解决 TCP 网络传输“粘包”问题 2015-09-28 11:04:13

当前在网络传输应用中，广泛采用的是 TCP/IP 通信协议及其标准的 socket 应用开发编程接口（API）。TCP/IP 传输层有两个并列的协议：TCP 和 UDP。其中 TCP（transport control protocol，传输控制协议）是面向连接的，提供高可靠性服务。UDP（user datagram protocol，用户数据报协议）是无连接的，提供高效率服务。在实际工程应用中，对可靠性和效率的选择取决于应用的环境和需求。一般情况下，普通数据的网络传输采用高效率的 udp，重要数据的网络传输采用高可靠性的 TCP。

在应用开发过程中，笔者发现基于 TCP 网络传输的应用程序有时会出现粘包现象（即发送方发送的若干包数据到接收方接收时粘成一包）。针对这种情况，我们进行了专题研究与实验。本文重点分析了 TCP 网络粘包问题，并结合实验结果提出了解决该问题的对策和方法，供有关工程技术人员参考。

一、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。数据编号使接收方能够处理数据的失序和重复问题。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。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。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。

二、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。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。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（图 1 所示）。

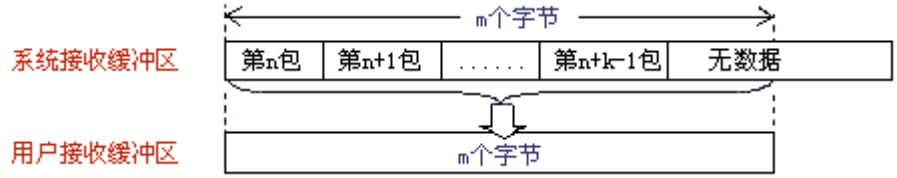


图 1

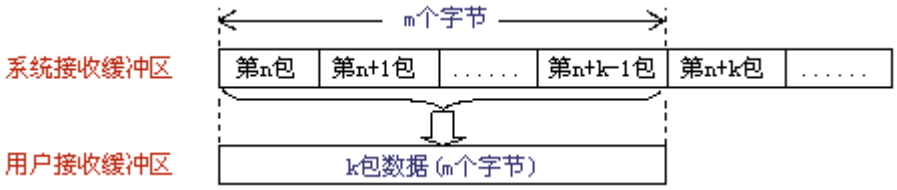


图 2

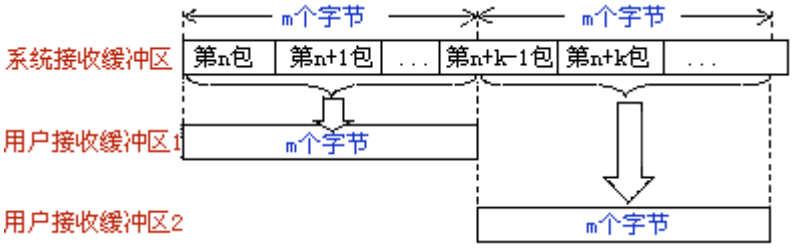


图 3

粘包情况有两种，一种是粘在一起的包都是完整的数据包（图 1、图 2 所示），另一种情况是粘在一起的包有不完整的包（图 3 所示），此处假设用户接收缓冲区长度为 m 个字节。

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。特别是如图 3 所示的粘包情况，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。一是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；二是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；三是由接收方控制，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。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。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。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。

