24 | C10K问题：高并发模型设计

2019-10-02 盛延敏

网络编程实战 进入课程 >

讲述：冯永吉
时长 10:05 大小 9.24M

你好，我是盛延敏，这里是网络编程实战第 24 讲，欢迎回来。

在性能篇的前 4 讲里，我们陆续讲解了 select、poll、epoll 等几种 I/O 多路复用技术，以及非阻塞 I/O 模型，为高性能网络编程提供了必要的知识储备。这一讲里，我们了解一下历史上有名的 C10K 问题，并借着 C10K 问题系统地梳理一下高性能网络编程的方法论。

C10K 问题

随着互联网的蓬勃发展，一个非常重要的问题摆在计算机工业界面前。这个问题就是如何使用最低的成本满足高性能和高并发的需求。这个问题在过去可能不是一个严重的问题，但是在 2000 年前后，互联网用户的人数井喷，如果说之前单机服务的用户数量还保持在一个比较低的水平，比如说只有上百个用户，那么在互联网逐渐普及的情况下，服务于成千上万个

用户就将是非常普遍的情形，在这种情形下，如果还按照之前单机的玩法，成本就将超过人们想象，只有超级有钱的大玩家才可以继续下去。

于是，C10K 问题应运而生。C10K 问题是这样的：如何在一台物理机上同时服务 10000 个用户？这里 C 表示并发，10K 等于 10000。得益于操作系统、编程语言的发展，在现在的条件下，普通用户使用 Java Netty、Libevent 等框架或库就可以轻轻松松写出支持并发超过 10000 的服务器端程序，甚至于经过优化之后可以达到十万，乃至百万的并发，但在二十年前，突破 C10K 问题可费了不少的心思，是一个了不起的突破。

C10K 问题是由一个叫 Dan Kegel 的工程师提出并总结归纳的，你可以通过访问 http://www.kegel.com/c10k.html"> 这个页面来获得最新有关这方面的信息。

操作系统层面

C10K 问题本质上是一个操作系统问题，要在一台主机上同时支持 1 万个连接，意味着什么呢？需要考虑哪些方面？

文件句柄

首先，通过前面的介绍，我们知道每个客户连接都代表一个文件描述符，一旦文件描述符不够用了，新的连接就会被放弃，产生如下的错误：

复制代码

```
Socket/File:Can't open so many files
```

在 Linux 下，单个进程打开的文件句柄数是有限制的，没有经过修改的值一般都是 1024。

复制代码

```
$ulimit -n
1024
```

这意味着最多可以服务的连接数上限只能是 1024。不过，我们可以对这个值进行修改，比如用 root 权限修改 /etc/sysctl.conf 文件，使得系统可用支持 10000 个描述符上限。

复制代码

```
fs.file-max = 10000
net.ipv4.ip_conntrack_max = 10000
net.ipv4.netfilter.ip_conntrack_max = 10000
```

系统内存

每个 TCP 连接占用的资源可不止一个连接套接字这么简单，在前面的章节中，我们多少接触到了类似发送缓冲区、接收缓冲区这些概念。每个 TCP 连接都需要占用一定的发送缓冲区和接收缓冲区。

我在文稿里放了一段 shell 代码，分别显示了在 Linux 4.4.0 下发送缓冲区和接收缓冲区的值。

复制代码

```
$cat   /proc/sys/net/ipv4/tcp_wmem
4096    16384   4194304
$ cat   /proc/sys/net/ipv4/tcp_rmem
4096    87380   6291456
```

这三个值分别表示了最小分配值、默认分配值和最大分配值。按照默认分配值计算，一万个连接需要的内存消耗为：

复制代码

```
发送缓冲区：  16384*10000/8 = 20M bytes
接收缓冲区：  87380*10000/8 = 110M bytes
```

当然，我们的应用程序本身也需要一定的缓冲区来进行数据的收发，为了方便，我们假设每个连接需要 128K 的缓冲区，那么 1 万个链接就需要大约 1.2G 的应用层缓冲。

