<<Linux多线程服务端编程>>

图书基本信息

书名：<<Linux多线程服务端编程>>

13位ISBN编号：9787121192821

10位ISBN编号：7121192829

出版时间：2013-1-15

出版时间：电子工业出版社

作者：陈硕

页数：616

前言

本书主要讲述采用现代C++ 在x86-64 Linux上编写多线程TCP网络服务程序的主流常规技术，这也是我对过去5年编写生产环境下的多线程服务端程序的经验总结。
本书重点讲解多线程网络服务器的一种IO 模型，即one loop per thread。
这是一种适应性较强的模型，也是Linux 下以native语言编写用户态高性能网络程序最成熟的模式，掌握之后可顺利地开发各类常见的服务端网络应用程序。
本书以muduo网络库为例，讲解这种编程模型的使用方法及注意事项。
muduo 是一个基于非阻塞IO 和事件驱动的现代C++ 网络库，原生支持oneloop per thread 这种IO 模型。
muduo 适合开发Linux 下的面向业务的多线程服务端网络应用程序，其中“ 面向业务的网络编程 ”的定义见附录A。
“ 现代C++ ”指的不是C++11 新标准，而是2005 年TR1 发布之后的C++ 语言和库。
与传统C++ 相比，现代C++ 的变化主要有两方面：资源管理（见第1 章）与事件回调（见第449 页）。
本书不是多线程编程教程，也不是网络编程教程，更不是C++教程。
读者应该已经大致读过《UNIX环境高级编程》、《UNIX网络编程》、《C++Primer》或与之内容相近的书籍。
本书不谈C++11，因为目前（2012年）主流的Linux 服务端发行版的g++版本都还停留在4.4，C++11 进入实用尚需一段时日。
本书适用的硬件环境是主流x86-64 服务器，多路多核CPU、几十GB 内存、千兆以太网互联。
除了第5章讲诊断日志之外，本书不涉及文件IO。
本书分为四大部分，第1 部分“ C++ 多线程系统编程 ”考察多线程下的对象生命期管理、线程同步方法、多线程与C++ 的结合、高效的多线程日志等。
第2 部分“ muduo 网络库 ”介绍使用现成的非阻塞网络库编写网络应用程序的方法，以及muduo的设计与实现。
第3部分“ 工程实践经验谈 ”介绍分布式系统的工程化开发方法和C++在工程实践中的功能特性取舍。
第4部分“ 附录 ”分享网络编程和C++语言的学习经验。
本书的宗旨是贵精不贵多。
掌握两种基本的同步原语就可以满足各种多线程同步的功能需求，还能写出更易用的同步设施。
掌握一种进程间通信方式和一种多线程网络编程模型就足以应对日常开发任务，编写运行于公司内网环境的分布式服务系统。
（本书不涉及分布式存储系统，也不涉及UDP。
）术语与排版范例本书大量使用英文术语，甚至有少量英文引文。
设计模式的名字一律用英文，例如Observer、Reactor、Singleton。
在中文术语不够突出时，也会使用英文，例如class、heap、event loop、STLalgorithm等。
注意几个中文C++术语：对象实体（instance）、函数重载决议（resolution）、模板具现化（instantiation）、覆写（override）虚函数、提领（dereference）指针。
本书中的英语可数名词一般不用复数形式，例如两个class，6 个syscall；但有时会用(s) 强调中文名词是复数。
fd 是文件描述符（file descriptor）的缩写。
“ CPU数目 ”一般指的是核（core）的数目。
容量单位kB、MB、GB表示的字节数分别为103、106、109，在特别强调准确数值时，会分别用KiB、MiB、GiB 表示210、220、230 字节。
用诸如 § 11.5 表示本书第11.5 节，L42 表示上下文中出现的第42 行代码。
[JCP]、[CC2e] 等是参考文献，见书末清单。

<<Linux多线程服务端编程>>

内容概要

《Linux多线程服务端编程：使用muduo C++网络库》主要讲述采用现代C++在x86-64 Linux上编写多线程TCP网络服务程序的主流常规技术，重点讲解一种适应性较强的多线程服务器的编程模型，即one loop per thread。
这是在Linux下以native语言编写用户态高性能网络程序最成熟的模式，掌握之后可顺利地开发各类常见的服务端网络应用程序。
本书以muduo网络库为例，讲解这种编程模型的使用方法及注意事项。
《Linux多线程服务端编程：使用muduo C++网络库》的宗旨是贵精不贵多。
掌握两种基本的同步原语就可以满足各种多线程同步的功能需求，还能写出更易用的同步设施。
掌握一种进程间通信方式和一种多线程网络编程模型就足以应对日常开发任务，编写运行于公司内网环境的分布式服务系统。

