**关于Tcp封包**

很多朋友已经对此作了不少研究，也花费不少心血编写了实现代码和blog文档。当然也充斥着一些各式的评论，自己看了一下，总结一些心得。

首先我们学习一下这些朋友的心得，他们是：

<http://blog.csdn.net/stamhe/article/details/4569530>

<http://www.cppblog.com/tx7do/archive/2011/05/04/145699.html>

//………………

当然还有太多，很多东西粘来粘区也不知道到底是谁的原作,

看这些朋友的blog是我建议亲自看一下TCP-IP详解卷1中的相关内容【原理性的内容一定要看】。

**TCP大致工作原理介绍：**

**工作原理**

TCP-IP详解卷1第17章中17.2节对TCP服务原理作了一个简明介绍(以下蓝色字体摘自《TCP-IP详解卷1第17章17.2节》)：

尽管T C P和U D P都使用相同的网络层（ I P），T C P却向应用层提供与U D P完全不同的服务。T C P提供一种面向连接的、可靠的字节流服务。

面向连接意味着两个使用T C P的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须先建立一个T C P连接。这一过程与打电话很相似，先拨号振铃，等待对方摘机说“喂”，然后才说明是谁。在第1 8章我们将看到一个T C P连接是如何建立的，以及当一方通信结束后如何断开连接。

在一个T C P连接中，仅有两方进行彼此通信。在第1 2章介绍的广播和多播不能用于T C P。

T C P通过下列方式来提供可靠性：

• 应用数据被分割成T C P认为最适合发送的数据块。这和U D P完全不同，应用程序产生的数据报长度将保持不变。由T C P传递给I P的信息单位称为报文段或段（ s e g m e n t）（参见图1 - 7）。在1 8 . 4节我们将看到T C P如何确定报文段的长度。

• 当T C P发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。在第2 1章我们将了解T C P协议中自适应的超时及重传策略。

• 当T C P收到发自T C P连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒，这将在1 9 . 3节讨论。

• T C P将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错， T C P将丢弃这个报文段和不确认收到此报文段（希望发端超时并重发）。

• 既然T C P报文段作为I P数据报来传输，而I P数据报的到达可能会失序，因此T C P报文段的到达也可能会失序。如果必要， T C P将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。

• 既然I P数据报会发生重复， T C P的接收端必须丢弃重复的数据。

• T C P还能提供流量控制。T C P连接的每一方都有固定大小的缓冲空间。T C P的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。两个应用程序通过T C P连接交换8 bit字节构成的字节流。T C P不在字节流中插入记录标识符。我们将这称为字节流服务（ byte stream service）。如果一方的应用程序先传1 0字节，又传2 0字节，再传5 0字节，连接的另一方将无法了解发方每次发送了多少字节。收方可以分4次接收这8 0个字节，每次接收2 0字节。一端将字节流放到T C P连接上，同样的字节流将出现在T C P连接的另一端。另外，T C P对字节流的内容不作任何解释。T C P不知道传输的数据字节流是二进制数据，还是A S C I I字符、E B C D I C字符或者其他类型数据。对字节流的解释由T C P连接双方的应用层解释。这种对字节流的处理方式与U n i x操作系统对文件的处理方式很相似。U n i x的内核对一个应用读或写的内容不作任何解释，而是交给应用程序处理。对U n i x的内核来说，它无法区分一个二进制文件与一个文本文件。

**T C P如何确定报文段的长度**

       我仍然引用官方解释《TCP-IP详解卷1》第18章18.4节：

最大报文段长度（ M S S）表示T C P传往另一端的最大块数据的长度。当一个连接建立时【三次握手】，连接的双方都要通告各自的M S S。我们已经见过M S S都是1 0 2 4。这导致I P数据报通常是4 0字节长：2 0字节的T C P首部和2 0字节的I P首部。

在有些书中，将它看作可“协商”选项。它并不是任何条件下都可协商。当建立一个连

接时，每一方都有用于通告它期望接收的M S S选项（M S S选项只能出现在S Y N报文段中）。如果一方不接收来自另一方的M S S值，则M S S就定为默认值5 3 6字节（这个默认值允许2 0字节的I P首部和2 0字节的T C P首部以适合5 7 6字节I P数据报)。

一般说来，如果没有分段发生， M S S还是越大越好（这也并不总是正确，参见图2 4 - 3和图2 4 - 4中的例子）。报文段越大允许每个报文段传送的数据就越多，相对I P和T C P首部有更高的网络利用率。当T C P发送一个S Y N时，或者是因为一个本地应用进程想发起一个连接，或者是因为另一端的主机收到了一个连接请求，它能将M S S值设置为外出接口上的M T U长度减去固定的I P首部和T C P首部长度。对于一个以太网， M S S值可达1 4 6 0字节。使用IEEE 802.3的封装（参见2 . 2节），它的M S S可达1 4 5 2字节。

