高性能网络编程2----TCP消息的发送

2013年07月18日 16:37:55 阅读数：28481

在上一篇中，我们已经建立好的TCP连接，对应着操作系统分配的1个套接字。操作TCP协议发送数据时，面对的是数据流。通常调用诸如send或者write方法来发送数据到另一台主机，那么，调用这样的方法时，在操作系统内核中发生了什么事情呢？我们带着以下3个问题来细细分析：发送方法成功返回时，能保证TCP另一端的主机接收到吗？能保证数据已经发送到网络上了吗？套接字为阻塞或者非阻塞时，发送方法做的事情有何不同？

要回答上面3个问题涉及了不少知识点，我们先在TCP层面上看看，发送方法调用时内核做了哪些事。我不想去罗列内核中的数据结构、方法等，毕竟大部分应用程序开发者不需要了解这些，仅以一幅示意图粗略表示，如下：

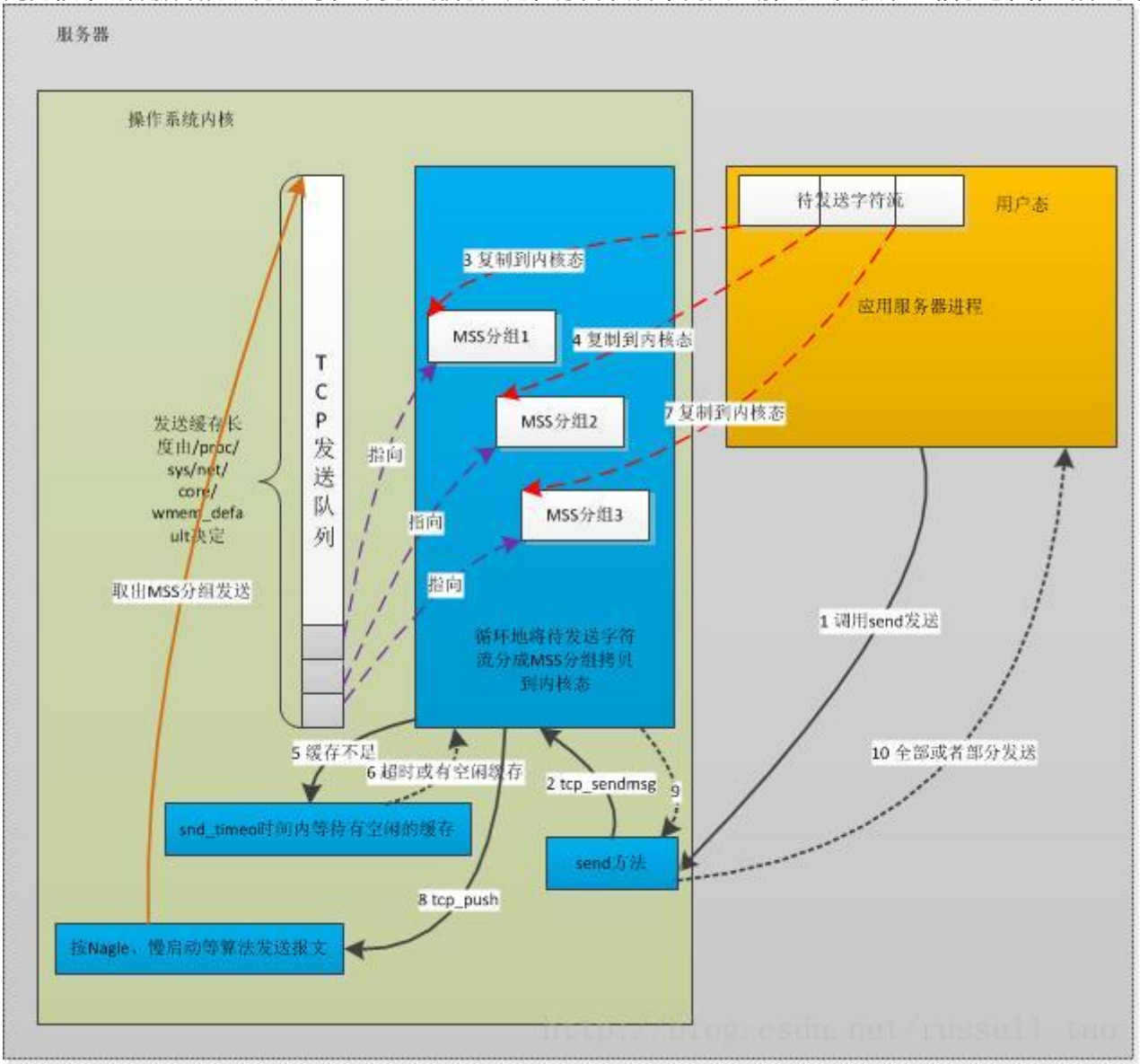

图1 一种典型场景下发送TCP消息的流程

再详述上图10个步骤前，先要澄清几个概念：MTU、MSS、tcp_write_queue发送队列、阻塞与非阻塞套接字、拥塞窗口、滑动窗口、Nagle算法。

当我们调用发送方法时，会把我们代码中构造好的消息流作为参数传递。这个消息流可大可小，例如几个字节，或者几兆字节。当消息流较大时，将有可能出现分片。我们先来讨论分片问题。

1、MSS与TCP的分片

由上一篇文中可知，TCP层是第4层传输层，第3层IP网络层、第2层数据链路层具备的约束条件同样对TCP层生效。下面来看看数据链路层中的一个概念：最大传输单元MTU。

无论何种类型的数据链路层，都会对网络分组的长度有一个限制。例如以太网限制为1500字节，802.3限制为1492字节。当内核的IP网络层试图发送报文时，若一个报文的长度大于MTU限制，就会被分成若干个小于MTU的报

文，每个报文都会有独立的IP头部。

看看IP头部的格式：

图2 IP头部格式

可以看到，其指定IP包总长度的是一个16位（2字节）的字段，这意味一个IP包最大可以是65535字节。

