**nginx对TCP\_CORK/TCP\_NOPUSH的使用**

2011年2月23日[Simon Liu](http://www.pagefault.info/?author=1)[发表评论](http://www.pagefault.info/?p=228#respond)[阅读评论](http://www.pagefault.info/?p=228#comments)

[WP Greet Box icon](http://www.pagefault.info/?feed=rss)

[**X**](http://www.pagefault.info/?p=228)

Hello there! If you are new here, you might want to [**subscribe to the RSS feed**](http://www.pagefault.info/?feed=rss) for updates on this topic.

Powered by [WP Greet Box](http://omninoggin.com/projects/wordpress-plugins/wp-greet-box-wordpress-plugin/) [WordPress Plugin](http://omninoggin.com/)

***原创文章，转载请注明：****转载自*[*pagefault*](http://www.pagefault.info/)

***本文链接地址:***[*nginx对TCP\_CORK/TCP\_NOPUSH的使用*](http://www.pagefault.info/?p=228)

在nginx中使用了send\_file 并且配合TCP\_CORK/TCP\_NOPUSH进行操作，我们一般的操作是这样子的，首先调用tcp\_cork，阻塞下层的数据发送，然后调用send\_file发送数据，最后关闭TCP\_CORK/TCP\_NOPUSH.而在nginx中不是这样处理的，前面两步都是一样的，最后一步，它巧妙的利用的http的特性，那就是基本都是短连接，也就是处理完当前的request之后，就会关闭当前的连接句柄，而在linux中，如果不是下面两种情况之一，那么关闭tcp句柄，就会发送完发送buf中的数据，才进行tcp的断开操作(具体可以看我以前写的那篇 “linux内核中tcp连接的断开处理”的 blog) :

1 接收buf中还有未读数据。  
2 so\_linger设置并且超时时间为0.

而如果调用shutdown来关闭写端的话，就是直接发送完写buf中的数据，然后发送fin。

ok，通过上面我们知道每次处理完请求，都会关闭连接(keepalive 会单独处理),而关闭连接就会帮我们将cork拔掉，所以这里就可以节省一个系统调用，从这里能看到nginx对细节的处理到了一个什么程度。

接下来还有一个单独要处理的就是keepalive的连接，由于keepalive是不会关闭当前的连接的，因此这里就必须显式的关闭tcp\_cork。

然后我们来看代码，首先来看TCP\_CORK/TCP\_NOPUSH相关的配置，我们知道在nginx中 TCP\_CORK/TCP\_NOPUSH是默认关闭的，除非我们显示的打开(tcp\_nopush on).而在nginx中对于TCP\_CORK/TCP\_NOPUSH选项的打开关闭是有两个位置，一个是ngx\_http\_core\_loc\_conf\_t的tcp\_nopush(这个对应配置文件里面的tcp\_nopush选项),一个是r->connection->tcp\_nopush，它默认是NGX\_TCP\_NOPUSH\_UNSET，可是如果没有设置tcp\_nopush on，则这个值就会被设为NGX\_TCP\_NOPUSH\_DISABLED，也就是关闭tcp\_cork.而这里主要使用的就是r->connection->tcp\_nopush。这里多出来的NGX\_TCP\_NOPUSH\_DISABLED主要就是针对keepalive的情况，后面我们会详细分析.

下面就是tcp\_nopush的可选值.

|  |  |
| --- | --- |
| 1  2  3  4  5 | typedef enum {       NGX\_TCP\_NOPUSH\_UNSET = 0,       NGX\_TCP\_NOPUSH\_SET,       NGX\_TCP\_NOPUSH\_DISABLED  } ngx\_connection\_tcp\_nopush\_e; |

来看core loc conf的值：

|  |  |
| --- | --- |
| 1 | ngx\_conf\_merge\_value(conf->tcp\_nopush, prev->tcp\_nopush, 0); |

可以看到他的默认值是0T，也就是tcp\_nopush没有打开的情况。

然后是r->connection->tcp\_nopush 的设置，它的值依赖于clcf->tcp\_nopush，如果clcf->tcp\_nopush为(默认值),则关闭tcp\_nopush,否则就是默认值打开。而connection中的tcp\_nopush默认是NGX\_TCP\_NOPUSH\_UNSET，也就是0。

|  |  |
| --- | --- |
| 1  2  3  4  5 | if (!clcf->tcp\_nopush) {          /\* disable TCP\_NOPUSH/TCP\_CORK use \*/  //设置关闭          r->connection->tcp\_nopush = NGX\_TCP\_NOPUSH\_DISABLED;      } |

ok，接下来我们就直接来看nginx中如何使用TCP\_CORK/TCP\_NOPUSH,主要的代码是在ngx\_linux\_sendfile\_chain（我们通过分析tcp\_cork来分析，bsd的tcp\_nopush和linux的处理类似)中，这个函数我前面的blog有详细分析过，这次主要是分析tcp\_cork部分，我们主要来看,第一次调用sendfile的时候，会先设置tcp\_cork.而设置之后就会设置c->tcp\_nopush为NGX\_TCP\_NOPUSH\_SET，也就是接下来的操作都不需要再次设置了。而最终当buf处理完毕，就直接返回。