如果目的I P地址为“非本地的( n o n l o c a l )”，M S S通常的默认值为5 3 6。而区分地址是本地还是非本地是简单的，如果目的I P地址的网络号与子网号都和我们的相同，则是本地的；如果目的I P地址的网络号与我们的完全不同，则是非本地的；如果目的I P地址的网络号与我们的相同而子网号与我们的不同，则可能是本地的，也可能是非本地的。大多数T C P实现版都提供了一个配置选项（附录E和图E - 1），让系统管理员说明不同的子网是属于本地还是非本地。这个选项的设置将确定M S S可以选择尽可能的大（达到外出接口的M T U长度）或是默认值5 3 6。

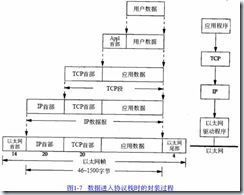
M S S让主机限制另一端发送数据报的长度。加上主机也能控制它发送数据报的长度，这将使以较小M T U连接到一个网络上的主机避免分段。

只有当一端的主机以小于5 7 6字节的M T U直接连接到一个网络中，避免这种分段才会有效。

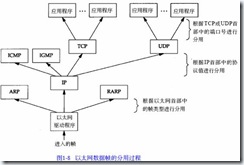
如果两端的主机都连接到以太网上，都采用5 3 6的M S S，但中间网络采用2 9 6的M T U，也将会

出现分段。使用路径上的M T U发现机制（参见2 4 . 2节）是关于这个问题的唯一方法。

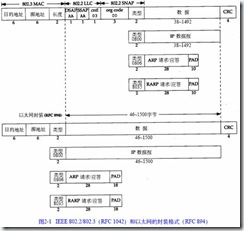
       以上说明MSS的值可以通过协商解决，这个协商过程会涉及MTU的值的大小，前面说了：【MSS=外出接口上的MTU-IP首部-TCP首部】，我们来看看数据进入TCP协议栈的封装过程：

[](http://images.cnblogs.com/cnblogs_com/jiangtong/201203/201203221733115740.jpg)

最后一层以太网帧的大小应该就是我们的出口ＭＴＵ大小了。当目的主机收到一个以太网数据帧时，数据就开始从协议栈中由底向上升，同时去掉各层协议加上的报文首部。每层协议盒都要去检查报文首部中的协议标识，以确定接收数据的上层协议。这个过程称作分用（ D e m u l t i p l e x i n g），图1 - 8显示了该过程是如何发生的。

[](http://images.cnblogs.com/cnblogs_com/jiangtong/201203/201203221733178493.jpg)

那么什么是MTU呢，这实际上是数据链路层的一个概念，以太网和802.3这两种局域网技术标准都对“链路层”的数据帧有大小限制：

[](http://images.cnblogs.com/cnblogs_com/jiangtong/201203/201203221733203735.jpg)

         最大传输单元MTU

正如在图2 - 1看到的那样，以太网和8 0 2 . 3对数据帧的长度都有一个限制，其最大值分别是1 5 0 0和1 4 9 2字节。链路层的这个特性称作M T U，最大传输单元。不同类型的网络大多数都有一个上限。

如果I P层有一个数据报要传，而且数据的长度比链路层的M T U还大，那么I P层就需要进行分片（ f r a g m e n t a t i o n），把数据报分成若干片，这样每一片都小于M T U。我们将在11 . 5节讨论I P分片的过程。

图2 - 5列出了一些典型的M T U值，它们摘自RFC 1191[Mogul and Deering 1990]。点到点的链路层（如S L I P和P P P）的M T U并非指的是网络媒体的物理特性。相反，它是一个逻辑限制，目的是为交互使用提供足够快的响应时间。在2 . 1 0节中，我们将看到这个限制值是如何计算出来的。在3 . 9节中，我们将用n e t s t a t命令打印出网络接口的M T U。

         路径MTU

当在同一个网络上的两台主机互相进行通信时，该网络的M T U是非常重要的。但是如果

两台主机之间的通信要通过多个网络，那么每个网络的链路层就可能有不同的M T U。重要的

不是两台主机所在网络的M T U的值，重要的是两台通信主机路径中的最小M T U。它被称作路

径M T U。

两台主机之间的路径M T U不一定是个常数。它取决于当时所选择的路由。而选路不一定

是对称的（从A到B的路由可能与从B到A的路由不同），因此路径M T U在两个方向上不一定是

一致的。

RFC 1191[Mogul and Deering 1990]描述了路径M T U的发现机制，即在任何时候确定路径

M T U的方法。我们在介绍了I C M P和I P分片方法以后再来看它是如何操作的。在11 . 6节中，我

们将看到I C M P的不可到达错误就采用这种发现方法。在11 . 7节中，还会看到， t r a c e r o u t e程序

也是用这个方法来确定到达目的节点的路径M T U。在11 . 8节和2 4 . 2节，将介绍当产品支持路

径M T U的发现方法时，U D P和T C P是如何进行操作的。

**ＴＣＰ的超时与重传**

前面谈到TCP如何保证传输可靠性是说到“当T C P发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段”，下面我看一下TCP的超时与重传。