若TCP层在以太网中试图发送一个大于1500字节的消息，调用IP网络层方法发送消息时，IP层会自动的获取所在局域网的MTU值，并按照所在网络的MTU大小来分片。IP层同时希望这个分片对于传输层来说是透明的，接收方的IP层会根据收到的多个IP包头部，将发送方IP层分片出的IP包重组为一个消息。

这种IP层的分片效率是很差的，因为必须所有分片都到达才能重组成一个包，其中任何一个分片丢失了，都必须重发所有分片。所以，TCP层会试图避免IP层执行数据报分片。

为了避免IP层的分片，TCP协议定义了一个新的概念：最大报文段长度MSS。它定义了一个TCP连接上，一个主机期望对端主机发送单个报文的最大长度。TCP3次握手建立连接时，连接双方都要互相告知自己期望接收到的MSS大小。例如（使用tcpdump抓包）：

*15:05:08.230782 IP 10.7.80.57.64569 > houyi-vm02.dev.sd.aliyun.com.tproxy: S 3027092051:3027092051(0) win 8192 <**mss 1460**,nop,wscale 8,nop,nop,sackOK>*

*15:05:08.234267 IP houyi-vm02.dev.sd.aliyun.com.tproxy > 10.7.80.57.64569: S 26006838:26006838(0) ack 3027092052 win 5840 <**mss 1460**,nop,nop,sackOK,nop,wscale 9>*

15:05:08.233320 IP 10.7.80.57.64543 > houyi-vm02.dev.sd.aliyun.com.tproxy: P 78972532:78972923(391) ack 12915963 win 255

由于例子中两台主机都在以太网内，以太网的MTU为1500，减去IP和TCP头部的长度，MSS就是1460，三次握手中，SYN包都会携带期望的MSS大小。

当应用层调用TCP层提供的发送方法时，内核的TCP模块在tcp_sendmsg方法里，会按照对方告知的MSS来分片，把消息流分为多个网络分组（如图1中的3个网络分组），再调用IP层的方法发送数据。

这个MSS就不会改变了吗?