这样，我们可以得出大致的结论，支持 1 万个并发连接，内存并不是一个巨大的瓶颈。

网络带宽

假设 1 万个连接，每个连接每秒传输大约 1KB 的数据，那么带宽需要 10000 x 1KB/s x8 = 80Mbps。这在今天的动辄万兆网卡的时代简直小菜一碟。

C10K 问题解决之道

通过前面我们对操作系统层面的资源分析，可以得出一个结论，在系统资源层面，C10K 问题是可以解决的。

但是，能解决并不意味着可以很好地解决。我们知道，在网络编程中，涉及到频繁的用户态 - 内核态数据拷贝，设计不够好的程序可能在低并发的情况下工作良好，一旦到了高并发情形，其性能可能呈现出指数级别的损失。

举一个例子，如果没有考虑好 C10K 问题，一个基于 select 的经典程序可能在一台服务器上可以很好处理 1000 的并发用户，但是在性能 2 倍的服务器上，却往往并不能很好地处理 2000 的并发用户。

要想解决 C10K 问题，就需要从两个层面上来统筹考虑。

第一个层面，应用程序如何和操作系统配合，感知 I/O 事件发生，并调度处理在上万个套接字上的 I/O 操作？前面讲过的阻塞 I/O、非阻塞 I/O 讨论的就是这方面的问题。

第二个层面，应用程序如何分配进程、线程资源来服务上万个连接？这在接下来会详细讨论。

这两个层面的组合就形成了解决 C10K 问题的几种解法方案，下面我们一起来看。

阻塞 I/O + 进程

这种方式最为简单直接，每个连接通过 fork 派生一个子进程进行处理，因为一个独立的子进程负责处理了该连接所有的 I/O，所以即便是阻塞 I/O，多个连接之间也不会互相影响。

这个方法虽然简单，但是效率不高，扩展性差，资源占用率高。

下面的伪代码描述了使用阻塞 I/O，为每个连接 fork 一个进程的做法：

```
do{
   accept connections
   fork for conneced connection fd
   process_run(fd)
}
```

虽然这个方式比较传统， 但是可以很好地帮我们理解父子进程、僵尸进程等，我们将在下一讲中详细讲一下如何使用这个技术设计一个服务器端程序。

阻塞 I/O + 线程

进程模型占用的资源太大，幸运的是，还有一种轻量级的资源模型，这就是线程。

通过为每个连接调用 pthread_create 创建一个单独的线程，也可以达到上面使用进程的效果。

复制代码

```
do{
   accept connections
   pthread_create for conneced connection fd
   thread_run(fd)
}while(true)
```

因为线程的创建是比较消耗资源的，况且不是每个连接在每个时刻都需要服务，因此，我们可以预先通过创建一个线程池，并在多个连接中复用线程池来获得某种效率上的提升。

复制代码

```
create thread pool
do{
   accept connections
   get connection fd
   push_queue(fd)
}while(true)
```

我将在第 26 讲中详细讲解这部分内容。

非阻塞 I/O + readiness notification + 单线程

应用程序其实可以采取轮询的方式来对保存的套接字集合进行挨个询问，从而找出需要进行 I/O 处理的套接字，像文稿中给出的伪码一样，其中 is_readble 和 is_writeable 可以通过对套接字调用 read 或 write 操作来判断。

复制代码

```
for fd in fdset{
   if(is_readable(fd) == true){
     handle_read(fd)
   }else if(is_writeable(fd)==true){
     handle_write(fd)
   }
}
```

但这个方法有一个问题，如果这个 fdset 有一万个之多，每次循环判断都会消耗大量的 CPU 时间，而且极有可能在一个循环之内，没有任何一个套接字准备好可读，或者可写。

既然这样，CPU 的消耗太大，那么干脆让操作系统来告诉我们哪个套接字可以读，哪个套接字可以写。在这个结果发生之前，我们把 CPU 的控制权交出去，让操作系统来把宝贵的 CPU 时间调度给那些需要的进程，这就是 select、poll 这样的 I/O 分发技术。

于是，程序就长成了这样：

复制代码

```
do {
    poller.dispatch()
    for fd in registered_fdset{
         if(is_readable(fd) == true){
           handle_read(fd)
         }else if(is_writeable(fd)==true){
           handle_write(fd)
     }
}while(ture)
```