T C P提供可靠的运输层。它使用的方法之一就是确认从另一端收到的数据。但数据和确认都有可能会丢失。T C P通过在发送时设置一个定时器来解决这种问题。如果当定时器溢出时还没有收到确认，它就重传该数据。对任何实现而言，关键之处就在于超时和重传的策略，即怎样决定超时间隔和如何确定重传的频率。

对每个连接，T C P管理4个不同的定时器。

1) 重传定时器使用于当希望收到另一端的确认。

2) 坚持( p e r s i s t )定时器使窗口大小信息保持不断流动，即使另一端关闭了其接收窗口。

3) 保活( k e e p a l i v e )定时器可检测到一个空闲连接的另一端何时崩溃或重启。

4) 2MSL定时器测量一个连接处于T I M E \_ WA I T状态的时间。

T C P超时与重传中最重要的部分就是对一个给定连接的往返时间（ RT T）的测量。由于路由器和网络流量均会变化，因此我们认为这个时间可能经常会发生变化， T C P应该跟踪这些变化并相应地改变其超时时间。

大多数源于伯克利的T C P实现在任何时候对每个连接仅测量一次RT T值。在发送一个报文段时，如果给定连接的定时器已经被使用，则该报文段不被计时。

具体RTT值的估算比较麻烦，需要可以参考《TCP-IP详解卷1第21章》

**TCP经受延时的确认**

交互数据总是以小于最大报文段长度的分组发送。对于这些小的报文段，接收方使用经受时延的确认方法来判断确认是否可被推迟发送，以便与回送数据一起发送。这样通常会减少报文段的数目       。

通常T C P在接收到数据时并不立即发送A C K；相反，它推迟发送，以便将A C K与需要沿该方向发送的数据一起发送（有时称这种现象为数据捎带A C K）。绝大多数实现采用的时延为200 ms，也就是说，T C P将以最大200 ms 的时延等待是否有数据一起发送。

我们看看另一位朋友的blog对此的介绍：

摘要：当使用TCP传输小型数据包时，程序的设计是相当重要的。如果在设计方案中不对TCP数据包的   
延迟应答，Nagle算法，Winsock缓冲作用引起重视，将会严重影响程序的性能。这篇文章讨论了这些   
问题，列举了两个案例，给出了一些传输小数据包的优化设计方案。

背景：当Microsoft TCP栈接收到一个数据包时，会启动一个200毫秒的计时器。当ACK确认数据包   
发出之后，计时器会复位，接收到下一个数据包时，会再次启动200毫秒的计时器。为了提升应用程序   
在内部网和Internet上的传输性能，Microsoft TCP栈使用了下面的策略来决定在接收到数据包后   
什么时候发送ACK确认数据包：   
1、如果在200毫秒的计时器超时之前，接收到下一个数据包，则立即发送ACK确认数据包。   
2、如果当前恰好有数据包需要发给ACK确认信息的接收端，则把ACK确认信息附带在数据包上立即发送。   
3、当计时器超时，ACK确认信息立即发送。   
为了避免小数据包拥塞网络，Microsoft TCP栈默认启用了Nagle算法，这个算法能够将应用程序多次   
调用Send发送的数据拼接起来，当收到前一个数据包的ACK确认信息时，一起发送出去。下面是Nagle   
算法的例外情况：   
1、如果Microsoft TCP栈拼接起来的数据包超过了MTU值，这个数据会立即发送，而不等待前一个数据   
包的ACK确认信息。在以太网中，TCP的MTU(Maximum Transmission Unit)值是1460字节。   
2、如果设置了TCP\_NODELAY选项，就会禁用Nagle算法，应用程序调用Send发送的数据包会立即被   
投递到网络，而没有延迟。   
为了在应用层优化性能，Winsock把应用程序调用Send发送的数据从应用程序的缓冲区复制到Winsock   
内核缓冲区。Microsoft TCP栈利用类似Nagle算法的方法，决定什么时候才实际地把数据投递到网络。   
内核缓冲区的默认大小是8K，使用SO\_SNDBUF选项，可以改变Winsock内核缓冲区的大小。如果有必要的话，   
Winsock能缓冲大于SO\_SNDBUF缓冲区大小的数据。在绝大多数情况下，应用程序完成Send调用仅仅表明数据   
被复制到了Winsock内核缓冲区，并不能说明数据就实际地被投递到了网络上。唯一一种例外的情况是：   
通过设置SO\_SNDBUT为0禁用了Winsock内核缓冲区。

