高性能网络编程5--IO复用与并发编程

2013年12月04日 15:57:24 阅读数：25458

对于服务器的并发处理能力，我们需要的是：每一毫秒服务器都能及时处理这一毫秒内收到的数百个不同TCP连接上的报文，与此同时，可能服务器上还有数以十万计的最近几秒没有收发任何报文的相对不活跃连接。同时处理多个并行发生事件的连接，简称为并发；同时处理万计、十万计的连接，则是高并发。服务器的并发编程所追求的就是处理的并发连接数目无限大，同时维持着高效率使用CPU等资源，直至物理资源首先耗尽。

并发编程有很多种实现模型，最简单的就是与“线程”捆绑，1个线程处理1个连接的全部生命周期。优点：这个模型足够简单，它可以实现复杂的业务场景，同时，线程个数是可以远大于CPU个数的。然而，线程个数又不是可以无限增大的，为什么呢？因为线程什么时候执行是由操作系统内核调度算法决定的，调度算法并不会考虑某个线程可能只是为了一个连接服务的，它会做大一统的玩法：时间片到了就执行一下，哪怕这个线程一执行就会不得不继续睡眠。这样来回的唤醒、睡眠线程在次数不多的情况下，是廉价的，但如果操作系统的线程总数很多时，它就是昂贵的（被放大了），因为这种技术性的调度损耗会影响到线程上执行的业务代码的时间。举个例子，这时大部分拥有不活跃连接的线程就像我们的国企，它们执行效率太低了，它总是唤醒就睡眠在做无用功，而它唤醒争到CPU资源的同时，就意味着处理活跃连接的民企线程减少获得了CPU的机会，CPU是核心竞争力，它的无效率进而影响了GDP总吞吐量。我们所追求的是并发处理数十万连接，当几千个线程出现时，系统的执行效率就已经无法满足高并发了。

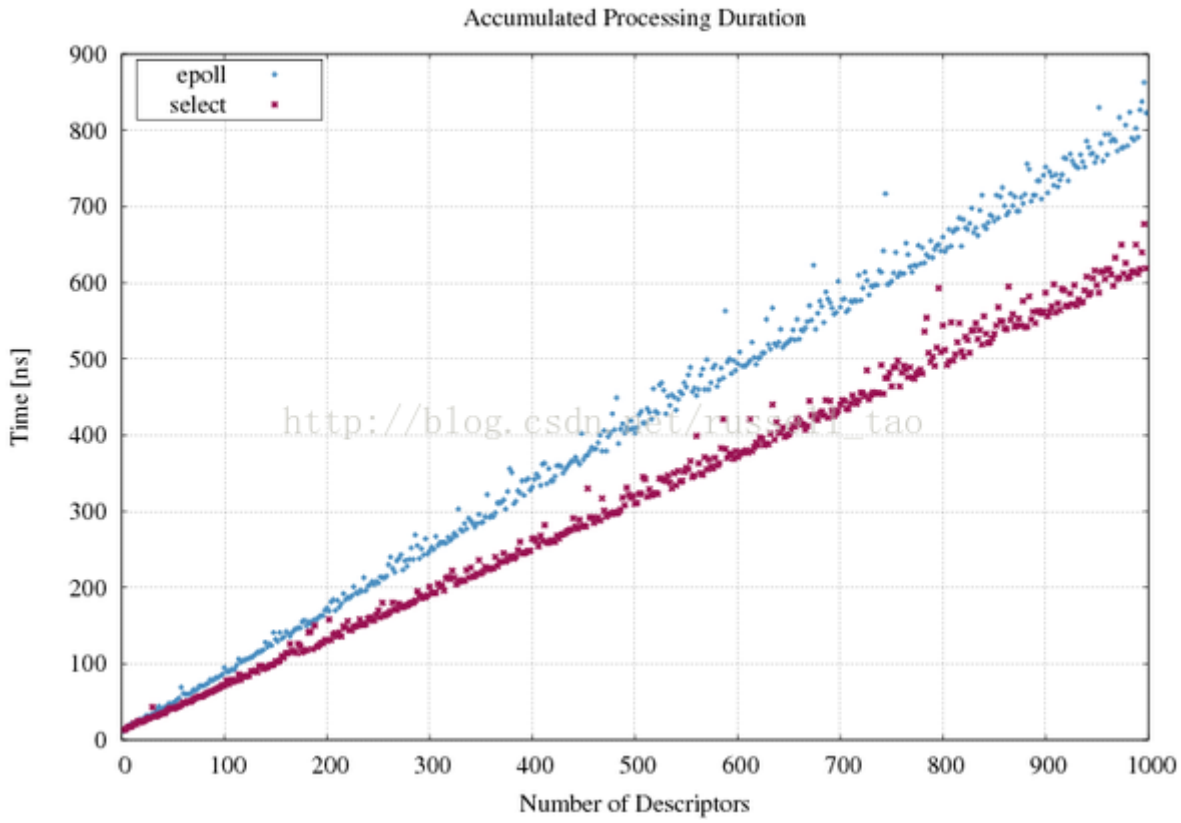
对高并发编程，目前只有一种模型，也是本质上唯一有效的玩法。
从这个系列的前4篇文章可知，连接上的消息处理，可以分为两个阶段：等待消息准备好、消息处理。当使用默认的阻塞套接字时（例如上面提到的1个线程捆绑处理1个连接），往往是把这两个阶段合而为一，这样操作套接字的代码所在的线程就得睡眠来等待消息准备好，这导致了高并发下线程会频繁的睡眠、唤醒，从而影响了CPU的使用效率。

