Linux 下突破限制实现高并发量服务器

1、修改用户进程可打开文件数限制

在 Linux 平台上，无论编写客户端程序还是服务端程序，在进行高并发 TCP 连接处理时，最高的并发数 量都要受到系统对用户单一进程同时可打开文件数量的 限制(这是因为系统为每个 TCP 连接都要创建一个 socket 句柄，每个 socket 句柄同时也是一个文件句柄)。可使用 ulimit 命令查看系统允许当前用户进程打开的文件数限制：

```
[speng@as4 ~]$ ulimit -n
1024
```

这表示当前用户的每个进程最多允许同 时打开 1024 个文件，这 1024 个文件中还得除去每个进程必然打开的标准输入，标准输出，标准错误，服务器监听 socket，进程间通讯的 unix 域 socket 等文件，那么剩下的可用于客户端 socket 连接的文件数就只有大概 1024-10=1014 个左右。 也就是说缺省情况下，基于 Linux 的通讯程序最多允许同时 1014 个 TCP 并发连接。

对于想支持更高数量的 TCP 并发连接的通讯处理 程序，就必须修改 Linux 对当前用户的进程同时打开的文件数量的软限制(soft limit)和硬限制(hardlimit)。其中软限制是指 Linux 在当前系统能够承受的范围内进一步限制用户同时打开的文件数；硬限制则是根据系统 硬件资源状况(主要是系统内存)计算出来的系统最多可同时打开的文件数量。通常软限制小于或等于硬限制。

修改上述限制的最简单的办法就是使用 ulimit 命令：

```
[speng@as4 ~]$ ulimit -n <file_num>
```

上 述命令中，在<file_num>中指定要设置的单一进程允许打开的最大文件数。如果系统回显类似于“Operation notpermitted”之类的话，说明上述限制修改失败，实际上是因为在<file_num>中指定的数值超过了 Linux 系统对该用户 打开文件数的软限制或硬限制。因此，就需要修改 Linux 系统对用户的关于打开文件数的软限制和硬限制。

第一步，修改/etc/security/limits.conf 文件，在文件中添加如下行：

```
speng soft nofile 10240
speng hard nofile 10240
```

其中 speng 指定了要修改哪个用户的打开文件数限制，可用'*'号表示修改所有用户的限制；soft 或 hard 指定要修改软限制还是硬限制；10240 则指定了想要修改的新的限制值，即最大打开文件数(请注意软限制值要小于或等于硬限制)。修改完后保存文件。

第二步，修改/etc/pam.d/login 文件，在文件中添加如下行：

```
session required /lib/security/pam_limits.so
```

这 是告诉 Linux 在用户完成系统登录后，应该调用 pam_limits.so 模块来设置系统对该用户可使用的各种资源数量的最大限制(包括用户可打开的 最大文件数限制)，而 pam_limits.so 模块就会从/etc/security/limits.conf 文件中读取配置来设置这些限制值。修改完 后保存此文件。

第三步，查看 Linux 系统级的最大打开文件数限制，使用如下命令：

```
[speng@as4 ~]$ cat /proc/sys/fs/file-max
12158
```

这 表明这台 Linux 系统最多允许同时打开(即包含所有用户打开文件数总和)12158 个文件，是 Linux 系统级硬限制，所有用户级的打开文件数限制都 不应超过这个数值。通常这个系统级硬限制是 Linux 系统在启动时根据系统硬件资源状况计算出来的最佳的最大同时打开文件数限制，如果没有特殊需要，不应 该修改此限制，除非想为用户级打开文件数限制设置超过此限制的值。修改此硬限制的方法是修改/etc/rc.local 脚本，在脚本中添加如下行：

```
echo 22158 > /proc/sys/fs/file-max
```