Winsock使用下面的规则来向应用程序表明一个Send调用的完成：   
1、如果socket仍然在SO\_SNDBUF限额内，Winsock复制应用程序要发送的数据到内核缓冲区，完成Send调用。   
2、如果Socket超过了SO\_SNDBUF限额并且先前只有一个被缓冲的发送数据在内核缓冲区，Winsock复制要发送   
的数据到内核缓冲区，完成Send调用。   
3、如果Socket超过了SO\_SNDBUF限额并且内核缓冲区有不只一个被缓冲的发送数据，Winsock复制要发送的数据   
到内核缓冲区，然后投递数据到网络，直到Socket降到SO\_SNDBUF限额内或者只剩余一个要发送的数据，才   
完成Send调用。

案例1   
一个Winsock TCP客户端需要发送10000个记录到Winsock TCP服务端，保存到数据库。记录大小从20字节到100   
字节不等。对于简单的应用程序逻辑，可能的设计方案如下：   
1、客户端以阻塞方式发送，服务端以阻塞方式接收。   
2、客户端设置SO\_SNDBUF为0，禁用Nagle算法，让每个数据包单独的发送。   
3、服务端在一个循环中调用Recv接收数据包。给Recv传递200字节的缓冲区以便让每个记录在一次Recv调用中   
被获取到。

性能：   
在测试中发现，客户端每秒只能发送5条数据到服务段，总共10000条记录，976K字节左右，用了半个多小时   
才全部传到服务器。

分析：   
因为客户端没有设置TCP\_NODELAY选项，Nagle算法强制TCP栈在发送数据包之前等待前一个数据包的ACK确认   
信息。然而，客户端设置SO\_SNDBUF为0，禁用了内核缓冲区。因此，10000个Send调用只能一个数据包一个数据   
包的发送和确认，由于下列原因，每个ACK确认信息被延迟200毫秒：   
1、当服务器获取到一个数据包，启动一个200毫秒的计时器。   
2、服务端不需要向客户端发送任何数据，所以，ACK确认信息不能被发回的数据包顺路携带。   
3、客户端在没有收到前一个数据包的确认信息前，不能发送数据包。   
4、服务端的计时器超时后，ACK确认信息被发送到客户端。

如何提高性能：   
在这个设计中存在两个问题。第一，存在延时问题。客户端需要能够在200毫秒内发送两个数据包到服务端。   
因为客户端默认情况下使用Nagle算法，应该使用默认的内核缓冲区，不应该设置SO\_SNDBUF为0。一旦TCP   
栈拼接起来的数据包超过MTU值，这个数据包会立即被发送，不用等待前一个ACK确认信息。第二，这个设计   
方案对每一个如此小的的数据包都调用一次Send。发送这么小的数据包是不很有效率的。在这种情况下，应该   
把每个记录补充到100字节并且每次调用Send发送80个记录。为了让服务端知道一次总共发送了多少个记录，   
客户端可以在记录前面带一个头信息。

案例二：   
一个Winsock TCP客户端程序打开两个连接和一个提供股票报价服务的Winsock TCP服务端通信。第一个连接   
作为命令通道用来传输股票编号到服务端。第二个连接作为数据通道用来接收股票报价。两个连接被建立后，   
客户端通过命令通道发送股票编号到服务端，然后在数据通道上等待返回的股票报价信息。客户端在接收到第一   
个股票报价信息后发送下一个股票编号请求到服务端。客户端和服务端都没有设置SO\_SNDBUF和TCP\_NODELAY   
选项。

性能：   
测试中发现，客户端每秒只能获取到5条报价信息。

分析：

这个设计方案一次只允许获取一条股票信息。第一个股票编号信息通过命令通道发送到服务端，立即接收到   
服务端通过数据通道返回的股票报价信息。然后，客户端立即发送第二条请求信息，send调用立即返回，   
发送的数据被复制到内核缓冲区。然而，TCP栈不能立即投递这个数据包到网络，因为没有收到前一个数据包的   
ACK确认信息。200毫秒后，服务端的计时器超时，第一个请求数据包的ACK确认信息被发送回客户端，客户端   
的第二个请求包才被投递到网络。第二个请求的报价信息立即从数据通道返回到客户端，因为此时，客户端的   
计时器已经超时，第一个报价信息的ACK确认信息已经被发送到服务端。这个过程循环发生。

如何提高性能：   
在这里，两个连接的设计是没有必要的。如果使用一个连接来请求和接收报价信息，股票请求的ACK确认信息会   
被返回的报价信息立即顺路携带回来。要进一步的提高性能，客户端应该一次调用Send发送多个股票请求，服务端   
一次返回多个报价信息。如果由于某些特殊原因必须要使用两个单向的连接，客户端和服务端都应该设置TCP\_NODELAY   
选项，让小数据包立即发送而不用等待前一个数据包的ACK确认信息。