三、编程与实现

1．实现框架

实验网络通信程序采用 TCP/IP 协议的 socket api 编程实现。socket 是面向客户机/服务器模型的。TCP 实现框架如图 4 所示。

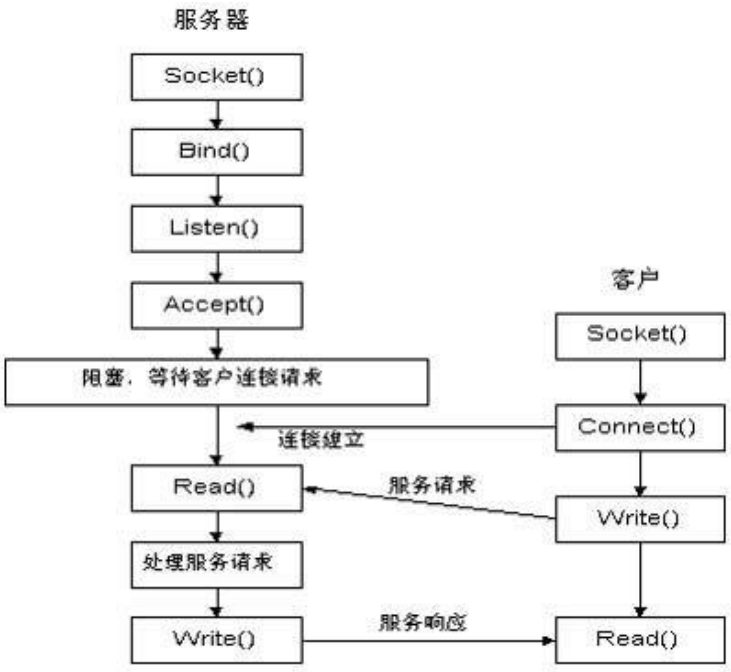


图 4

2．实验硬件环境：

服务器：pentium 350 微机

客户机：pentium 166 微机

网络平台：由 10 兆共享式 hub 连接而成的局域网

3．实验软件环境：

操作系统：windows 98

编程语言：visual c++ 5.0

4．主要线程

编程采用多线程方式，服务器端共有两个线程：发送数据线程、发送统计显示线程。客户端共有三个线程：接收数据线程、接收预处理粘包线程、接收统计显示线程。其中，发送和接收线程优先级设为 thread_priority_time_critical（最高优先级），预处理线程优先级为 thread_priority_above_normal（高于普通优先级），显示线程优先级为 thread_priority_normal（普通优先级）。

实验发送数据的数据结构如图 5 所示：

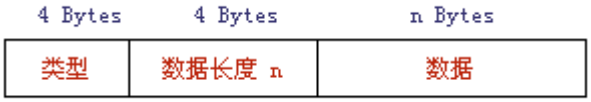


图 5

5．分包算法

针对三种不同的粘包现象，分包算法分别采取了相应的解决办法。其基本思路是首先将待处理的接收数据流（长度设为 m）强行转换成预定的结构数据形式，并从中取出结构数据长度字段，即图 5 中的 n，而后根据 n 计算得到第一包数据长度。

1)若 n

2)若 n=m，则表明数据流内容恰好是一完整结构数据，直接将其存入临时缓冲区即可。

3)若 n>m，则表明数据流内容尚不够构成一完整结构数据，需留待与下一包数据合并后再行处理。

对分包算法具体内容及软件实现有兴趣者，可与作者联系。

四、实验结果分析

实验结果如下：

1．在上述实验环境下，当发送方连续发送的若干包数据长度之和小于 1500b 时，常会出现粘包现象，接收方经预处理线程处理后能正确解开粘在一起的包。若程序中设置了“发送不延迟”：（setsockopt (socket_name，ipproto_tcp，tcp_nodelay，(char *) &on，sizeof on) ，其中 on=1），则不存在粘包现象。

2．当发送数据为每包 1kb～2kb 的不定长数据时，若发送间隔时间小于 10ms，偶尔会出现粘包，接收方经预处理线程处理后能正确解开粘在一起的包。

3．为测定处理粘包的时间，发送方依次循环发送长度为 1.5kb、1.9kb、1.2kb、1.6kb、1.0kb 数据，共计 1000 包。为制造粘包现象，接收线程每次接收前都等待 10ms，接收缓冲区设为 5000b，结果接收方收到 526 包数据，其中长度为 5000b 的有 175 包。经预处理线程处理可得到 1000 包正确数据，粘包处理总时间小于 1ms。

实验结果表明，TCP 粘包现象确实存在，但可通过接收方的预处理予以解决，而且处理时间非常短（实验中 1000 包数据总共处理时间不到 1ms），几乎不影响应用程序的正常工作。