高并发编程方法当然就是把两个阶段分开处理。即，等待消息准备好的代码段，与处理消息的代码段是分离的。当然，这也要求套接字必须是非阻塞的，否则，处理消息的代码段很容易导致条件不满足时，所在线程又进入了睡眠等待阶段。那么问题来了，等待消息准备好这个阶段怎么实现？它毕竟还是等待，这意味着线程还是要睡眠的！解决办法就是，线程主动查询，或者让1个线程为所有连接而等待！
这就是IO多路复用了。多路复用就是处理等待消息准备好这件事的，但它可以同时处理多个连接！它也可能“等待”，所以它也会导致线程睡眠，然而这不要紧，因为它一对多、它可以监控所有连接。这样，当我们的线程被唤醒执行时，就一定是有一些连接准备好被我们的代码执行了，这是有效率的！没有那么多个线程都在争抢处理“等待消息准备好”阶段，整个世界终于清净了！

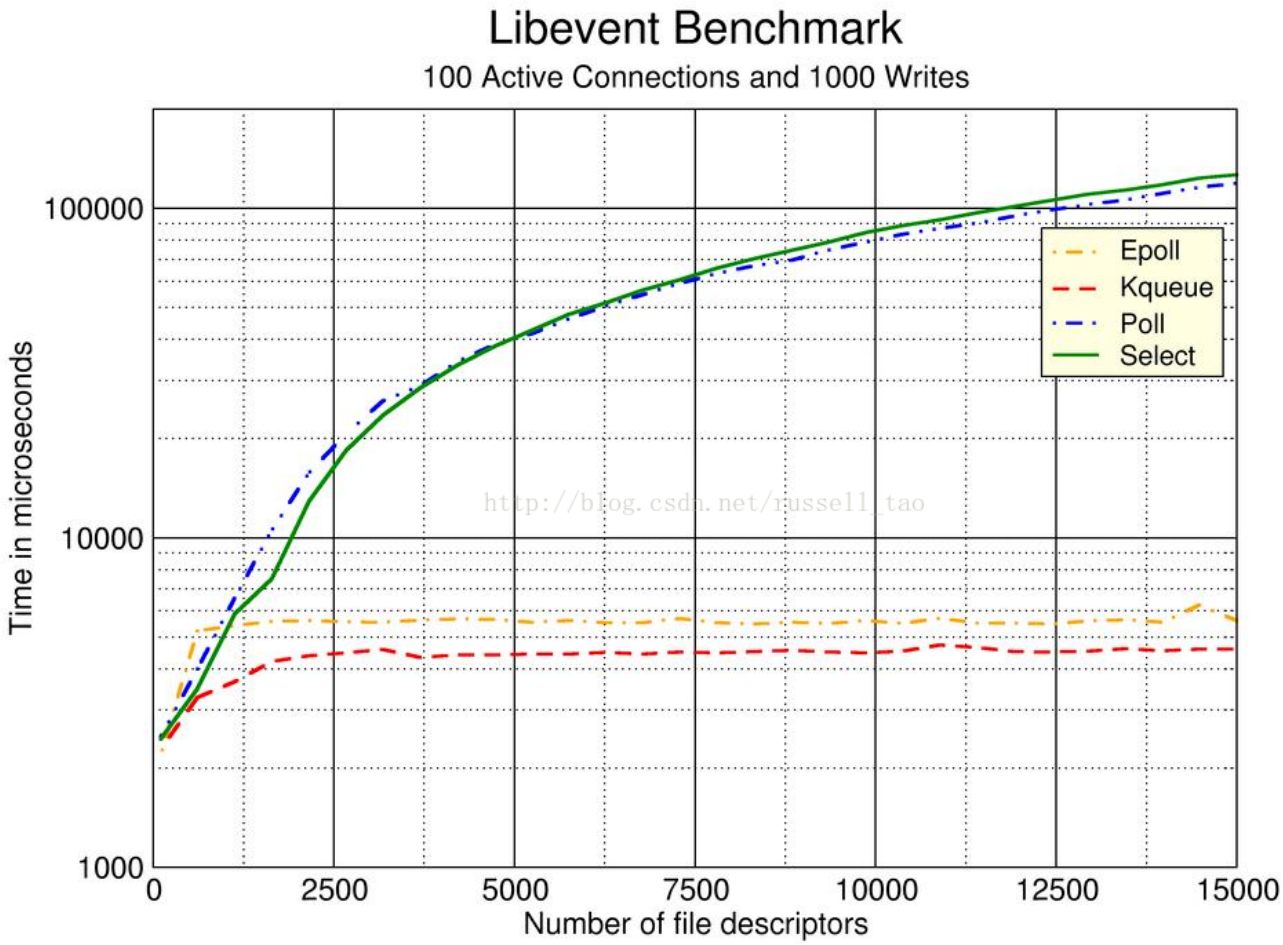
多路复用有很多种实现，在linux上，2.4内核前主要是select和poll，现在主流是epoll，它们的使用方法似乎很不同，但本质是一样的。
效率却也不同，这也是epoll完全替代了select的原因。