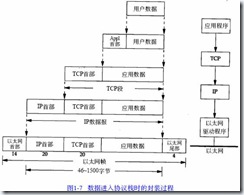
提高性能的建议：   
上面两个案例说明了一些最坏的情况。当设计一个方案解决大量的小数据包发送和接收时，应该遵循以下的建议：   
1、如果数据片段不需要紧急传输的话，应用程序应该将他们拼接成更大的数据块，再调用Send。因为发送缓冲区   
很可能被复制到内核缓冲区，所以缓冲区不应该太大，通常比8K小一点点是很有效率的。只要Winsock内核缓冲区   
得到一个大于MTU值的数据块，就会发送若干个数据包，剩下最后一个数据包。发送方除了最后一个数据包，都不会   
被200毫秒的计时器触发。   
2、如果可能的话，避免单向的Socket数据流接连。   
3、不要设置SO\_SNDBUF为0，除非想确保数据包在调用Send完成之后立即被投递到网络。事实上，8K的缓冲区适合大多数   
情况，不需要重新改变，除非新设置的缓冲区经过测试的确比默认大小更高效。   
4、如果数据传输不用保证可靠性，使用UDP。

**结论：**

1.         TCP提供了面向“连续字节流”的可靠的传输服务，TCP并不理解流所携带的数据内容，这个内容需要应用层自己解析。

2.         “字节流”是连续的、非结构化的，而我们的应用需要的是有序的、结构化的数据信息，因此我们需要定义自己的“规则”去解读这个“连续的字节流“，那解决途径就是定义自己的封包类型，然后用这个类型去映射“连续字节流”。

如何定义封包，我们回顾一下前面这个数据进入协议栈的封装过程图：

[](http://images.cnblogs.com/cnblogs_com/jiangtong/201203/201203221733242750.jpg)

封包其实就是将上图中进入协议栈的用户数据[即用户要发送的数据]定义为一种方便识别和交流的类型，这有点类似信封的概念,信封就是一种人们之间通信的格式,信封格式如下：

信封格式：

       收信人邮编

       收信人地址

       收信人姓名

       信件内容

那么在程序里面我们也需要定义这种格式：在C++里面只有结构和类这种两种类型适合表达这个概念了。网络上很多朋友对此表述了自己的看法并贴出了代码：比如

       /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 数据封包信息定义开始                                                 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma pack(push,1)   //将原对齐方式压栈,采用新的1字节对齐方式

/\* 封包类型枚举[此处根据需求列举] \*/

typedef enum{

              NLOGIN=1,

              NREG=2,

              NBACKUP=3,

              NRESTORE=3,

              NFILE\_TRANSFER=4,

              NHELLO=5

} PACKETTYPE;

/\* 包头 \*/

typedef struct tagNetPacketHead{

       byte version;//版本

       PACKETTYPE ePType;//包类型

       WORD nLen;//包体长度

} NetPacketHead;

/\* 封包对象[包头&包体] \*/

typedef struct tagNetPacket{

       NetPacketHead netPacketHead;//包头

       char \* packetBody;//包体

} NetPacket;

#pragma pack(pop)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据封包信息定义结束\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

3.         发包顺序与收包问题

a)         由于TCP要通过协商解决发送出去的报文段的长度，因此我们发送的数据很有可能被分割甚至被分割后再重组交给网络层发送，而网络层又是采用分组传送，即网络层数据报到达目标的顺序完全无法预测，那么收包会出现半包、粘包问题。举个例子，发送端连续发送两端数据msg1和msg2，那么发送端[传输层]可能会出现以下情况：

                                       i.              Msg1和msg2小于TCP的MSS，两个包按照先后顺序被发出，没有被分割和重组

                                     ii.              Msg1过大被分割成两段TCP报文msg1-1、msg2-2进行传送,msg2较小直接被封装成一个报文传送

                                    iii.              Msg1过大被分割成两段TCP报文msg1-1、msg2-2,msg1-1先被传送,剩下的msg1-2和msg2[较小]被组合成一个报文传送

                                   iv.              Msg1过大被分割成两段TCP报文msg1-1、msg2-2,msg1-1先被传送,剩下的msg1-2和msg2[较小]组合起来还是太小,组合的内容在和后面再发送的msg3的前部分数据组合起来发送

                                     v.              ……………………….太多……………………..

