# TCP 粘包问题分析和解决（全）

在 socket 网络程序中，TCP 和 UDP 分别是面向连接和非面向连接的。因此 TCP 的 socket 编程，收发两端（客户端和服务器端）都要有成对的 socket，因此，发送端为了将多个发往接收端的包，更有效的发到对方，使用了优化方法（Nagle 算法），将多次间隔较小、数据量小的数据，合并成一个大的数据块，然后进行封包。这样，接收端，就难于分辨出来了，必须提供科学的拆包机制。

对于 UDP，不会使用块的合并优化算法，这样，实际上目前认为，是由于 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。所以 UDP 不会出现粘包问题。

==========

在介绍 TCP 之前先普及下两个相关的概念，长连接和短连接。

1.长连接

Client 方与 Server 方先建立通讯连接，连接建立后 不断开， 然后再进行报文发送和接收。

2.短连接

Client 方与 Server 每进行一次报文收发交易时才进行通讯连接，交易完毕后立即断开连接。此种方式常用于一点对多点通讯，比如多个 Client 连接一个 Server.

## TCP 协议简介

TCP 是一个面向连接的传输层协议，虽然 TCP 不属于 ISO 制定的协议集，但由于其在商业界和工业界的成功应用，它已成为事实上的网络标准，广泛应用于各种网络主机间的通信。

作为一个**面向连接的传输层协议**，TCP 的目标是为用户提供可靠的端到端连接，保证信息有序无误的传输。它除了提供基本的数据传输功能外，还为保证可靠性采用了数据编号、校验和计算、数据确认等一系列措施。它对传送的每个数据字节都进行编号，**并请求接收方回传确认信息（ACK）**。发送方如果在规定的时间内没有收到数据确认，就重传该数据。

（1） 数据编号使接收方能够处理数据的失序和重复问题。

（2） 数据误码问题通过在每个传输的数据段中增加校验和予以解决，接收方在接收到数据后检查校验和，若校验和有误，则丢弃该有误码的数据段，并要求发送方重传。

（3） 流量控制也是保证可靠性的一个重要措施，若无流控，可能会因接收缓冲区溢出而丢失大量数据，导致许多重传，造成网络拥塞恶性循环。

（4） TCP 采用可变窗口进行流量控制，由接收方控制发送方发送的数据量。

TCP 为用户提供了高可靠性的网络传输服务，但可靠性保障措施也影响了传输效率。因此，在实际工程应用中，只有关键数据的传输才采用 TCP，而普通数据的传输一般采用高效率的 UDP。

**保护消息边界和流**

那么什么是保护消息边界和流呢?

保护消息边界，就是指传输协议把数据当作一条独立的消息在网上传输，接收端只能接收独立的消息。也就是说存在保护消息边界，接收端一次只能接收发送端发出的一个数据包。而面向流则是指无保护**消息保护边界**的，如果发送端连续发送数据，接收端有可能在一次接收动作中，会接收两个或者更多的数据包。

例如，我们连续发送三个数据包，大小分别是 2k，4k ，8k,这三个数据包，都已经到达了接收端的网络堆栈中，如果使用 UDP 协议，不管我们使用多大的接收缓冲区去接收数据，我们必须有三次接收动作，才能够把所有的数据包接收完.而使用 **TCP 协议**，我们只要把接收的**缓冲区大小设置在 14k 以上**，我们就能够一次把所有的数据包接收下来，只需要有一次接收动作。

**注意：**

这就是因为 UDP 协议的保护消息边界使得每一个消息都是独立的。而流传输却把数据当作一串数据流，他不认为数据是一个一个的消息。所以有很多人在使用 tcp 协议通讯的时候，并不清楚 **tcp 是基于流的传输**，当连续发送数据的时候，他们时常会认识 tcp 会丢包。其实不然，因为当他们使用的缓冲区足够大时，他们有可能会一次接收到两个甚至更多的数据包，而很多人往往会忽视这一点，只解析检查了第一个数据包，而已经接收的其他数据包却被忽略了。所以大家如果要作这类的网络编程的时候,必须要注意这一点。

