<https://blog.csdn.net/weiyuefei/article/details/51894715>

没有多少高深的东西，全当记录，虽然简单，但是没有做过测试还是挺容易让人糊涂的  
int nRecvBuf=32\*1024;//设置为32K  
setsockopt(s,SOL\_SOCKET,SO\_RCVBUF,(const char\*)&nRecvBuf,sizeof(int));  
1、通过上面语句可以简单设置缓冲区大小,测试证明：跟epoll结合的时候只有当单次发送的数据全被从缓冲区读完毕之后才会再次被触发，多次发送数据如果没有读取完毕当缓冲区未满的时候数据不会丢失，会累加到后面。  
2、   如果缓冲区未满，同一连接多次发送数据会多次收到EPOLLIN事件。  
单次发送数据>socket缓冲区大小的数据数据会被阻塞分次发送，所以循环接收可以用ENLIGE错误判断。  
3、如果缓冲区满，新发送的数据不会触发epoll事件（也无异常），每次recv都会为缓冲区腾出空间，只有当缓冲区空闲大小能够再次接收数据epollIN事件可以再次被触发  
接收时接收大小为0表示客户端断开（不可能有0数据包触发EPOLLIN）,-1表示异常，针对errorno进行判断可以确定是合理异常还是需要终止的异常，>0而不等于缓冲区大小表示单次发送结束。  
4、 如果中途临时调整接收缓存区大小，并且在上一次中数据没有完全接收到用户空间，数据不会丢失，会累加在一起  
所以总结起来，系统对于数据的完整性还是做了相当的保正，至于稳定性没有作更深一步的测试  
新增加：  
5、如果主accept监听的soctet  
fd也设置为非阻塞，那么单纯靠epoll事件来驱动的服务器模型会存在问题，并发压力下发现，每次accept只从系统中取得第一个，所以如果恰冯多个  
连接同时触发server  
fd的EPOLLIN事件，在返回的event数组中体现不出来，会出现丢失事件的现象，所以当用ab等工具简单的压载就会发现每次都会有最后几条信息得  
不到处理，原因就在于此，我现在的解决办法是将server fd的监听去掉，用一个线程阻塞监听，accept成功就处理检测client  
fd，然后在主线程循环监听client事件，这样epoll在边缘模式下出错的概率就小，测试表明效果明显  
6、对于SIG部分信号还是要做屏蔽处理，不然对方socket中断等正常事件都会引起整个服务的退出  
7、sendfile(fd, f->SL->sendBuffer.inFd, (off\_t  
\*)&f->SL->sendBuffer.offset,  
size\_need);注意sendfile函数的地三个变量是传送地址，偏移量会自动增加，不需要手动再次增加，否则就会出现文件传送丢失现象  
8、单线程epoll驱动模型误解：以前我一直认为单线程是无法处理web服务器这样的有严重网络延迟的服务，但nginx等优秀服务器都是机遇事件驱动  
模型，开始我在些的时候也是担心这些问题，后来测试发现，当client  
socket设为非阻塞模式的时候，从读取数据到解析http协议，到发送数据均在epoll的驱动下速度非常快，没有必要采用多线程，我的单核cpu  
（奔三）就可以达到10000page／second，这在公网上是远远无法达到的一个数字（网络延迟更为严重），所以单线程的数据处理能力已经很高了，  
就不需要多线程了，所不同的是你在架构服务器的时候需要将所有阻塞的部分拆分开来，当epoll通知你可以读取的时候，实际上部分数据已经到了  
socket缓冲区，你所读取用的事件是将数据从内核空间拷贝到用户空间，同理，写也是一样的，所以epoll重要的地方就是将这两个延时的部分做了类似  
的异步处理，如果不需要处理更为复杂的业务，那单线程足以满足1000M网卡的最高要求，这才是单线程的意义。  
  
我以前构建的web服务器就没有理解epoll，采用epoll的边缘触发之后怕事件丢失，或者单线程处理阻塞，所以自己用多线程构建了一个任务调度器，  
所有收到的事件统统压进任无调度器中，然后多任务处理，我还将read和write分别用两个调度器处理，并打算如果中间需要特殊的耗时的处理就增加一套  
调度器，用少量线程＋epoll的方法来题高性能，后来发现read和write部分调度器是多余的，epoll本来就是一个事件调度器，在后面再次缓存  
事件分部处理还不如将epoll设为水平模式，所以多此一举，但是这个调度起还是有用处的  
  