b)        接收端[传输层]可能出现的情况

                                       i.              先收到msg1,再收到msg2,这种方式太顺利了。

                                     ii.              先收到msg1-1，再收到msg1-2，再收到msg2

                                    iii.              先收到msg1，再收到msg2-1,再收到msg2-2

                                   iv.              先收到msg1和msg2-1,再收到msg2-2

                                     v.              //…………还有很多………………

c)        其实“接收端网络层”接收到的分组数据报顺序和发送端比较可能完全是乱的，比如发“送端网络层”发送1、2、3、4、5，而接收端网络层接收到的数据报顺序却可能是2、1、5、4、3，但是“接收端的传输层”会保证链接的有序性和可靠性，“接收端的传输层”会对“接收端网络层”收到的顺序紊乱的数据报重组成有序的报文[即发送方传输层发出的顺序]，然后交给“接收端应用层”使用，所以“接收端传输层”总是能够保证数据包的有序性，“接收端应用层”[我们编写的socket程序]不用担心接收到的数据的顺序问题。

d)        但是如上所述，粘包问题和半包问题不可避免。我们在接收端应用层需要自己编码处理粘包和半包问题。一般做法是定义一个缓冲区或者是使用标准库/框架提供的容器循环存放接收到数据，边接收变判断缓冲区数据是否满足包头大小，如果满足包头大小再判断缓冲区剩下数据是否满足包体大小，如果满足则提取。详细步骤如下：

1.         接收数据存入缓冲区尾部

2.         缓冲区数据满足包头大小否

3.         缓冲区数据不满足包头大小，回到第1步；缓冲区数据满足包头大小则取出包头，接着判断缓冲区剩余数据满足包头中定义的包体大小否，不满足则回到第1步。

4.         缓冲区数据满足一个包头大小和一个包体大小之和，则取出包头和包体进行使用,此处使用可以采用拷贝方式转移缓冲区数据到另外一个地方，也可以为了节省内存直接采取调用回调函数的方式完成数据使用。

5.         清除缓冲区的第一个包头和包体信息，做法一般是将缓冲区剩下的数据拷贝到缓冲区首部覆盖“第一个包头和包体信息”部分即可。

粘包、半包处理具体实现很多朋友都有自己的做法，比如最前面贴出的链接，这里我也贴出一段参考：

缓冲区实现头文件：

#include <windows.h>

#ifndef \_CNetDataBuffer\_H\_

#define \_CNetDataBuffer\_H\_

#ifndef TCPLAB\_DECLSPEC

#define TCPLAB\_DECLSPEC \_declspec(dllimport)

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 数据封包信息定义开始                                                 \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma pack(push,1)   //将原对齐方式压栈,采用新的1字节对齐方式

/\* 封包类型枚举[此处根据需求列举] \*/

typedef enum{

              NLOGIN=1,

              NREG=2,

              NBACKUP=3,

              NRESTORE=3,

              NFILE\_TRANSFER=4,

              NHELLO=5

} PACKETTYPE;

/\* 包头 \*/

typedef struct tagNetPacketHead{

       byte version;//版本

       PACKETTYPE ePType;//包类型

       WORD nLen;//包体长度

} NetPacketHead;

/\* 封包对象[包头&包体] \*/

typedef struct tagNetPacket{

       NetPacketHead netPacketHead;//包头

       char \* packetBody;//包体

} NetPacket;

#pragma pack(pop)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*数据封包信息定义结束\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//缓冲区初始大小

#define BUFFER\_INIT\_SIZE 2048

//缓冲区膨胀系数[缓冲区膨胀后的大小=原大小+系数\*新增数据长度]

#define BUFFER\_EXPAND\_SIZE 2

//计算缓冲区除第一个包头外剩下的数据的长度的宏[缓冲区数据总长度-包头大小]

#define BUFFER\_BODY\_LEN (m\_nOffset-sizeof(NetPacketHead))

//计算缓冲区数据当前是否满足一个完整包数据量[包头&包体]

#define HAS\_FULL\_PACKET ( \

                                                 (sizeof(NetPacketHead)<=m\_nOffset) && \

                                                 ((((NetPacketHead\*)m\_pMsgBuffer)->nLen) <= BUFFER\_BODY\_LEN) \

                                          )

//检查包是否合法[包体长度大于零且包体不等于空]

#define IS\_VALID\_PACKET(netPacket) \

       ((netPacket.netPacketHead.nLen>0) && (netPacket.packetBody!=NULL))

//缓冲区第一个包的长度

#define FIRST\_PACKET\_LEN (sizeof(NetPacketHead)+((NetPacketHead\*)m\_pMsgBuffer)->nLen)

/\* 数据缓冲 \*/

class /\*TCPLAB\_DECLSPEC\*/ CNetDataBuffer

{

       /\* 缓冲区操作相关成员 \*/

private:

       char \*m\_pMsgBuffer;//数据缓冲区

       int m\_nBufferSize;//缓冲区总大小

       int m\_nOffset;//缓冲区数据大小

public:

       int GetBufferSize() const;//获得缓冲区的大小

       BOOL ReBufferSize(int);//调整缓冲区的大小

       BOOL IsFitPacketHeadSize() const;//缓冲数据是否适合包头大小

       BOOL IsHasFullPacket() const;//缓冲区是否拥有完整的包数据[包含包头和包体]