**结论：**

（1）TCP 为了保证可靠传输，尽量减少额外开销（每次发包都要验证），因此采用了流式传输，面向流的传输，相对于面向消息的传输，可以减少发送包的数量，从而减少了额外开销。但是，对于数据传输频繁的程序来讲，使用 TCP 可能会容易粘包。当然，对接收端的程序来讲，如果机器负荷很重，也会在接收缓冲里粘包。这样，就需要接收端额外拆包，增加了工作量。因此，这个特别适合的是数据要求可靠传输，但是不需要太频繁传输的场合（两次操作间隔 100ms，具体是由 TCP 等待发送间隔决定的，取决于内核中的 socket 的写法）

（2）UDP，由于面向的是消息传输，它把所有接收到的消息都挂接到缓冲区的接受队列中，因此，它对于数据的提取分离就更加方便，但是，它没有粘包机制，因此，当发送数据量较小的时候，就会发生数据包有效载荷较小的情况，也会增加多次发送的系统发送开销（系统调用，写硬件等）和接收开销。因此，应该**最好设置一个比较合适的数据包的包长，来进行 UDP 数据的发送。（UDP 最大载荷为 1472，因此最好能每次传输接近这个数的数据量，这特别适合于视频，音频等大块数据的发送，同时，通过减少握手来保证流媒体的实时性）**

==========

**粘包问题分析与对策**

TCP 粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。

出现粘包现象的原因是多方面的，它既可能由发送方造成，也可能由接收方造成。

**什么时候需要考虑粘包问题**

1 如果利用 tcp 每次发送数据，就与对方建立连接，然后双方发送完一段数据后，就关闭连接，这样就不会出现粘包问题（因为只有一种包结构,类似于 http 协议）。

关闭连接主要是要双方都发送 close 连接（参考 tcp 关闭协议）。如：A 需要发送一段字符串给 B，那么 A 与 B 建立连接，然后发送双方都默认好的协议字符如"hello give me sth about yourself"，然后 B 收到报文后，就将缓冲区数据接收，然后关闭连接，这样粘包问题不用考虑到，因为大家都知道是发送一段字符。

2 如果发送数据无结构，如文件传输，这样发送方只管发送，接收方只管接收存储就 ok，也不用考虑粘包 3 如果双方建立连接，需要在连接后一段时间内发送不同结构数据，如连接后，有好几种结构：

1)"hellogive me sth abour yourself"

2)"Don'tgive me sth abour yourself"

那这样的话，如果发送方连续发送这个两个包出去，接收方一次接收可能会是"hellogive me sth abour yourselfDon't give me sth abour yourself"这样接收方就傻了，到底是要干嘛？不知道，因为协议没有规定这么诡异的字符串，所以要处理把它分包，怎么分也需要双方组织一个比较好的包结构，所以一般可能会**在头加一个数据长度之类的包**，以确保接收。

## 粘包出现原因

简单得说，在流传输中出现，UDP 不会出现粘包，因为它有**消息边界**(参考 Windows 网络编程)

1 发送端需要等缓冲区满才发送出去，造成粘包

2 接收方不及时接收缓冲区的包，造成多个包接收

具体点：

（1）发送方引起的粘包是由 TCP 协议本身造成的，TCP 为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据。若连续几次发送的数据都很少，通常 TCP 会根据优化算法把这些数据合成一包后一次发送出去，这样接收方就收到了粘包数据。

（2）接收方引起的粘包是由于接收方用户进程不及时接收数据，从而导致粘包现象。这是因为接收方先把收到的数据放在系统接收缓冲区，用户进程从该缓冲区取数据，若下一包数据到达时前一包数据尚未被用户进程取走，则下一包数据放到系统接收缓冲区时就接到前一包数据之后，而用户进程根据预先设定的缓冲区大小从系统接收缓冲区取数据，这样就一次取到了多包数据。

粘包情况有两种，一种是粘在一起的包都是完整的数据包，另一种情况是粘在一起的包有不完整的包。

不是所有的粘包现象都需要处理，若传输的数据为不带结构的连续流数据（如文件传输），则不必把粘连的包分开（简称分包）。但在实际工程应用中，传输的数据一般为带结构的数据，这时就需要做分包处理。