上面讲到如果中间有耗时的工作，比如数据库读写，外部资源请求（文件，socket）等这些操作就不能阻塞在主线程里面，所以我设计的这个任务调度器就有  
用了，在epoll能处理的事件驱动部分就借用epoll的，中间部分采用模块化的设计，用函数指针达到面相对象语言中的“委托”的作用，就可以满足不同  
的需要将任务（fd标识）加入调度器，让多线程循环执行，如果中间再次遇到阻塞就会再次加入自定义的阻塞器，检测完成就加入再次存入调度器，这样就可以将  
多种复杂的任务划分开来，相当于在处理的中间环节在自己购置一个类似于epoll的事件驱动器  
  
9、多系统兼容：我现在倒是觉得与其构建一个多操作系统都支持的服务器不如构建特定系统的，如果想迁移再次改动，因为一旦兼顾到多个系统的化会大大增加系  
统的复杂度，并且不能最优性能，每个系统都有自己的独有的优化选项，所以我觉得迁移的工作量远远小于兼顾的工作量  
10模块化编程，虽然用c还是要讲求一些模块化的设计的，我现在才发现几乎面相对想的语言所能实现的所有高级特性在c里面几乎都有对应的解决办法（暂时发现除了操作符重载），所有学过高级面相对象的语言的朋友不放把模式用c来实现，也是一种乐趣，便于维护和自己阅读  
11、养成注释的好习惯

SO\_RCVBUF SO\_SNDBUF

先明确一个概念：每个TCP socket在内核中都有一个发送缓冲区和一个接收缓冲区，TCP的全双工的工作模式以及TCP的滑动窗口便是依赖于这两个独立的buffer以及此buffer的填充状态。接收缓冲区把数据缓存入内核，应用进程一直没有调用read进行读取的话，此数据会一直缓存在相应socket的接收缓冲区内。再啰嗦一点，不管进程是否读取socket，对端发来的数据都会经由内核接收并且缓存到socket的内核接收缓冲区之中。read所做的工作，就是把内核缓冲区中的数据拷贝到应用层用户的buffer里面，仅此而已。进程调用send发送的数据的时候，最简单情况（也是一般情况），将数据拷贝进入socket的内核发送缓冲区之中，然后send便会在上层返回。换句话说，send返回之时，数据不一定会发送到对端去（和write写文件有点类似），send仅仅是把应用层buffer的数据拷贝进socket的内核发送buffer中。后续我会专门用一篇文章介绍read和send所关联的内核动作。每个UDP socket都有一个接收缓冲区，没有发送缓冲区，从概念上来说就是只要有数据就发，不管对方是否可以正确接收，所以不缓冲，不需要发送缓冲区。

接收缓冲区被TCP和UDP用来缓存网络上来的数据，一直保存到应用进程读走为止。对于TCP，如果应用进程一直没有读取，buffer满了之后，发生的动作是：通知对端TCP协议中的窗口关闭。这个便是滑动窗口的实现。保证TCP套接口接收缓冲区不会溢出，从而保证了TCP是可靠传输。因为对方不允许发出超过所通告窗口大小的数据。 这就是TCP的流量控制，如果对方无视窗口大小而发出了超过窗口大小的数据，则接收方TCP将丢弃它。 UDP：当套接口接收缓冲区满时，新来的数据报无法进入接收缓冲区，此数据报就被丢弃。UDP是没有流量控制的；快的发送者可以很容易地就淹没慢的接收者，导致接收方的UDP丢弃数据报。

以上便是TCP可靠，UDP不可靠的实现。

TCP\_CORK TCP\_NODELAY

这两个选项是互斥的，打开或者关闭TCP的nagle算法，下面用场景来解释

典型的webserver向客户端的应答，应用层代码实现流程粗略来说，一般如下所示：

  if（条件1）{

     向buffer\_last\_modified填充协议内容“Last-Modified: Sat, 04 May 2012 05:28:58 GMT”；

     send（buffer\_last\_modified）；

  }

   if（条件2）{

     向buffer\_expires填充协议内容“Expires: Mon, 14 Aug 2023 05:17:29 GMT”；

     send（buffer\_expires）；

   }

   。。。

   if（条件N）{

     向buffer\_N填充协议内容“。。。”；

     send（buffer\_N）；

   }

对于这样的实现，当前的http应答在执行这段代码时，假设有M（M<=N）个条件都满足，那么会有连续的M个send调用，那是不是下层会依次向客户端发出M个TCP包呢？答案是否定的，包的数目在应用层是无法控制的，并且应用层也是不需要控制的。