会的。上文说过，MSS就是为了避免IP层分片，在建立握手时告知对方期望接收的MSS值并不一定靠得住。因为这个值是预估的，TCP连接上的两台主机若处于不同的网络中，那么，连接上可能有许多中间网络，这些网络分别具有不同的数据链路层，这样，TCP连接上有许多个MTU。特别是，若中间途径的MTU小于两台主机所在的网络MTU时，选定的MSS仍然太大了，会导致中间路由器出现IP层的分片。

怎样避免中间网络可能出现的分片呢?

通过IP头部的DF标志位，这个标志位是告诉IP报文所途经的所有IP层代码：不要对这个报文分片。如果一个IP报文太大必须要分片，则直接返回一个ICMP错误，说明必须要分片了，且待分片路由器网络接受的MTU值。这样，连

接上的发送方主机就可以重新确定MSS。

2、发送方法返回成功后，数据一定发送到了TCP的另一端吗?
答案当然是否定的。解释这个问题前，先来看看TCP是如何保证可靠传输的。
TCP把自己要发送的数据流里的每一个字节都看成一个序号，可靠性是要求连接对端在接收到数据后，要发送ACK确认，告诉它已经接收到了多少字节的数据。也就是说，怎样确保数据一定发送成功了呢？必须等待发送数据对应序号的ACK到达，才能确保数据一定发送成功。TCP层提供的send或者write这样的方法是不会做这件事的，看看图1，它究竟做了哪些事。

图1中分为10步。
（1）应用程序试图调用send方法来发送一段较长的数据。
（2）内核主要通过tcp_sendmsg方法来完成。
（3）（4）内核真正执行报文的发送，与send方法的调用并不是同步的。即，send方法返回成功了，也不一定把IP报文都发送到网络中了。因此，需要把用户需要发送的用户态内存中的数据，拷贝到内核态内存中，不依赖于用户态内存，也使得进程可以快速释放发送数据占用的用户态内存。但这个拷贝操作并不是简单的复制，而是把待发送数据，按照MSS来划分成多个尽量达到MSS大小的分片报文段，复制到内核中的sk_buff结构来存放，同时把这些分片组成队列，放到这个TCP连接对应的tcp_write_queue发送队列中。
（5）内核中为这个TCP连接分配的内核缓存是有限的（/proc/sys/net/core/wmem_default）。当没有多余的内核态缓存来复制用户态的待发送数据时，就需要调用一个方法sk_stream_wait_memory来等待滑动窗口移动，释放出一些缓存出来（收到ACK后，不需要再缓存原来已经发送出的报文，因为既然已经确认对方收到，就不需要定时重发，自然就释放缓存了）。例如：

```
wait_for_memory:
                        if (copied)

                             tcp_push(sk, tp, flags & ~MSG_MORE, mss_now, TCP_NAGLE
    |                        if ((err = sk_stream_wait_memory(sk, &timeo)) != 0)
                                goto do_error;
```

这里的sk_stream_wait_memory方法接受一个参数timeo，就是等待超时的时间。这个时间是tcp_sendmsg方法刚开始就拿到的，如下：

```
timeo = sock_sndtimeo(sk, flags & MSG_DONTWAIT);
```

看看其实现：

```
static inline long sock_sndtimeo(const struct sock *sk, int noblock)
{
        return noblock ? 0 : sk->sk_sndtimeo;
}
```

也就是说，当这个套接字是阻塞套接字时，timeo就是SO_SNDTIMEO选项指定的发送超时时间。如果这个套接字是非阻塞套接字， timeo变量就会是0。
实际上，sk_stream_wait_memory对于非阻塞套接字会直接返回，并将 errno错误码置为EAGAIN。
（6）在图1的例子中，我们假定使用了阻塞套接字，且等待了足够久的时间，收到了对方的ACK，滑动窗口释放出了缓存。

（7）将剩下的用户态数据都组成MSS报文拷贝到内核态的sk_buff中。
（8）最后，调用tcp_push等方法，它最终会调用IP层的方法来发送tcp_write_queue队列中的报文。
注意，IP层返回时，并不一定是把报文发送了出去。
（9）（10）发送方法返回。