在处理定长结构数据的粘包问题时，分包算法比较简单；在处理不定长结构数据的粘包问题时，分包算法就比较复杂。特别是粘在一起的包有不完整的包的粘包情况，由于一包数据内容被分在了两个连续的接收包中，处理起来难度较大。实际工程应用中应尽量避免出现粘包现象。

为了避免粘包现象，可采取以下几种措施：

（1）对于发送方引起的粘包现象，用户可通过编程设置来避免，TCP 提供了强制数据立即传送的操作指令 push，TCP 软件收到该操作指令后，就立即将本段数据发送出去，而不必等待发送缓冲区满；

（2）对于接收方引起的粘包，则可通过优化程序设计、精简接收进程工作量、提高接收进程优先级等措施，使其及时接收数据，从而尽量避免出现粘包现象；

（3）由接收方控制，将一包数据按结构字段，人为控制分多次接收，然后合并，通过这种手段来避免粘包。

以上提到的三种措施，都有其不足之处。

（1）第一种编程设置方法虽然可以避免发送方引起的粘包，但它关闭了优化算法，降低了网络发送效率，影响应用程序的性能，一般不建议使用。

（2）第二种方法只能减少出现粘包的可能性，但并不能完全避免粘包，当发送频率较高时，或由于网络突发可能使某个时间段数据包到达接收方较快，接收方还是有可能来不及接收，从而导致粘包。

（3）第三种方法虽然避免了粘包，但应用程序的效率较低，对实时应用的场合不适合。

一种比较周全的对策是：接收方创建一预处理线程，对接收到的数据包进行预处理，将粘连的包分开。对这种方法我们进行了实验，证明是高效可行的。

具体可以参考：http://blog.csdn.net/soli/article/details/1297109

TCP 无保护消息边界的解决

针对这个问题，一般有 3 种解决方案：

(1)发送固定长度的消息

(2)把消息的尺寸与消息一块发送

(3)使用特殊标记来区分消息间隔

其解决方法具体解决可以参考：http://blog.csdn.net/zhangxinrun/article/details/6721427

==========

网络通讯的封包和拆包

对于基于 TCP 开发的通讯程序，有个很重要的问题需要解决，就是封包和拆包。

为什么基于 TCP 的通讯程序需要进行封包和拆包

TCP 是个"流"协议，所谓流，就是没有界限的一串数据，大家可以想想河里的流水，是连成一片的,其间是没有分界线的。但一般通讯程序开发是需要定义一个个相互独立的数据包的，比如用于登陆的数据包，用于注销的数据包。由于 TCP"流"的特性以及网络状况，在进行数据传输时会出现以下几种情况。

假设我们连续调用两次 send 分别发送两段数据 data1 和 data2,在接收端有以下几种接收情况(当然不止这几种情况,这里只列出了有代表性的情况).

A.先接收到 data1,然后接收到 data2.

B.先接收到 data1 的部分数据,然后接收到 data1 余下的部分以及 data2 的全部.

C.先接收到了 data1 的全部数据和 data2 的部分数据,然后接收到了 data2 的余下的数据.

D.一次性接收到了 data1 和 data2 的全部数据.

对于 A 这种情况正是我们需要的,不再做讨论.对于 B,C,D 的情况就是大家经常说的"粘包",就需要我们把接收到的数据进行拆包,拆成一个个独立的数据包,为了拆包就必须在发送端进行封包。

另:对于 UDP 来说就不存在拆包的问题,因为 UDP 是个"数据包"协议,也就是两段数据间是有界限的,在接收端要么接收不到数据要么就是接收一个完整的一段数据,不会少接收也不会多接收。

## 为什么会出现 B.C.D 的情况

1.由 Nagle 算法造成的发送端的粘包:Nagle 算法是一种改善网络传输效率的算法.简单的说,当我们提交一段数据给 TCP 发送时,TCP 并不立刻发送此段数据,而是等待一小段时间,看看在等待期间是否还有要发送的数据,若有则会一次把这两段数据发送出去.这是对 Nagle 算法一个简单的解释,详细的请看相关书籍. C 和 D 的情况就有可能是 Nagle 算法造成的.