简单的谈下epoll为何会替代select。
前面提到过，高并发的核心解决方案是1个线程处理所有连接的“等待消息准备好”，这一点上epoll和select是无争议的。但select预估错误了一件事，就像我们开篇所说，当数十万并发连接存在时，可能每一毫秒只有数百个活跃的连接，同时其余数十万连接在这一毫秒是非活跃的。select的使用方法是这样的：
返回的活跃连接 ==select（全部待监控的连接）
什么时候会调用select方法呢？在你认为需要找出有报文到达的活跃连接时，就应该调用。所以，调用select在高并发时是会被频繁调用的。这样，这个频繁调用的方法就很有必要看看它是否有效率，因为，它的轻微效率损失都会被“频繁”二字所放大。它有效率损失吗？显而易见，全部待监控连接是数以十万计的，返回的只是数百个活跃连接，这本身就是无效率的表现。被放大后就会发现，处理并发上万个连接时，select就完全力不从心了。

看几个图。当并发连接为一千以下，select的执行次数不算频繁，与epoll似乎并无多少差距：



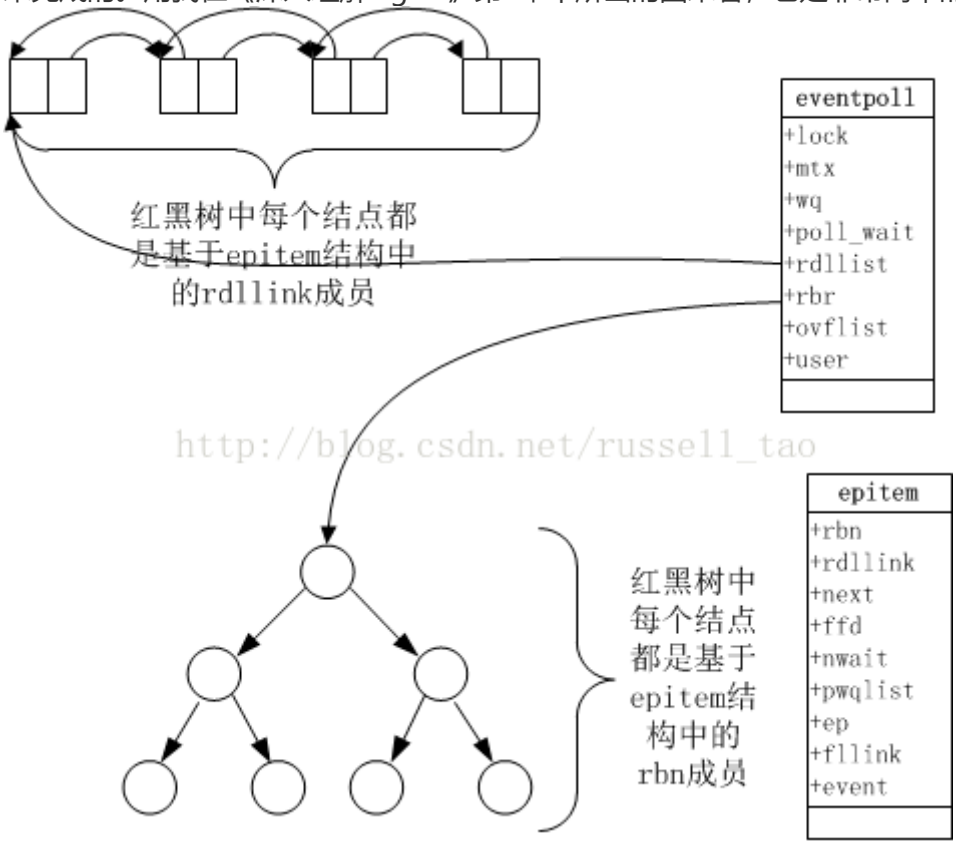
然而，并发数一旦上去，select的缺点被“执行频繁”无限放大了，且并发数越多越明显：