从图1的10个步骤中可知，无论是使用阻塞还是非阻塞套接字，发送方法成功返回时（无论全部成功或者部分成功），既不代表TCP连接的另一端主机接收到了消息，也不代表本机把消息发送到了网络上，只是说明，内核将会试图保证把消息送达对方。

3、Nagle算法、滑动窗口、拥塞窗口对发送方法的影响
图1第8步tcp_push方法做了些什么呢？先来看看主要的流程：

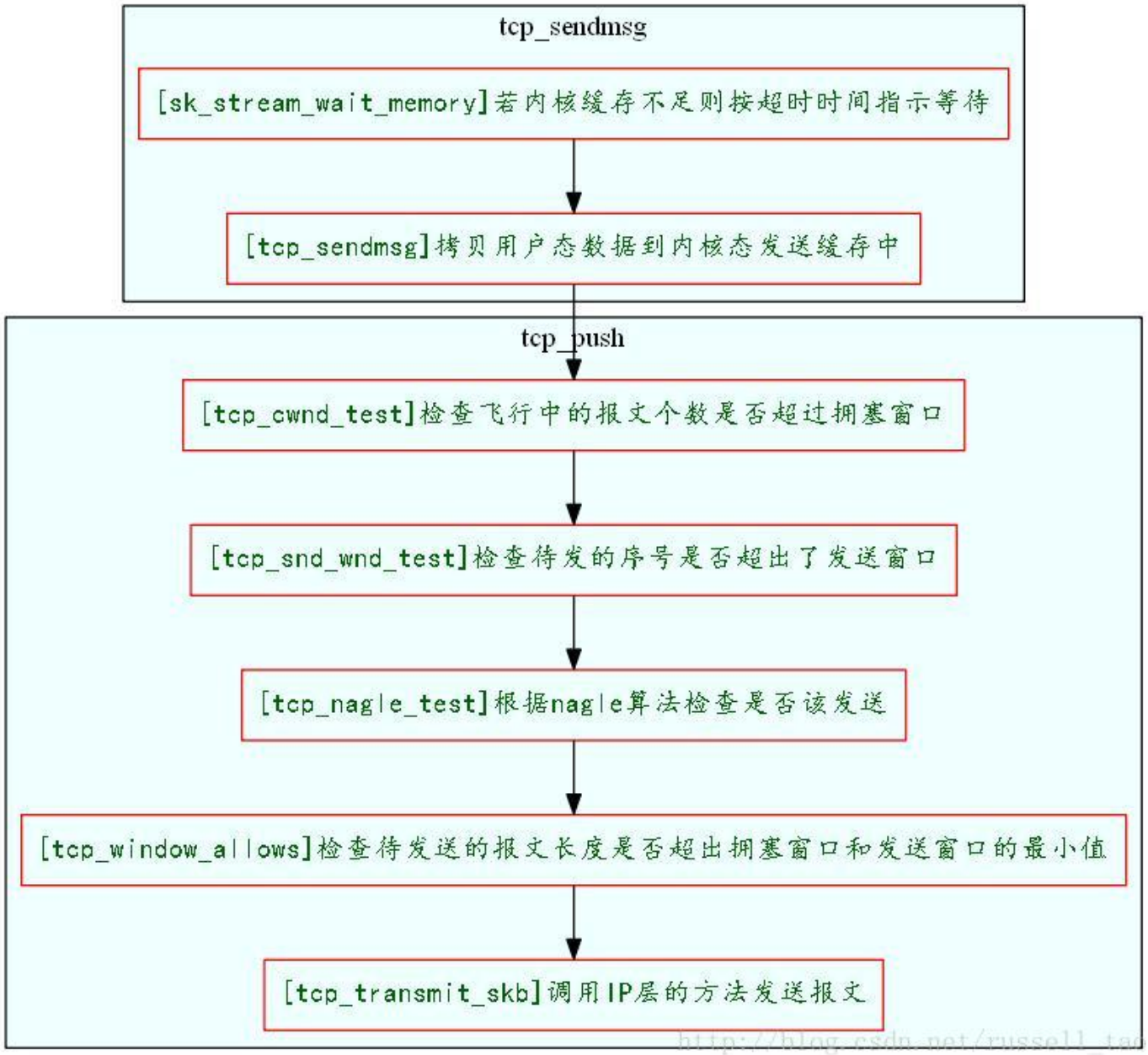

图3 发送TCP消息的简易流程

下面简单看看这几个概念：
（1）滑动窗口
滑动窗口大家都比较熟悉，就不详细介绍了。TCP连接上的双方都会通知对方自己的接收窗口大小。而对方的接收窗口大小就是自己的发送窗口大小。tcp_push在发送数据时当然需要与发送窗口打交道。发送窗口是一个时刻变化的值，随着ACK的到达会变大，随着发出新的数据包会变小。当然，最大也只能到三次握手时对方通告的窗口大小。tcp_push在发送数据时，最终会使用tcp_snd_wnd_test方法来判断当前待发送的数据，其序号是否超出了发送滑动窗口的大小，例如：

```
//检查这一次要发送的报文最大序号是否超出了发送滑动窗口大小

static inline int tcp_snd_wnd_test(struct tcp_sock *tp, struct sk_buff *skb, unsigned
{       //end_seq待发送的最大序号
        u32 end_seq = TCP_SKB_CB(skb)->end_seq;

        if (skb->len > cur_mss)
                end_seq = TCP_SKB_CB(skb)->seq + cur_mss;

        //snd_una是已经发送过的数据中，最小的没被确认的序号；而snd_wnd就是发送窗口的大小
        return !after(end_seq, tp->snd_una + tp->snd_wnd);
}
```

（2）慢启动和拥塞窗口