这是让 Linux 在启动完成后强行将系统级打开文件数硬限制设置为 22158。修改完后保存此文件。

完 成上述步骤后重启系统，一般情况下就可以将 Linux 系统对指定用户的单一进程允许同时打开的最

大文件数限制设为指定的数值。如果重启后用 ulimit-n 命令查看用户可打开文件数限制仍然低于上述步骤中设置的最大值，这可能是因为在用户登录脚本/etc/profile 中使用 ulimit -n 命令已经将用户可同时打开的文件数做了限制。由于通过 ulimit-n 修改系统对用户可同时打开文件的最大数限制时，新修改的值只能小于或等于上次 ulimit-n 设置的值，因此想用此命令增大这个限制值是不可能的。所以，如果有上述问题存在，就只能去打开/etc/profile 脚本文件，在文件 中查找是否使用了 ulimit-n 限制了用户可同时打开的最大文件数量，如果找到，则删除这行命令，或者将其设置的值改为合适的值，然后保存文件，用户退 出并重新登录系统即可。
通过上述步骤，就为支持高并发 TCP 连接处理的通讯处理程序解除关于打开文件数量方面的系统限制。

2、 修改网络内核对 TCP 连接的有关限制

在 Linux 上编写支持高并发 TCP 连接的客户端通讯处理程序时，有时会发现尽管已经解除了系统对用户同时打开文件数的限制，但仍会出现并发 TCP 连接数 增加到一定数量时，再也无法成功建立新的 TCP 连接的现象。出现这种现在的原因有多种。

第 一种原因可能是因为 Linux 网络内核对本地端口号范围有限制。此时，进一步分析为什么无法建立 TCP 连接，会发现问题出在 connect()调用返回 失败，查看系统错误提示消息是“Can't assign requestedaddress”。同时，如果在此时用 tcpdump 工具监视网络，会发现根本没有 TCP 连接时客户端发 SYN 包的网络流量。这些情况 说明问题在于本地 Linux 系统内核中有限制。其实，问题的根本原因在于 Linux 内核的 TCP/IP 协议实现模块对系统中所有的客户端 TCP 连接对应的 本地端口号的范围进行了限制(例如，内核限制本地端口号的范围为 1024~32768 之间)。当系统中某一时刻同时存在太多的 TCP 客户端连接时，由于每 个 TCP 客户端连接都要占用一个唯一的本地端口号(此端口号在系统的本地端口号范围限制中)，如果现有的 TCP 客户端连接已将所有的本地端口号占满，则此时就无法为新的 TCP 客户端连接分配一个本地端口号了，因此系统会在这种情况下在 connect()调用中返回失败，并将错误提示消息设为“Can't assignrequested address”。有关这些控制逻辑可以查看 Linux 内核源代码，以 linux2.6 内核为例，可以查看 tcp_ipv4.c 文件中如下函数：
static int tcp_v4_hash_connect(struct sock *sk)
请注意上述函数中对变量 sysctl_local_port_range 的访问控制。变量 sysctl_local_port_range 的初始化则是在 tcp.c 文件中的如下函数中设置：
void __init tcp_init(void)
内核编译时默认设置的本地端口号范围可能太小，因此需要修改此本地端口范围限制。
第一步，修改/etc/sysctl.conf 文件，在文件中添加如下行：
net.ipv4.ip_local_port_range = 1024 65000
这表明将系统对本地端口范围限制设置为 1024~65000 之间。请注意，本地端口范围的最小值必须大于或等于 1024；而端口范围的最大值则应小于或等 于 65535。修改完后保存此文件。
第二步，执行 sysctl 命令：
[speng@as4 ~]$ sysctl -p
如果系统没有错误提示，就表明新的本地端口范围设置成功。如果按上述端口范围进行设置，则理论上单独一个进程最多可以同时建立 60000 多个 TCP 客户端 连接。