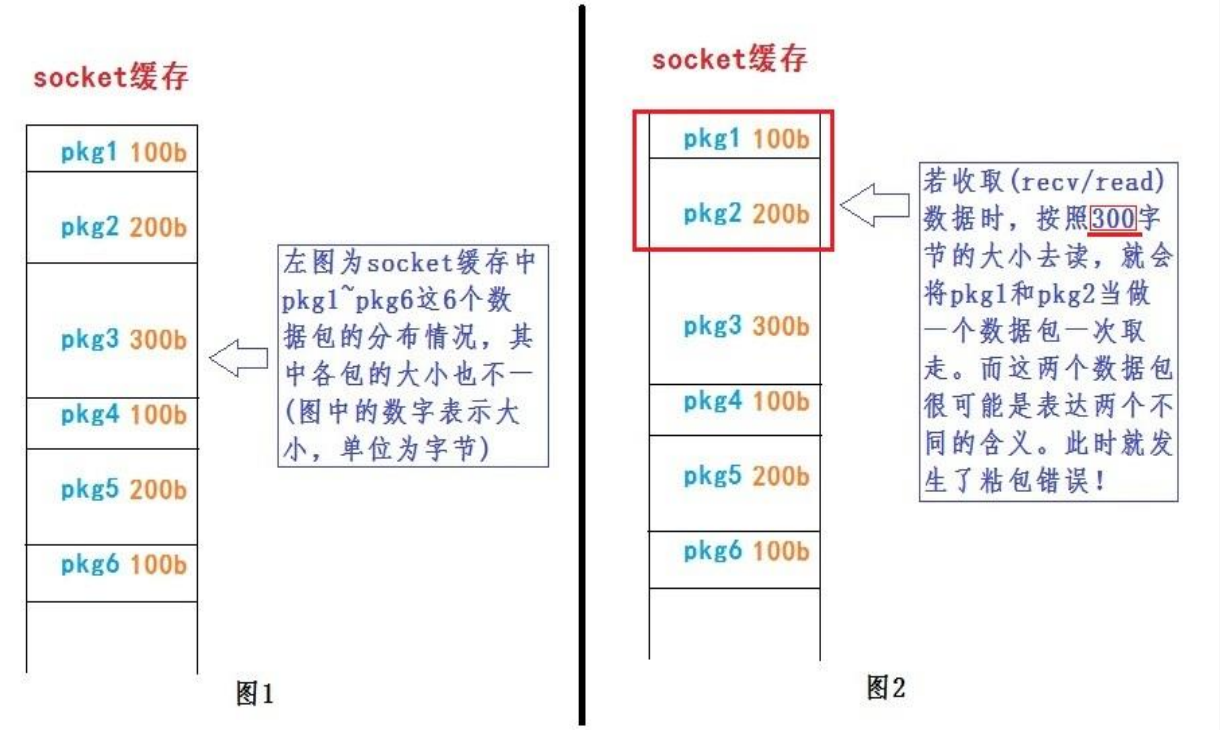
第一章 粘包问题概述

1.1 描述背景

采用 TCP 协议进行网络数据传送的软件设计中，普遍存在粘包问题。这主要是由于现代操作系统的网络传输机制所产生的。我们知道，网络通信采用的套接字(socket)技术，其实现实际是由系统内核提供一片连续缓存(流缓冲)来实现应用层程序与网卡接口之间的中转功能。多个数据包被连续存储于连续的缓存中，在对数据包进行读取时由于无法确定发生方的发送边界，而采用某一估测值大小来进行数据读出，若双方的 size 不一致时就会使数据包的边界发生错位，导致读出错误的数据分包，进而曲解原始数据含义。

1.2 粘包的概念

粘包问题的本质就是数据读取边界错误所致，通过下图可以形象地理解其现象。



如图 1 所示，当前的 socket 缓存中已经有 6 个数据分组到达，其大小如图中数字。而应用程序在对数据进行收取时(如图 2)，采用了 300 字节的要求去读取，则会误将 pkg1 和 pkg2 一起收走当做一个包来处理。而实际上，很可能 pkg1 是一个文本文件的内容，而 pkg2 则可能是一个音频内容，这风马牛不相及的两个数据包却被揉进一个包进行处理，显然有失妥当。严重时可能因为丢了 pkg2 而导致软件陷入异常分支产生乌龙事件。

因此，**粘包问题必须引起所有软件设计者（项目经理）的高度重视**！

那么，或许会有读者发问，为何不让接收程序按照 100 字节来读取呢？我想如果您了解一些 TCP 编程的话就不会有这样的问题。网络通信程序中，数据包通常是不能确定大小的，尤其在软件设计阶段无法真的做到确定为一个固定值。比如聊天软件客户端若采用 TCP 传输一个用户名和密码到服务端进行验证登陆，我想这个数据包不过是几十字节，至多几百字节即可发送完毕，而有时候要传输一个很大的视频文件，即使分包发送也应该一个包在几千字节吧。（据说，某国电信平台的 MW 中见到过一次发送 1.5 万字节的电话数据）这种情况下，发送数据的分包大小无法固定，接收端也就无法固定。所以一般采用一个较为合理的预估值进行轮询接收。（网卡的 MTU 都是 1500 字节，因此这个预估值一般为 MTU 的 1~3 倍）。