2.接收端接收不及时造成的接收端粘包:TCP 会把接收到的数据存在自己的缓冲区中,然后通知应用层取数据.当应用层由于某些原因不能及时的把 TCP 的数据取出来,就会造成 TCP 缓冲区中存放了几段数据.

## 怎样封包和拆包

最初遇到"粘包"的问题时,我是通过在两次 send 之间调用 sleep 来休眠一小段时间来解决。这个解决方法的缺点是显而易见的,使传输效率大大降低,而且也并不可靠。后来就是通过应答的方式来解决,尽管在大多数时候是可行的,但是不能解决 B 的那种情况,而且采用应答方式增加了通讯量,加重了网络负荷. 再后来就是对数据包进行封包和拆包的操作。

## 封包

封包就是给一段数据加上包头,这样一来数据包就分为包头和包体两部分内容了(以后讲过滤非法包时封包会加入"包尾"内容)。包头其实上是个大小固定的结构体,其中有个结构体成员变量表示包体的长度,这是个很重要的变量,其他的结构体成员可根据需要自己定义。根据包头长度固定以及包头中含有包体长度的变量就能正确的拆分出一个完整的数据包。

## 拆包

对于拆包目前我最常用的是以下两种方式：

（1）动态缓冲区暂存方式。之所以说缓冲区是动态的是因为当需要缓冲的数据长度超出缓冲区的长度时会增大缓冲区长度。

大概过程描述如下：

A,为每一个连接动态分配一个缓冲区,同时把此缓冲区和 SOCKET 关联,常用的是通过结构体关联.

B,当接收到数据时首先把此段数据存放在缓冲区中.

C,判断缓存区中的数据长度是否够一个包头的长度,如不够,则不进行拆包操作.

D,根据包头数据解析出里面代表包体长度的变量.

E,判断缓存区中除包头外的数据长度是否够一个包体的长度,如不够,则不进行拆包操作.

F,取出整个数据包.这里的"取"的意思是不光从缓冲区中拷贝出数据包,而且要把此数据包从缓存区中删除掉.删除的办法就是把此包后面的数据移动到缓冲区的起始地址.

这种方法有两个缺点.

1） 为每个连接动态分配一个缓冲区增大了内存的使用.

2）有三个地方需要拷贝数据,一个地方是把数据存放在缓冲区,一个地方是把完整的数据包从缓冲区取出来,一个地方是把数据包从缓冲区中删除.第二种拆包的方法会解决和完善这些缺点.

前面提到过这种方法的缺点.下面给出一个改进办法， 即采用环形缓冲.但是这种改进方法还是不能解决第一个缺点以及第一个数据拷贝,只能解决第三个地方的数据拷贝(这个地方是拷贝数据最多的地方).第 2 种拆包方式会解决这两个问题.

环形缓冲实现方案是定义两个指针,分别指向有效数据的头和尾.在存放数据和删除数据时只是进行头尾指针的移动.

（2）利用底层的缓冲区来进行拆包

由于 TCP 也维护了一个缓冲区,所以我们完全可以利用 TCP 的缓冲区来缓存我们的数据，这样一来就不需要为每一个连接分配一个缓冲区了。另一方面我们知道 recv 或者 wsarecv 都有一个参数,用来表示我们要接收多长长度的数据。利用这两个条件我们就可以对第一种方法进行优化。

对于阻塞 SOCKET 来说，我们可以利用一个循环来接收包头长度的数据，然后解析出代表包体长度的那个变量，再用一个循环来接收包体长度的数据。

编程实现见：http://blog.csdn.net/zhangxinrun/article/details/6721495

---

这个问题产生于编程中遇到的几个问题：

1、使用 TCP 的 Socket 发送数据的时候，会出现发送出错，WSAEWOULDBLOCK，在 TCP 中不是会保证发送的数据能够安全的到达接收端的吗？也有窗口机制去防止发送速度过快,为什么还会出错呢？

2、TCP 协议，在使用 Socket 发送数据的时候，每次发送一个包，接收端是完整的接受到一个包还是怎么样？如果是每发一个包，就接受一个包，为什么还会出现粘包问题，具体是怎么运行的？