       BOOL AddMsg(char \*pBuf,int nLen);//添加消息到缓冲区

       const char \*GetBufferContents() const;//得到缓冲区内容

       void Reset();//缓冲区复位[清空缓冲区数据,但并未释放缓冲区]

       void Poll();//移除缓冲区首部的第一个数据包

public:

       CNetDataBuffer();

       ~CNetDataBuffer();

};

#endif

缓冲区实现文件:

#define TCPLAB\_DECLSPEC \_declspec(dllexport)

#include "CNetDataBuffer.h"

/\* 构造 \*/

CNetDataBuffer::CNetDataBuffer()

{

       m\_nBufferSize = BUFFER\_INIT\_SIZE;//设置缓冲区大小

       m\_nOffset = 0;//设置数据偏移值[数据大小]为0

       m\_pMsgBuffer = NULL;

       m\_pMsgBuffer = new char[BUFFER\_INIT\_SIZE];//分配缓冲区为初始大小

       ZeroMemory(m\_pMsgBuffer,BUFFER\_INIT\_SIZE);//缓冲区清空

}

/\* 析构 \*/

CNetDataBuffer::~CNetDataBuffer()

{

       if (m\_nOffset!=0)

       {

              delete [] m\_pMsgBuffer;//释放缓冲区

              m\_pMsgBuffer = NULL;

              m\_nBufferSize=0;

              m\_nOffset=0;

       }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Description:       获得缓冲区中数据的大小                                  \*/

/\* Return:              缓冲区中数据的大小                                                                   \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INT CNetDataBuffer::GetBufferSize() const

{

       return this->m\_nOffset;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Description:       缓冲区中的数据大小是否足够一个包头大小                  \*/

/\* Return:              如果满足则返回True,否则返回False

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL CNetDataBuffer::IsFitPacketHeadSize() const

{

       return sizeof(NetPacketHead)<=m\_nOffset;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Description:       判断缓冲区是否拥有完整的数据包(包头和包体)              \*/

/\* Return:              如果缓冲区包含一个完整封包则返回True,否则False                   \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL CNetDataBuffer::IsHasFullPacket() const

{

       //如果连包头大小都不满足则返回

       //if (!IsFitPacketHeadSize())

       //     return FALSE;

       return HAS\_FULL\_PACKET;//此处采用宏简化代码

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Description:       重置缓冲区大小                                                  \*/

/\* nLen:          新增加的数据长度                                                                      \*/

/\* Return:              调整结果                                                                                    \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL CNetDataBuffer::ReBufferSize(int nLen)

{

       char \*oBuffer = m\_pMsgBuffer;//保存原缓冲区地址

       try

       {

              nLen=(nLen<64?64:nLen);//保证最小增量大小

              //新缓冲区的大小=增加的大小+原缓冲区大小

              m\_nBufferSize = BUFFER\_EXPAND\_SIZE\*nLen+m\_nBufferSize;

              m\_pMsgBuffer = new char[m\_nBufferSize];//分配新的缓冲区,m\_pMsgBuff指向新缓冲区地址

              ZeroMemory(m\_pMsgBuffer,m\_nBufferSize);//新缓冲区清零

              CopyMemory(m\_pMsgBuffer,oBuffer,m\_nOffset);//将原缓冲区的内容全部拷贝到新缓冲区

       }

       catch(...)

       {

              throw;

       }

       delete []oBuffer;//释放原缓冲区

       return TRUE;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Description:       向缓冲区添加消息                                        \*/

/\* pBuf:          要添加的数据                                                                             \*/

/\* nLen:          添加的消息长度

/\* return:        添加成功返回True,否则False                                                      \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL CNetDataBuffer::AddMsg(char \*pBuf,int nLen)

{

       try

       {

              //检查缓冲区长度是否满足,不满足则重新调整缓冲区大小

              if (m\_nOffset+nLen>m\_nBufferSize)

                     ReBufferSize(nLen);

              //拷贝新数据到缓冲区末尾

              CopyMemory(m\_pMsgBuffer+sizeof(char)\*m\_nOffset,pBuf,nLen);

              m\_nOffset+=nLen;//修改数据偏移

       }

       catch(...)