相信读者对粘包问题应该有了初步认识了。

第二章 粘包回避设计

2.0 闲扯

作者在此提出三种可解之法，这都是从软件设计的角度去考虑的，当然代码实现也是可以验证没问题的。下面一一为读者解开其谜底。

读者在别的文献中还能看到一种叫做【短连接】的方法，根据经验不建议采用此法，开销太大得不偿失。故而本文对该方案不做解释。

2.1 设计方案一：定长发送

在进行数据发送时采用固定长度的设计，也就是无论多大数据发送都分包为固定长度（为便于描述，此处定长为记为 LEN），也就是发送端在发送数据时都以 LEN 为长度进行分包。这样接收方都以固定的 LEN 进行接收，如此一来发送和接收就能一一对应了。分包的时候不一定能完整的恰好分成多个完整的 LEN 的包，最后一个包一般都会小于 LEN，这时候最后一个包可以在不足的部分填充空白字节。

当然，这种方法会有缺陷。1. 最后一个包的不足长度被填充为空白部分，也即无效字节序。那么接收方可能难以辨别这无效的部分，它本身就是为了补位的，并无实际含义。这就为接收端处理其含义带来了麻烦。当然也有解决办法，可以通过增添标志位的方法来弥补，即在每一个数据包的最前面增加一个定长的报头，然后将该数据包的末尾标记一并发送。接收方根据这个标记确认无效字节序列，从而实现数据的完整接收。2. 在发送包长度随机分布的情况下，会造成带宽浪费。比如发送长度可能为 1, 100, 1000, 4000 字节等等，则都需要按照定长最大值即 4000 来发送，数据包小于 4000 字节的其他包也会被填充至 4000，造成网络负载的无效浪费。

综上，此方案适在发送数据包长度较为稳定(趋于某一固定值)的情况下有较好的效果。

2.2 设计方案二：尾部标记序列

在每个要发送的数据包的尾部设置一个特殊的字节序列，此序列带有特殊含义，跟字符串的结束符标识"\0"一样的含义，用来标示这个数据包的末尾，接收方可对接收的数据进行分析，通过尾部序列确认数据包的边界。

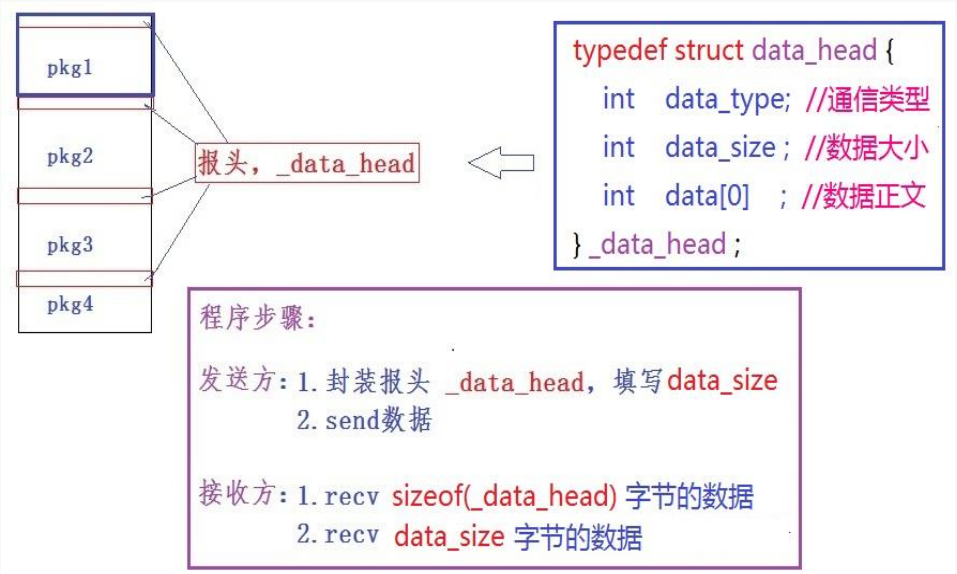
这种方法的缺陷较为明显：1.接收方需要对数据进行分析，甄别尾部序列。2.尾部序列的确定本身是一个问题。什么样的序列可以向"\0"一样来做一个结束符呢？这个序列必须是不具备通常任何人类或者程序可识别的带含义的数据序列，就像"\0"是一个无效字符串内容，因而可以作为字符串的结束标记。那普通的网络通信中，这个序列是什么呢？我想一时间很难找到恰当的答案。