这里有一个要注意的就是tcp\_nodelay和tcp\_nopush是互斥的。不过如果你同时设置了两个值的话，将会在第一个buf发送的时候，强制push数据，而第二个buf时，将会调用tcp\_cork来打开nagle算法，也就是后面的都会应用tcp\_nopush.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | //如果tcp\_nopush为0(也就是配置文件中tcp\_nopush on)，则进入tcp\_nopush的处理。          if (c->tcp\_nopush == NGX\_TCP\_NOPUSH\_UNSET              && header.nelts != 0              && cl              && cl->buf->in\_file)          {  .....................................  //如果nodelay有设置，则进入相关处理，也就是关闭nagle算法。                          if (c->tcp\_nodelay == NGX\_TCP\_NODELAY\_SET) {                                                  tcp\_nodelay = 0;                             if (setsockopt(c->fd, IPPROTO\_TCP, TCP\_NODELAY,                                 (const void \*) &tcp\_nodelay, sizeof(int)) == -1)                                 {                                     .....................                                   }                                   else {  //如果设置成功，则设置nodelay为NGX\_TCP\_NODELAY\_UNSET.这样下次就不会进入nodelay的处理。                                       c->tcp\_nodelay = NGX\_TCP\_NODELAY\_UNSET;                                       ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,                                     "no tcp\_nodelay");                                   }                           }  //如果tcp\_nodelay没有设置，则进入tcp\_nopush的处理。           if (c->tcp\_nodelay == NGX\_TCP\_NODELAY\_UNSET) {  //设置tcp\_nopush(tcp\_cork)                  if (ngx\_tcp\_nopush(c->fd) == NGX\_ERROR) {                      err = ngx\_errno;                        /\*                       \* there is a tiny chance to be interrupted, however,                       \* we continue a processing without the TCP\_CORK                       \*/                        if (err != NGX\_EINTR) {                          wev->error = 1;                          ngx\_connection\_error(c, err,                                               ngx\_tcp\_nopush\_n " failed");                          return NGX\_CHAIN\_ERROR;                      }                    } else {  //如果设置成功，则将c->tcp\_nopush设置为set，这样当再次需要调用sendfile的时候，就跳过设置tcp\_nopush的部分                      c->tcp\_nopush = NGX\_TCP\_NOPUSH\_SET;                        ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,                                     "tcp\_nopush");                  }              }  ...............................................  } |

通过上面我们看到nginx最终会调用sendfile发送数据，然后当前的请求结束，最终会调用tcp\_close或者tcp\_shutdown(设置linger\_timeout)来关闭连接，而当关闭连接的同时，内核会将写buf中缓存的数据发送出去，这样，就不需要我们关闭tcp\_cork,因为内核会帮我们做这个，于是减少了一次系统调用。

到了这里，会有一个问题，那就是keepalive的情况，这时就会有问题了，因为keepalive是不会关闭连接的，这样，就需要我们显示的调用tcp\_push来将数据push出去。接下来我就来分析上篇介绍keepalive的文章中没有涉及到的部分。

接下来的代码就在ngx\_http\_set\_keepalive中。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | //如果tcp\_nopush为NGX\_TCP\_NOPUSH\_SET，则说明我们需要关闭tcp\_cork,也就是将数据push出去.      if (c->tcp\_nopush == NGX\_TCP\_NOPUSH\_SET) {  //push数据          if (ngx\_tcp\_push(c->fd) == -1) {              ngx\_connection\_error(c, ngx\_socket\_errno, ngx\_tcp\_push\_n " failed");              ngx\_http\_close\_connection(c);              return;          }  //reset tcp\_nopush          c->tcp\_nopush = NGX\_TCP\_NOPUSH\_UNSET;          tcp\_nodelay = ngx\_tcp\_nodelay\_and\_tcp\_nopush ? 1 : 0;        } else {          tcp\_nodelay = 1;      }    //如果tcp\_nodelay存在，则说明我们并没有使用tcp\_nopush，此时如果clcf->tcp\_nodelay被设置，则此时需要重新设置tcp\_nodelay,也就是关闭nagle算法。      if (tcp\_nodelay          && clcf->tcp\_nodelay          && c->tcp\_nodelay == NGX\_TCP\_NODELAY\_UNSET)      {          ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0, "tcp\_nodelay");            if (setsockopt(c->fd, IPPROTO\_TCP, TCP\_NODELAY,                         (const void \*) &tcp\_nodelay, sizeof(int))              == -1)          {  #if (NGX\_SOLARIS)              /\* Solaris returns EINVAL if a socket has been shut down \*/              c->log\_error = NGX\_ERROR\_IGNORE\_EINVAL;  #endif                ngx\_connection\_error(c, ngx\_socket\_errno,                                   "setsockopt(TCP\_NODELAY) failed");                c->log\_error = NGX\_ERROR\_INFO;              ngx\_http\_close\_connection(c);              return;          }  //reset tcp\_nodelay          c->tcp\_nodelay = NGX\_TCP\_NODELAY\_SET;      } |