环直到长度大于等于数据长度
7 while length < data_length:
8     data = client.recv(1024)
9     length += len(data)
10    all_data += data
11 print('数据的总长度: %s' % data_length)
```

```
1 #我的电脑是Windows系统，所以用gbk解码系统发出的信息
2 print(all_data.decode('gbk'))
```

复制代码

总结：




- 1.TCP协议中，会产生粘包现象。粘包现象产生本质就是读取数据长度未知。
 - 2.解决粘包现象本质就是处理读取数据长度。
 - 3.报头的作用就是解决数据传输过程中数据长度怎么计算传达和传输其他额外信息的。
- 原文地址<https://www.cnblogs.com/5j421/p/10574390.html>

文章知识点与官方知识档案匹配，可进一步学习相关知识

Python入门技能树 进阶语法 常用标准库 90957 人正在系统学习中

“相关推荐”对你有帮助么？

 非常没帮助  没帮助  一般  有帮助  非常有帮助

关于我们 招贤纳士 商务合作 寻求报道  400-660-0108  kefu@csdn.net  在线客服 工作时间 8:30-22:00

公安备案号11010502030143 京ICP备19004658号 京网文〔2020〕1039-165号 经营性网站备案信息 北京互联网违法和不良信息举报中心
家长监护 网络110报警服务 中国互联网举报中心 Chrome商店下载 账号管理规范 版权与免责声明 版权申诉 出版物许可证 营业执照

©1999-2022北京创新乐知网络技术有限公司



weixin_34290390

关注

👍 0