作者简介

陈硕，北京师范大学硕士，擅长C++多线程网络编程和实时分布式系统架构。
曾在摩根士丹利IT部门工作5年，从事实时外汇交易系统开发。
现在在美国加州硅谷某互联网大公司工作，从事大规模分布式系统的可靠性工程。
编写了开源C++网络库muduo，参与翻译了《代码大全（第2版）》和《C++编程规范（繁体版）》，整理了《C++ Primer（第4版）（评注版）》，并曾多次在各地技术大会演讲。

<<Linux多线程服务端编程>>

书籍目录

章节摘录

接使用过Sockets API。
对于TCP 网络编程，我认为核心是处理“ 三个半事件 ”， 见 § 6.4.1 “ TCP 网络编程本质论 ”。
程序员的主要工作是在事件处理函数中实现业务逻辑，而不是和Sockets API “ 较劲 ”。
这里还是没有说清楚“ 网络编程 ”是什么，请继续阅读后文 § A.1.5 “ 网络编程的各种任务角色 ”。
A.1.2 学习网络编程有用吗 以上说的是比较底层的网络编程，程序代码直接面对从TCP 或UDP 收到的数据以及构造数据包发出去。
在实际工作中，另一种常见的情况是通过各种client library来与服务端打交道，或者在现成的框架中填空来实现server，或者采用更上层的通信方式。
比如用libmemcached 与memcached 打交道，使用libpq 来与PostgreSQL 打交道，编写Servlet 来响应HTTP 请求，使用某种RPC 与其他进程通信，等等。
这些情况都会发生网络通信，但不一定算作“ 网络编程 ”。
如果你的工作是前面列举的这些，学习TCP/IP 网络编程还有用吗？
我认为还是有必要学一学，至少在troubleshooting 的时候有用。
无论如何，这些library 或framework 都会调用底层的Sockets API 来实现网络功能。
当你的程序遇到一个线上问题时，如果你熟悉Sockets API，那么从strace 不难发现程序卡在哪里，尽管可能你没有直接调用这些Sockets API。
另外，熟悉TCP/IP 协议、会用tcpdump 也非常有助于分析解决线上网络服务问题。
A.1.3 在什么平台上学习网络编程 对于服务端网络编程，我建议在Linux 上学习。
如果在10年前，这个问题的答案或许是FreeBSD，因为FreeBSD “ 根正苗红 ”， 在2000年那一次互联网浪潮中扮演了重要角色，是很多公司首选的免费服务器操作系统。
2000 年那会儿Linux 还远未成熟，连epoll 都还没有实现。
（FreeBSD 在2001年发布4.1 版，加入了kqueue，从此C10k 不是问题。
） 10年后的今天，事情起了一些变化，Linux成为市场份额最大的服务器操作系统。
在Linux这种大众系统上学网络编程，遇到什么问题会比较容易解决。
因为用的人多，你遇到的问题别人多半也遇到过；同样因为用的人多，如果真的有什么内核bug，很快就会得到修复，至少有work around 的办法。
如果用别的系统，可能一个问题发到论坛上半个月都不会有人理。
从内核源码的风格看，FreeBSD 更干净整洁，注释到位，但是无奈它的市场份额远不如Linux，学习Linux 是更好的技术投资。
A.1.4 可移植性重要吗 写网络程序要不要考虑移植性？
要不要跨平台？
这取决于项目需要，如果贵公司做的程序要卖给其他公司，而对方可能使用Windows、Linux、FreeBSD、Solaris、AIX、HP-UX 等等操作系统，这时候当然要考虑移植性。
如果编写公司内部的服务器上用的网络程序，那么大可只关注一个平台，比如Linux。
因为编写和维护可移植的网络程序的代价相当高，平台间的差异可能远比想象中大，即便是POSIX 系统之间也有不小的差异（比如Linux 没有SO_NOSIGPIPE 选项，Linux 的pipe（2） 是单向的，而FreeBSD 是双向的），错误的返回码也大不一样。
我就不打算把muduo 往Windows 或其他操作系统移植。
如果需要编写可移植的网络程序，我宁愿用libevent、libuv、Java Netty 这样现成的库，把“ 脏活、累活 ”留给别人。
A.1.5 网络编程的各种任务角色 计算机网络是个big topic，涉及很多人物和角色，既有开发人员，也有运维人员。
比方说：公司内部两台机器之间ping 不通，通常由网络运维人员解决，看看是布线有问题还是路由器设置不对；两台机器能ping 通，但是程序连不上，经检查是本机防火墙设置有问题，通常由系统管理员解决；两台机器能连上，但是丢包很严重，发现是网卡或者交换机的网口故障，由硬件维修人员解决；两台机器的程序能连上，但是偶尔发过去的请求得不到响应，通常是程序bug，应该由开发人员解决。