第 27 讲中，我将会讨论这样的技术实现。

但是，这样的方法需要每次 dispatch 之后，对所有注册的套接字进行逐个排查，效率并不是最高的。如果 dispatch 调用返回之后只提供有 I/O 事件或者 I/O 变化的套接字，这样排查的效率不就高很多了么？这就是前面我们讲到的 epoll 设计。

于是，基于 epoll 的程序就长成了这样：

复制代码

```
do {
    poller.dispatch()
    for fd_event in active_event_set{
         if(is_readable_event(fd_event) == true){
           handle_read(fd_event)
         }else if(is_writeable_event(fd_event)==true){
           handle_write(fd_event)
     }
}while(ture)
```

Linux 是互联网的基石，epoll 也就成为了解决 C10K 问题的钥匙。FreeBSD 上的 kqueue，Windows 上的 IOCP，Solaris 上的 /dev/poll，这些不同的操作系统提供的功能都是为了解决 C10K 问题。

非阻塞 I/O + readiness notification + 多线程

前面的做法是所有的 I/O 事件都在一个线程里分发，如果我们把线程引入进来，可以利用现代 CPU 多核的能力，让每个核都可以作为一个 I/O 分发器进行 I/O 事件的分发。

这就是所谓的主从 reactor 模式。基于 epoll/poll/select 的 I/O 事件分发器可以叫做 reactor，也可以叫做事件驱动，或者事件轮询（eventloop）。

我没有把基于 select/poll 的所谓“level triggered”通知机制和基于 epoll 的“edge triggered”通知机制分开（C10K 问题总结里是分开的），我觉得这只是 reactor 机制的实现高效性问题，而不是编程模式的巨大区别。

从 27 讲开始，我们就会引入 reactor 模式，并使用一个自己编写的简单 reactor 框架来逐渐掌握它。

异步 I/O+ 多线程

异步非阻塞 I/O 模型是一种更为高效的方式，当调用结束之后，请求立即返回，由操作系统后台完成对应的操作，当最终操作完成，就会产生一个信号，或者执行一个回调函数来完成 I/O 处理。

这就涉及到了 Linux 下的 aio 机制，我们在第 30 讲对 Linux 下的 aio 机制进行简单的讨论。

总结

支持单机 1 万并发的问题被称为 C10K 问题，为了解决 C10K 问题，需要重点考虑两个方面的问题：

- 如何和操作系统配合，感知 I/O 事件的发生？
- 如何分配和使用进程、线程资源来服务上万个连接？

基于这些组合，产生了一些通用的解决方法，在 Linux 下，解决高性能问题的利器是非阻塞 I/O 加上 epoll 机制，再利用多线程能力。

思考题

最后给大家布置两道思考题：

第一道，查询一下资料，看看著名的 Netty 网络编程库，用的是哪一种 C10K 解决方法呢？

第二道，现在大家又把眼光放到了更有挑战性的 C10M 问题，即单机处理千万级并发，你认为能实现吗？挑战和瓶颈又在哪里呢？

欢迎你在评论区写下你对这两个问题的思考，我会和你一起交流，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

上一篇　23 | Linux利器：epoll的前世今生

下一篇　25 | 使用阻塞I/O和进程模型：最传统的方式

精选留言 (4)

写留言

衬衫的价格是19美元
2019-10-04

c10M的话，前面分析的缓存大小和带宽分别要放大1000倍，也就是，需要1.2 * 1000 = 1.2T的内存，80Mbps *1000 = 80Gbps 的带宽, 同时，连接数也要达到10000000

张立华
2019-10-03

C10M，10个处理epoll队列的线程。 每个线程处理一个epoll队列，每个epoll队列容纳最多100万个socket

刘丹

2019-10-02

可以介绍一下C1M的解决方案吗？毕竟C10M很少有需求。


程序水果宝

2019-10-02

C10M问题应该不能再通过应用层和硬件资源的优化来解决了，性能瓶颈应该是冗长的内核协议栈了，要通过已经有的解决方案有dpdk

展开