再来说说epoll是如何解决的。它很聪明的用了3个方法来实现select方法要做的事：
新建的epoll描述符==epoll_create()
epoll_ctrl(epoll描述符，添加或者删除所有待监控的连接)

返回的活跃连接 ==epoll_wait （epoll描述符）
这么做的好处主要是：分清了频繁调用和不频繁调用的操作。例如，epoll_ctrl是不太频繁调用的，而epoll_wait是非常频繁调用的。这时，epoll_wait却几乎没有入参，这比select的效率高出一大截，而且，它也不会随着并发连接的增加使得入参越发多起来，导致内核执行效率下降。

epoll是怎么实现的呢？其实很简单，从这3个方法就可以看出，它比select聪明的避免了每次频繁调用“哪些连接已经处在消息准备好阶段”的 epoll_wait时，是不需要把所有待监控连接传入的。这意味着，它在内核态维护了一个数据结构保存着所有待监控的连接。这个数据结构就是一棵红黑树，它的结点的增加、减少是通过epoll_ctrl来完成的。用我在《深入理解Nginx》第8章中所画的图来看，它是非常简单的：



图中左下方的红黑树由所有待监控的连接构成。左上方的链表，同是目前所有活跃的连接。于是，epoll_wait执行时只是检查左上方的链表，并返回左上方链表中的连接给用户。这样，epoll_wait的执行效率能不高吗？

最后，再看看epoll提供的2种玩法ET和LT，即翻译过来的边缘触发和水平触发。其实这两个中文名字倒也有些贴切。这2种使用方式针对的仍然是效率问题，只不过变成了epoll_wait返回的连接如何能够更准确些。
例如，我们需要监控一个连接的写缓冲区是否空闲，满足“可写”时我们就可以从用户态将响应调用write发送给客户端 。但是，或者连接可写时，我们的“响应”内容还在磁盘上呢，此时若是磁盘读取还未完成呢？肯定不能使线程阻塞的，那么就不发送响应了。但是，下一次epoll_wait时可能又把这个连接返回给你了，你还得检查下是否要处理。可能，我们的程序有另一个模块专门处理磁盘IO，它会在磁盘IO完成时再发送响应。那么，每次epoll_wait都返回这个“可写”的、却无法立刻处理的连接，是否符合用户预期呢？

于是，ET和LT模式就应运而生了。LT是每次满足期待状态的连接，都得在epoll_wait中返回，所以它一视同仁，都在一条水平线上。ET则不然，它倾向更精确的返回连接。在上面的例子中，连接第一次变为可写后，若是程序未向连接上写入任何数据，那么下一次epoll_wait是不会返回这个连接的。ET叫做 边缘触发，就是指，只有连接从一个状态转到另一个状态时，才会触发epoll_wait返回它。可见，ET的编程要复杂不少，至少应用程序要小心的防止epoll_wait的返回的连接出现：可写时未写数据后却期待下一次“可写”、可读时未读尽数据却期待下一次“可读”。

当然，从一般应用场景上它们性能是不会有什么大的差距的，ET可能的优点是，epoll_wait的调用次数会减少一些，某些场景下连接在不必要唤醒时不会被唤醒（此唤醒指epoll_wait返回）。但如果像我上面举例所说的，有时它不单纯是一个网络问题，跟应用场景相关。当然，大部分开源框架都是基于ET写的，框架嘛，它追求的是纯技术问题，当然力求尽善尽美。