2.3 设计方案三：头部标记分步接收

这个方法是作者有限学识里最好的办法了。它既不损失效率，还完美解决了任何大小的数据包的边界问题。

这个方法的实现是这样的，定义一个用户报头，在报头中注明每次发送的数据包大小。接收方每次接收时先以报头的 size 进行数据读取，这必然只能读到一个报头的数据，从报头中得到该数据包的数据大小，然后再按照此大小进行再次读取，就能读到数据的内容了。这样一来，每个数据包发送时都封装一个报头，然后接收方分两次接收一个包，第一次接收报头，根据报头大小第二次才接收数据内容。（此处的 `data[0]`的本质是一个指针，指向数据的正文部分，也可以是一篇连续数据区的起始位置。因此可以设计成 `data[user_size]`，这样的话。）

下面通过一个图来展现设计思想。



由图看出，数据发送多了封装报头的动作；接收方将每个包的接收拆分成了两次。

这方案看似精妙，实则也有缺陷：1.报头虽小，但每个包都需要多封装 sizeof(_data_head)的数据，积累效应也不可完全忽略。2.接收方的接收动作分成了两次，也就是进行数据读取的操作被增加了一倍，而数据读取操作的 recv 或者 read 都是系统调用，这对内核而言的开销是一个不能完全忽略的影响，对程序而言性能影响可忽略（系统调用的速度非常快）。

优点：避免了程序设计的复杂性，其有效性便于验证，对软件设计的稳定性要求来说更容易达标。综上，方案三乃上上策！

我选一，有什么潜在问题吗？

如果发送长度不确定的话，定长会造成短包被动拉伸填充，浪费带宽，且不便于找到尾部边界

这两天看 csdn 有一些关于 socket 粘包，socket 缓冲区设置的问题，发现自己不是很清楚，所以查资料了解记录一下：

一 .两个简单概念长连接与短连接：

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.

二 .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 abour 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)"hello give me sth abour yourself"

2)"Don't give 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"hello give 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。

三 .粘包出现原因：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

解决办法：

为了避免粘包现象，可采取以下几种措施。一是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；二是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；三是由接收方控制，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。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。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。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。

载自：http://blog.csdn.NET/binghuazh/archive/2009/05/28/4222516.aspx

==========

网络通讯的封包和拆包 收藏

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.

一.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.

二.为什么会出现 B.C.D 的情况.
"粘包"可发生在发送端也可发生在接收端.
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.象 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.

三.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,但是不能解决象 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.
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.
相关代码如下:

```
char PackageHead[1024];
char PackageContext[1024*20];

int len;
PACKAGE_HEAD *pPackageHead;
while( m_bClose == false )
{
memset(PackageHead,0,sizeof(PACKAGE_HEAD));
len = m_TcpSock.ReceiveSize((char*)PackageHead,sizeof(PACKAGE_HEAD));
if( len == SOCKET_ERROR )
{
    break;
}
if(len == 0)
{
    break;
}
pPackageHead = (PACKAGE_HEAD *)PackageHead;
memset(PackageContext,0,sizeof(PackageContext));
if(pPackageHead->nDataLen>0)
{
len = m_TcpSock.ReceiveSize((char*)PackageContext,pPackageHead->nDataLen);
}
        }
```

m_TcpSock 是一个封装了 SOCKET 的类的变量,其中的 ReceiveSize 用于接收一定长度的数据,直到接收了一定长度的数据或者网络出错才返回.

```
int winSocket::ReceiveSize( char* strData, int iLen )
{
if( strData == NULL )
return ERR_BADPARAM;
char *p = strData;
int len = iLen;
int ret = 0;
int returnlen = 0;
while( len > 0)
{
ret = recv( m_hSocket, p+(iLen-len), iLen-returnlen, 0 );
if ( ret == SOCKET_ERROR || ret == 0 )
{

return ret;
}

len -= ret;
returnlen += ret;
}

return returnlen;
}
```

对于非阻塞的 SOCKET,比如完成端口,我们可以提交接收包头长度的数据的请求,当 GetQueuedCompletionStatus 返回时,我们判断接收的数据长度是否等于包头长度,若等于,则提交接收包体长度的数据的请求,若不等于则提交接收剩余数据的请求.当接收包体时,采用类似的方法.

==========

几个问题：http://www.qqgb.com/Program/VC/VCJQ/Program_200509.html

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（又回到第一个问题了，汗一个）

实在是有点晕了，如果我的问题有不清楚的地方，或者分数有问题，欢迎指出，谢谢

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.也就所所谓的包.

3 阻塞也会出现这种现象,出现后继续发送没发送出去的.

4 tcp 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 应该是你的缓冲区不够大,

2 tcp 是流,没有界限.也就无所谓包.

3 阻塞也会出现这种现象,出现后继续发送没发送出去的.

4 tcp 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 3 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 4 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 5 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 6 个回答：

up

这个问题第 7 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