由于两台主机间的网络可能很复杂，通过广域网时，中间的路由器转发能力可能是瓶颈。也就是说，如果一方简单的按照另一方主机三次握手时通告的滑动窗口大小来发送数据的话，可能会使得网络上的转发路由器性能雪上加霜，最终丢失更多的分组。这时，各个操作系统内核都会对TCP的发送阶段加入慢启动和拥塞避免算法。慢启动算法说白了，就是对方通告的窗口大小只表示对方接收TCP分组的能力，不表示中间网络能够处理分组的能力。所以，发送方请悠着点发，确保网络非常通畅了后，再按照对方通告窗口来敞开了发。

拥塞窗口就是下面的cwnd，它用来帮助慢启动的实现。连接刚建立时，拥塞窗口的大小远小于发送窗口，它实际上是一个MSS。每收到一个ACK，拥塞窗口扩大一个MSS大小，当然，拥塞窗口最大只能到对方通告的接收窗口大小。当然，为了避免指数式增长，拥塞窗口大小的增长会更慢一些，是线性的平滑的增长过程。

所以，在tcp_push发送消息时，还会检查拥塞窗口，飞行中的报文数要小于拥塞窗口个数，而发送数据的长度也要小于拥塞窗口的长度。

如下所示，首先用unsigned int tcp_cwnd_test方法检查飞行的报文数是否小于拥塞窗口个数（多少个MSS的个数）：

```
static inline unsigned int tcp_cwnd_test(struct tcp_sock *tp, struct sk_buff *skb)
{
        u32 in_flight, cwnd;

        /* Don't be strict about the congestion window for the final FIN.  */
        if (TCP_SKB_CB(skb)->flags & TCPCB_FLAG_FIN)
                return 1;

        //飞行中的数据，也就是没有ACK的字节总数
        in_flight = tcp_packets_in_flight(tp);
        cwnd = tp->snd_cwnd;
        //如果拥塞窗口允许，需要返回依据拥塞窗口的大小，还能发送多少字节的数据
        if (in_flight < cwnd)
                return (cwnd - in_flight);

        return 0;
}
```