我用下列四个假设场景来解释一下这个答案

由于TCP是流式的，对于TCP而言，每个TCP连接只有syn开始和fin结尾，中间发送的数据是没有边界的，多个连续的send所干的事情仅仅是：

假如socket的文件描述符被设置为阻塞方式，而且发送缓冲区还有足够空间容纳这个send所指示的应用层buffer的全部数据，那么把这些数据从应用层的buffer，拷贝到内核的发送缓冲区，然后返回。

假如socket的文件描述符被设置为阻塞方式，但是发送缓冲区没有足够空间容纳这个send所指示的应用层buffer的全部数据，那么能拷贝多少就拷贝多少，然后进程挂起，等到TCP对端的接收缓冲区有空余空间时，通过滑动窗口协议（ACK包的又一个作用----打开窗口）通知TCP本端：“亲，我已经做好准备，您现在可以继续向我发送X个字节的数据了”，然后本端的内核唤醒进程，继续向发送缓冲区拷贝剩余数据，并且内核向TCP对端发送TCP数据，如果send所指示的应用层buffer中的数据在本次仍然无法全部拷贝完，那么过程重复。。。直到所有数据全部拷贝完，返回。

请注意，对于send的行为，我用了“拷贝一次”，send和下层是否发送数据包，没有任何关系。

假如socket的文件描述符被设置为非阻塞方式，而且发送缓冲区还有足够空间容纳这个send所指示的应用层buffer的全部数据，那么把这些数据从应用层的buffer，拷贝到内核的发送缓冲区，然后返回。

假如socket的文件描述符被设置为非阻塞方式，但是发送缓冲区没有足够空间容纳这个send所指示的应用层buffer的全部数据，那么能拷贝多少就拷贝多少，然后返回拷贝的字节数。多涉及一点，返回之后有两种处理方式：

1.死循环，一直调用send，持续测试，一直到结束（基本上不会这么搞）。

2.非阻塞搭配epoll或者select，用这两种东西来测试socket是否达到可发送的活跃状态，然后调用send（高性能服务器必需的处理方式）。

综上，以及请参考本文前述的SO\_RCVBUF和SO\_SNDBUF，你会发现，在实际场景中，你能发出多少TCP包以及每个包承载多少数据，除了受到自身服务器配置和环境带宽影响，对端的接收状态也能影响你的发送状况。

至于为什么说“应用层也是不需要控制发送行为的”，这个说法的原因是：

软件系统分层处理、分模块处理各种软件行为，目的就是为了各司其职，分工。应用层只关心业务实现，控制业务。数据传输由专门的层面去处理，这样应用层开发的规模和复杂程度会大为降低，开发和维护成本也会相应降低。

再回到发送的话题上来：）之前说应用层无法精确控制和完全控制发送行为，那是不是就是不控制了？非也！虽然无法控制，但也要尽量控制！

如何尽量控制？现在引入本节主题----TCP\_CORK和TCP\_NODELAY。

cork:塞子，塞住

nodelay：不要延迟

TCP\_CORK：尽量向发送缓冲区中攒数据，攒到多了再发送，这样网络的有效负载会升高。简单粗暴地解释一下这个有效负载的问题。假如每个包中只有一个字节的数据，为了发送这一个字节的数据，再给这一个字节外面包装一层厚厚的TCP包头，那网络上跑的几乎全是包头了，有效的数据只占其中很小的部分，很多访问量大的服务器，带宽可以很轻松的被这么耗尽。那么，为了让有效负载升高，我们可以通过这个选项指示TCP层，在发送的时候尽量多攒一些数据，把他们填充到一个TCP包中再发送出去。这个和提升发送效率是相互矛盾的，空间和时间总是一堆冤家！！

TCP\_NODELAY：尽量不要等待，只要发送缓冲区中有数据，并且发送窗口是打开的，就尽量把数据发送到网络上去。

很明显，两个选项是互斥的。实际场景中该怎么选择这两个选项呢？再次举例说明

webserver,，[下载](http://www.2cto.com/soft" \t "_blank)服务器（ftp的发送文件服务器），需要带宽量比较大的服务器，用TCP\_CORK。

涉及到交互的服务器，比如ftp的接收命令的服务器，必须使用TCP\_NODELAY。默认是TCP\_CORK。设想一下，用户每次敲几个字节的命令，而下层在攒这些数据，想等到数据量多了再发送，这样用户会等到发疯。这个糟糕的场景有个专门的词汇来形容-----粘（nian拼音二声）包。