       {

              return FALSE;

       }

       return TRUE;

}

/\* 得到缓冲区内容 \*/

const char \* CNetDataBuffer::GetBufferContents() const

{

       return m\_pMsgBuffer;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 缓冲区复位                                                           \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void CNetDataBuffer::Reset()

{

       if (m\_nOffset>0)

       {

              m\_nOffset = 0;

              ZeroMemory(m\_pMsgBuffer,m\_nBufferSize);

       }

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 移除缓冲区首部的第一个数据包                                         \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void CNetDataBuffer::Poll()

{

       if(m\_nOffset==0 || m\_pMsgBuffer==NULL)

              return;

       if (IsFitPacketHeadSize() && HAS\_FULL\_PACKET)

       {

              CopyMemory(m\_pMsgBuffer,m\_pMsgBuffer+FIRST\_PACKET\_LEN\*sizeof(char),m\_nOffset-FIRST\_PACKET\_LEN);

       }

}

对TCP发包和收包进行简单封装：

头文件：

#include <windows.h>

#include "CNetDataBuffer.h"

// #ifndef TCPLAB\_DECLSPEC

// #define TCPLAB\_DECLSPEC \_declspec(dllimport)

// #endif

#ifndef \_CNETCOMTEMPLATE\_H\_

#define \_CNETCOMTEMPLATE\_H\_

//通信端口

#define TCP\_PORT 6000

/\* 通信终端[包含一个Socket和一个缓冲对象] \*/

typedef struct {

       SOCKET m\_socket;//通信套接字

       CNetDataBuffer m\_netDataBuffer;//该套接字关联的数据缓冲区

} ComEndPoint;

/\* 收包回调函数参数 \*/

typedef struct{

       NetPacket \*pPacket;

       LPVOID processor;

       SOCKET comSocket;

} PacketHandlerParam;

class CNetComTemplate{

       /\* Socket操作相关成员 \*/

private:

public:

       void SendPacket(SOCKET m\_connectedSocket,NetPacket &netPacket);//发包函数

       BOOL RecvPacket(ComEndPoint &comEndPoint,void (\*recvPacketHandler)(LPVOID)=NULL,LPVOID=NULL);//收包函数

public:

       CNetComTemplate();

       ~CNetComTemplate();

};

#endif

实现文件：

#include "CNetComTemplate.h"

CNetComTemplate::CNetComTemplate()

{

}

CNetComTemplate::~CNetComTemplate()

{

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Description:发包                                                     \*/

/\* m\_connectedSocket:建立好连接的套接字                                                             \*/

/\* netPacket:要发送的数据包                                                                                   \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void CNetComTemplate::SendPacket(SOCKET m\_connectedSocket,NetPacket &netPacket)

{

       if (m\_connectedSocket==NULL || !IS\_VALID\_PACKET(netPacket))//如果尚未建立连接则退出

       {

              return;

       }

       ::send(m\_connectedSocket,(char\*)&netPacket.netPacketHead,sizeof(NetPacketHead),0);//先发送包头

       ::send(m\_connectedSocket,netPacket.packetBody,netPacket.netPacketHead.nLen,0);//在发送包体

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Description:收包                                                       \*/

/\* comEndPoint:通信终端[包含套接字和关联的缓冲区]                                     \*/

/\* recvPacketHandler:收包回调函数,当收到一个包后调用该函数进行包的分发处理\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BOOL CNetComTemplate::RecvPacket(ComEndPoint &comEndPoint,void (\*recvPacketHandler)(LPVOID),LPVOID pCallParam)

{

       if (comEndPoint.m\_socket==NULL)

              return FALSE;

       int nRecvedLen = 0;

       char pBuf[1024];

       //如果缓冲区数据不够包大小则继续从套接字读取tcp报文段

       while (!(comEndPoint.m\_netDataBuffer.IsHasFullPacket()))

       {

              nRecvedLen = recv(comEndPoint.m\_socket,pBuf,1024,0);

              if (nRecvedLen==SOCKET\_ERROR || nRecvedLen==0)//若果Socket错误或者对方连接已经正常关闭则结束读取

                     break;

              comEndPoint.m\_netDataBuffer.AddMsg(pBuf,nRecvedLen);//将新接收的数据存入缓冲区

       }

       //执行到此处可能是三种情况:

       //1.已经读取到的数据满足一个完整的tcp报文段

       //2.读取发生socket\_error错误

       //3.在还未正常读取完毕的过程中对方连接已经关闭

       //如果没有读取到数据或者没有读取到完整报文段则返回返回

       if (nRecvedLen==0 || (!(comEndPoint.m\_netDataBuffer.IsHasFullPacket())))

       {

              return FALSE;

       }

       if (recvPacketHandler!=NULL)

       {

              //构造准备传递给回调函数的数据包

              NetPacket netPacket;

              netPacket.netPacketHead = \*(NetPacketHead\*)comEndPoint.m\_netDataBuffer.GetBufferContents();

              netPacket.packetBody = new char[netPacket.netPacketHead.nLen];//动态分配包体空间

              //构造回调函数参数

              PacketHandlerParam packetParam;

              packetParam.pPacket = &netPacket;

              packetParam.processor = pCallParam;

              //呼叫回调函数

              recvPacketHandler(&packetParam);

              delete []netPacket.packetBody;

       }

       //移除缓冲区的第一个包

       comEndPoint.m\_netDataBuffer.Poll();

       return TRUE;

}

分类: [网络通信](http://www.cnblogs.com/jiangtong/category/365894.html)