本文主要关心开发人员这一角色。
下面简单列出一些我能想到的跟网络打交道的编程任务，其中前三项是面向网络本身，后面几项是在计算机网络之上构建信息系统。
1.开发网络设备，编写防火墙、交换机、路由器的固件（firmware）。
2.开发或移植网卡的驱动。
3.移植或维护TCP/IP 协议栈（特别是在嵌入式系统上）。
4.开发或维护标准的网络协议程序，HTTP、FTP、DNS、SMTP、POP3、NFS。
5.开发标准网络协议的“ 附加品 ”，比如HAProxy、squid、varnish 等Web loadbalancer。
6.开发标准或非标准网络服务的客户端库，比如ZooKeeper 客户端库、memcached客户端库。
7.开发与公司业务直接相关的网络服务程序，比如即时聊天软件的后台服务器、网游服务器、金融交易系统、互联网企业用的分布式海量存储、微博发帖的内部广播通知等等。
8.客户端程序中涉及网络的部分，比如邮件客户端中与POP3、SMTP 通信的部分，以及网游的客户端程序中与服务器通信的部分。
本文所指的“ 网络编程 ”专指第7 项，即在TCP/IP 协议之上开发业务软件。
换句话说，不是用Sockets API 开发muduo 这样的网络库，而是用libevent、muduo、Netty、gevent 这样现成的库开发业务软件，muduo 自带的十几个示例程序是业务软件的代表。
A.1.6 面向业务的网络编程的特点 与通用的网络服务器不同，面向公司业务的专用网络程序有其自身的特点。
业务逻辑比较复杂，而且时常变化 如果写一个HTTP 服务器，在大致实现HTTP 1.1 标准之后，程序的主体功能一般不会有太大的变化，程序员会把时间放在性能调优和bug 修复上。
而开发针对公司业务的专用程序时，功能说明书（spec）很可能不如HTTP 1.1 标准那么细致明确。
更重要的是，程序是快速演化的。
以即时聊天工具的后台服务器为例，可能第一版只支持在线聊天；几个月之后发布第二版，支持离线消息；又过了几个月，第三版支持隐身聊天；随后，第四版支持上传头像；如此等等。
这要求程序员能快速响应新的业务需求，公司才能保持竞争力。
由于业务时常变化（假设每月一次版本升级），也会降低服务程序连续运行时间的要求。
相反，我们要设计一套流程，通过轮流重启服务器来完成平滑升级（§9.2.2）。
不一定需要遵循公认的通信协议标准 比方说网游服务器就没什么协议标准，反正客户端和服务端都是本公司开发的，如果发现目前的协议设计有问题，两边一起改就行了。
由于可以自己设计协议，因此我们可以绕开一些性能难点，简化程序结构。
比方说，对于多线程的服务程序，如果用短连接TCP 协议，为了优化性能通常要精心设计accept 新连接的机制2，避免惊群并减少上下文切换。
但是如果改用长连接，用最简单的单线程accept 就行了。
程序结构没有定论 对于高并发大吞吐的标准网络服务，一般采用单线程事件驱动的方式开发，比如HAProxy、lighttpd 等都是这个模式。
但是对于专用的业务系统，其业务逻辑比较复杂，占用较多的CPU 资源，这种单线程事件驱动方式不见得能发挥现在多核处理器的优势。
这留给程序员比较大的自由发挥空间，做好了“ 横扫千军 ”，做烂了一败涂地。
我认为目前one loop per thread 是通用性较高的一种程序结构，能发挥多核的优势，见§3.3 和§6.6。
性能评判的标准不同 如果开发httpd 这样的通用服务，必然会和开源的Nginx、lighttpd 等高性能服务器比较，程序员要投入相当的精力去优化程序，才能在市场上占有一席之地。
而面向业务的专用网络程序不一定是IO bound，也不一定有开源的实现以供对比性能，优化方向也可能不同。
程序员通常更加注重功能的稳定性与开发的便捷性。
性能只要一代比一代强即可。
网络编程起到支撑作用，但不处于主导地位 程序员的主要工作是实现业务逻辑，而不只是实现网络通信协议。