3、关于 Send，是不是只有在非阻塞状态下才会出现实际发送的比指定发送的小？在阻塞状态下会不会出现实际发送的比指定发送的小，就是说只能出现要么全发送，要么不发送？在非阻塞状态下，如果之发送了一些数据，要怎么处理，调用了 Send 函数后，发现返回值比指定的要小，具体要怎么做？

4、最后一个问题，就是 TCP/IP 协议和 Socket 是什么关系？是指具体的实现上，Socket 是 TCP/IP 的实现？那么为什么会出现使用 TCP 协议的 Socket 会发送出错。

---

这个问题第 1 个回答：

1 应该是你的缓冲区不够大，

2 tcp 是流,没有界限.也就没所谓的包.

3 阻塞也会出现这种现象,出现后继续发送没发送出去的.

4tcp是协议,socket是一种接口,没必然联系.错误取决于你使用接口的问题,跟tcp没关系.

---

这个问题第 2 个回答:

1、应该不是缓冲区大小问题，我试过设置缓冲区大小，不过这里有个问题，就是就算我把缓冲区设置成几 G，也返回成功，不过实际上怎么可能设置那么大

3、出现没发送完的时候要手动发送吧，有没有具体的代码实现?

4、当选择 TCP 的 Socket 发送数据的时候,TCP 中的窗口机制不是能防止发送速度过快的吗?为什么 Socket 在出现了 WSAEWOULDBLOCK 后没有处理?

---

这个问题第 3 个回答:

1.在使用非阻塞模式的情况下，如果系统发送缓冲区已满，并示及时发送到对端，就会产生该错误，继续重试即可。

3.如果没有发完就继续发送后续部分即可。

---

这个问题第 4 个回答:

1、使用非阻塞模式时，如果当前操作不能立即完成则会返回失败，错误码是 WSAEWOULDBLOCK，这是正常的，程序可以先执行其它任务，过一段时间后再重试该操作。

2、发送与接收不是一一对应的，TCP 会把各次发送的数据重新组合，可能合并也可能拆分，但发送次序是不变的。

3、在各种情况下都要根据 send 的返回值来确定发送了多少数据，没有发送完就再接着发。

4、socket 是 Windows 提供网络编程接口，TCP/IP 是网络传输协议，使用 socket 是可以使用多种协议，其中包括 TCP/IP。

---

这个问题第 5 个回答：

发送的过程是：发送到缓冲区和从缓冲区发送到网络上

WSAEWOULDBLOCK 和粘包都是出现在发送到缓冲区这个过程的

---

## Socket 编程（异步通讯,解决 Tcp 粘包）

前面提到，TCP 会出现粘包问题，下面将以实例演示解决方案：

问题一般会出现的情况如下,假设我们连续发送两条两天记录("我是 liger_zql")：

模拟发送示例：

```
#region 测试消息发送,并匹配协议
 TcpClient client =new TcpClient();
 client.AsynConnect();
 Console.WriteLine("下面将连续发送 2 条测试消息...");
 Console.ReadKey();
 MessageProtocol msgPro;
  for (int i = 0; i<2; i++)
   {
     msgPro =newMessageProtocol("我是 liger_zql");
     Console.WriteLine("第{0}条: {1}", i +1,msgPro.MessageInfo.Content);
```

```
    client.AsynSend(msgPro);
  }
 #endregion
```

接收端接受两条信息会出现如下三种情况：

1.(1)我是 liger_zql(2)我是 liger_zql

2.(1)我是 liger_zql 我是(2)liger_zql

3.(1)我是 liger_zql 我是 liger_zql

通过以上三种情况，显然 2、3 都不是我们想要的结果。那么如何处理这中情况呢？

解决方案：通过自定义协议...

我们可以以将信息以 xml 的格式发送出去,列入<protocol>content</protocol>通过正则匹配信息是否完整，如果不完整，我们可以先将本次接受信息缓存接受下一次信息，再次匹配得到相应的结果。

（1）将信息对象转换成一定格式的 xml 字符串：

（2）对接收的信息通过正则进行匹配处理：

（3）将该定义的协议换换成信息对象，通过对象获取自己想要的信息。