第 二种无法建立 TCP 连接的原因可能是因为 Linux 网络内核的 IP_TABLE 防火墙对最大跟踪的 TCP 连接数有限制。此时程序会表现为在 connect()调用中阻塞，如同死机，如果用 tcpdump 工具监视网络，也会发现根本没有 TCP 连接时客户端发 SYN 包的网络流量。由于 IP_TABLE 防火墙在内核中会对每个 TCP 连接的状态进行跟踪，跟踪信息将会放在位于内核内存中的 conntrackdatabase 中，这个数据库 的大小有限，当系统中存在过多的 TCP 连接时，数据库容量不足，IP_TABLE 无法为新的 TCP 连接建立跟踪信息，于是表现为在 connect()调用 中阻塞。此时就必须修改内核对最大跟踪的 TCP 连接数的限制，方法同修改内核对本地端口号范围的限制是类似的：
第一步，修改/etc/sysctl.conf 文件，在文件中添加如下行：
net.ipv4.ip_conntrack_max = 10240
这表明将系统对最大跟踪的 TCP 连接数限制设置为 10240。请注意，此限制值要尽量小，以节省对内核内存的占用。
第二步，执行 sysctl 命令：

[speng@as4 ~]$ sysctl -p
如果系统没有错误提示，就表明系统对新的最大跟踪的 TCP 连接数限制修改成功。如果按上述参数进行设置，则理论上单独一个进程最多可以同时建立 10000 多个 TCP 客户端连接。

3、使用支持高并发网络 I/O 的编程技术

在 Linux 上编写高并发 TCP 连接应用程序时，必须使用合适的网络 I/O 技术和 I/O 事件分派机制。

可 用的 I/O 技术有同步 I/O，非阻塞式同步 I/O(也称反应式 I/O)，以及异步 I/O。在高 TCP 并发的情形下，如果使用同步 I/O，这会严重阻塞程 序的运转，除非为每个 TCP 连接的 I/O 创建一个线程。但是，过多的线程又会因系统对线程的调度造成巨大开销。因此，在高 TCP 并发的情形下使用同步 I/O 是不可取的，这时可以考虑使用非阻塞式同步 I/O 或异步 I/O。非阻塞式同步 I/O 的技术包括使用 select()，poll()，epoll 等机 制。异步 I/O 的技术就是使用 AIO。

从 I/O 事件分派机制来看，使用 select()是不合适的，因为它所支持的并发连接数有限(通 常在 1024 个以内)。如果考虑性能，poll()也是不 合适的，尽管它可以支持的较高的 TCP 并发数，但是由于其采用“轮询”机制，当并发数较高时，其运行效率相当低，并可能存在 I/O 事件分派不均，导致部分 TCP 连接上的 I/O 出现“饥饿”现象。而如果使用 epoll 或 AIO，则没有上述问题(早期 Linux 内核的 AIO 技术实现是通过在内核中为每个 I/O 请求创建一个线程来实现的，这种实现机制在高并发 TCP 连接的情形下使用其实也有严重的性能问题。但在最新的 Linux 内核中，AIO 的实现已经得 到改进)。

综上所述，在开发支持高并发 TCP 连接的 Linux 应用程序时，应尽量使用 epoll 或 AIO 技术来实现并发的 TCP 连接上的 I/O 控制，这将为提升程序 对高并发 TCP 连接的支持提供有效的 I/O 保证。

Date: 2007-01-31
OS: Red Hat Enterprise Linux AS release 4 (kernel version 2.6.9-5.EL)

五种 I/O 模式

在 Linux/UNIX 下，有下面这五种 I/O 操作方式：
阻塞 I/O
非阻塞 I/O
I/O 多路复用
信号驱动 I/O（SIGIO）
异步 I/O

程序进行输入操作有两步：
等待有数据可以读
将数据从系统内核中拷贝到程序的数据区。

对于一个对套接字的输入操作:
第一步一般来说是,等待数据从网络上传到本地,当数据包到达的时候，数据将会从网络层拷贝到内核的缓存中;
第二步是从内核中把数据拷贝到程序的数据区中