这要求程序员深入理解业务。
程序的性能瓶颈不一定在网络上，瓶颈有可能是CPU、Disk IO、数据库等，这时优化网络方面的代码并不能提高整体性能。
只有对所在的领域有深入的了解，明白各种因素的权衡（trade-off），才能做出一些有针对性的优化。
现在的机器上，简单的并发长连接echo服务程序不用特别优化就做到十多万qps，但是如果每个业务请求需要1ms 密集计算，在8 核机器上充其量能达到8000 qps，优化IO 不如去优化业务计算（如果投入产出合算的话）。
A.1.7 几个术语 互联网上的很多“ 口水战 ”是由对同一术语的不同理解引起的，比如我写的《多线程服务器的适用场合》3，就曾经被人说是“ 挂羊头卖狗肉 ”，因为这篇文章中举的master 例子“ 根本就算不上是个网络服务器。
因为它的瓶颈根本就跟网络无关。
” 网络服务器“ 网络服务器 ”这个术语确实含义模糊，到底指硬件还是软件？
到底是服务于网络本身的机器（交换机、路由器、防火墙、NAT），还是利用网络为其他人或程序提供服务的机器（打印服务器、文件服务器、邮件服务器）？
每个人根据自己熟悉的领域，可能会有不同的解读。
比方说，或许有人认为只有支持高并发、高吞吐量的才算是网络服务器。
为了避免无谓的争执，我只用“ 网络服务程序 ”或者“ 网络应用程序 ”这种含义明确的术语。
“ 开发网络服务程序 ”通常不会造成误解。
客户端？
服务端？
在TCP 网络编程中，客户端和服务端很容易区分，主动发起连接的是客户端，被动接受连接的是服务端。
当然，这个“ 客户端 ”本身也可能是个后台服务程序，HTTP proxy 对HTTP server 来说就是个客户端。
客户端编程？
服务端编程？
但是“ 服务端编程 ”和“ 客户端编程 ”就不那么好区分了。
比如Web crawler，它会主动发起大量连接，扮演的是HTTP客户端的角色，但似乎应该归入“ 服务端编程 ”。
又比如写一个HTTP proxy，它既会扮演服务端——被动接受Web browser 发起的连接，也会扮演客户端——主动向HTTP server发起连接，它究竟算服务端还是客户端？
我猜大多数人会把它归入服务端编程。
那么究竟如何定义“ 服务端编程 ”？
服务端编程需要处理大量并发连接？
也许是，也许不是。
比如云风在一篇介绍网游服务器的博客4中就谈到，网游中用到的“ 连接服务器 ”需要处理大量连接，而“ 逻辑服务器 ”只有一个外部连接。
那么开发这种网游“ 逻辑服务器 ”算服务端编程还是客户端编程呢？
又比如机房的服务进程监控软件，并发数跟机器数成正比，至多也就是两三千的并发连接。
（再大规模就超出本书的范围了。
） 我认为，“ 服务端网络编程 ”指的是编写没有用户界面的长期运行的网络程序，程序默默地运行在一台服务器上，通过网络与其他程序打交道，而不必和人打交道。
与之对应的是客户端网络程序，要么是短时间运行，比如wget；要么是有用户界面（无论是字符界面还是图形界面）。
本文主要谈服务端网络编程。

编辑推荐

《Linux多线程服务端编程:使用muduo C++网络库》编辑推荐：示范在多核时代采用现代C++编写，多线程TCP网络服务器的正规做法。

版权说明