再通过tcp_window_allows方法获取拥塞窗口与滑动窗口的最小长度，检查待发送的数据是否超出：

```
static unsigned int tcp_window_allows(struct tcp_sock *tp, struct sk_buff *skb, unsi
{
        u32 window, cwnd_len;

        window = (tp->snd_una + tp->snd_wnd - TCP_SKB_CB(skb)->seq);
        cwnd_len = mss_now * cwnd;
        return min(window, cwnd_len);
}
```

（3）是否符合NAGLE算法?
Nagle算法的初衷是这样的：应用进程调用发送方法时，可能每次只发送小块数据，造成这台机器发送了许多小的TCP报文。对于整个网络的执行效率来说，小的TCP报文会增加网络拥塞的可能，因此，如果有可能，应该将相临的TCP报文合并成一个较大的TCP报文（当然还是小于MSS的）发送。
Nagle算法要求一个TCP连接上最多只能有一个发送出去还没被确认的小分组，在该分组的确认到达之前不能发送其他的小分组。
内核中是通过 tcp_nagle_test方法实现该算法的。我们简单的看下：

```
[cpp]
1. <code class="language-cpp">static inline int tcp_nagle_test(struct tcp_sock *tp, struct sk_buff *s
   kb,
2.                 unsigned int cur_mss, int nonagle)
3. {
4.     //nonagle标志位设置了，返回1表示允许这个分组发送出去
5.     if (nonagle & TCP_NAGLE_PUSH)
6.         return 1;
7.
8.     //如果这个分组包含了四次握手关闭连接的FIN包，也可以发送出去
9.     if (tp->urg_mode ||
10.        (TCP_SKB_CB(skb)->flags & TCPCB_FLAG_FIN))
11.        return 1;
12.
13.        //检查Nagle算法
14.    if (!tcp_nagle_check(tp, skb, cur_mss, nonagle))
15.        return 1;
16.
17.    return 0;
18. }</code>
```

再来看看tcp_nagle_check方法，它与上一个方法不同，返回0表示可以发送，返回非0则不可以，正好相反。

```
static inline int tcp_nagle_check(const struct tcp_sock *tp,
                                  const struct sk_buff *skb,
                                  unsigned mss_now, int nonagle)
{
        //先检查是否为小分组，即报文长度是否小于MSS
        return (skb->len < mss_now &&
                ((nonagle&TCP_NAGLE_CORK) ||
        //如果开启了Nagle算法
                 (!nonagle &&
        //若已经有小分组发出（packets_out表示“飞行”中的分组）还没有确认
                  tp->packets_out &&
                  tcp_minshall_check(tp))));
}
```

最后看看tcp_minshall_check做了些什么：

```
static inline int tcp_minshall_check(const struct tcp_sock *tp)
{
        //最后一次发送的小分组还没有被确认
        return after(tp->snd_sml,tp->snd_una) &&
                //将要发送的序号是要大于等于上次发送分组对应的序号
                !after(tp->snd_sml, tp->snd_nxt);
}
```

想象一种场景，当对请求的时延非常在意且网络环境非常好的时候（例如同一个机房内），Nagle算法可以关闭，这实在也没必要。使用TCP_NODELAY套接字选项就可以关闭Nagle算法。看看setsockopt是怎么与上述方法配合工作的：

```
static int do_tcp_setsockopt(struct sock *sk, int level,
                int optname, char __user *optval, int optlen)
        ...
        switch (optname) {
        ...
        case TCP_NODELAY:
                if (val) {
                        //如果设置了TCP_NODELAY，则更新monagle标志
                        tp->nonagle |= TCP_NAGLE_OFF|TCP_NAGLE_PUSH;
                        tcp_push_pending_frames(sk, tp);
                } else {
                        tp->nonagle &= ~TCP_NAGLE_OFF;
                }
```

```
                break;
            }
        }
```

可以看到，nonagle标志位就是这么更改的。

当然，调用了IP层的方法返回后，也未必就保证此时数据一定发送到网络中去了。
下一篇我们探讨如何接收TCP消息，以及接收到ack后内核做了些什么。