.阻塞 I/O 模式
简单的说，阻塞就是 " 睡眠 " 的同义词
如 你运行上面的 listener 的时候，它只不过是简单的在那里等待接收数据。它调用 recvfrom()函数，但是那个时候（listener 调用 recvfrom()函数的时候），它并没有数据可以接收．所以

recvfrom()函数阻塞在那里（也就是程序停在 recvfrom()函数处睡大 觉）直到有数据传过来阻塞．你应该明白它的意思。

阻塞 I/O 模式是最普遍使用的 I/O 模式。大部分程序使用的都是阻塞模式的 I/O 。
缺省的，一个套接字建立后所处于的模式就是阻塞 I/O 模式。

对于一个 UDP 套接字来说，数据就绪的标志比较简单：
已经收到了一整个数据报
没有收到。

而 TCP 这个概念就比较复杂，需要附加一些其他的变量
一个进程调用 recvfrom ，然后系统调用并不返回知道有数据报到达本地系统，然后系统将数据拷贝到进程的缓存中。
（如果系统调用收到一个中断信号，则它的调用会被中断）我们称这个进程在调用 recvfrom 一直到从 recvfrom 返回这段时间是阻塞的。
当 recvfrom 正常返回时，我们的进程继续它的操作。

.非阻塞模式 I/O
当我们将一个套接字设置为非阻塞模式，我们相当于告诉了系统内核：“当我请求的 I/O 操作不能够马上完成，你想让我的进程进行休眠等待的时候，不要这么做，请马上返回一个错误给我。”

如我们开始对 recvfrom 的三次调用，因为系统还没有接收到网络数据，所以内核马上返回一个 EWOULDBLOCK 的错误。
第四次我们调用 recvfrom 函数，一个数据报已经到达了，内核将它拷贝到我们的应用程序的缓冲区中，然后 recvfrom 正常返回，我们就可以对接收到的数据进行处理了。

当一个应用程序使用了非阻塞模式的套接字，它需要使用一个循环来不听的测试是否一个文件描述符有数据可读（称做 polling）。
应用程序不停的 polling 内核来检查是否 I/O 操作已经就绪。这将是一个极浪费 CPU 资源的操作。
这种模式使用中不是很普遍

.I/O 多路复用 select()
在使用 I/O 多路技术的时候，我们调用 select()函数和 poll()函数，在调用它们的时候阻塞，而不是我们来调用 recvfrom（或 recv）的时候阻塞。
当我们调用 select 函数阻塞的时候，select 函数等待数据报套接字进入读就绪状态。当 select 函数返回的时候，也就是套接字可以读取数据的时候。这时候我们就可以调用 recvfrom 函数来将数据拷贝到我们的程序缓冲区中。
和阻塞模式相比较，select()和 poll()并没有什么高级的地方，而且，在阻塞模式下只需要调用一个函数：读取或发送，在使用了多路复用技术后， 我们需要调用两个函数了：先调用 select()函数或 poll()函数，然后才能进行真正的读写。

多路复用的高级之处在于，它能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回

假 设我们运行一个网络客户端程序，要同时处理套接字传来的网络数据又要处理本地的标准输入输出。在我们的程序处于阻塞状态等待标准输入的数据的时候，假如 服务器端的程序被 kill（或是自己 Down 掉了），那么服务器程端的 TCP 协议会给客户端（我们这端）的 TCP 协议发送一个 FIN 数据代表终止连接。但是我们的程序阻塞在等待标准输入的数据上，在它读取套接字数据之前（也许是很长一段时间），它不会看见结束标志．我们就不能够使用阻 塞模式的套接字。

I/O 多路技术一般在下面这些情况中被使用：

当一个客户端需要同时处理多个文件描述符的输入输出操作的时候（一般来说是标准的输入输出和网络套接字）， I/O 多路复用技术将会有机会得到使用。
当程序需要同时进行多个套接字的操作的时候。
如果一个 TCP 服务器程序同时处理正在侦听网络连接的套接字和已经连接好的套接字。
如果一个服务器程序同时使用 TCP 和 UDP 协议。
如果一个服务器同时使用多种服务并且每种服务可能使用不同的协议（比如 inetd 就是这样的）。

I/O 多路服用技术并不只局限与网络程序应用上。几乎所有的程序都可以找到应用 I/O 多路复用的地方。