谈一谈网络编程学习经验

陈硕
giantchen@gmail.com
blog.csdn.net/Solstice
weibo.com/giantchen
2012-02-13

本文谈一谈我在学习网络编程方面的一些个人经验。“网络编程”这个术语的范围很广，本文指用 Sockets API 开发基于 TCP/IP 的网络应用程序，具体定义见“网络编程的各种任务角色”一节。

受限于本人的经历和经验，这篇文章的适应范围是：

- x86-64 Linux 服务端网络编程，直接或间接使用 Sockets API
- 公司内网。不一定是局域网，但总体位于公司防火墙之内，环境可控

本文可能不适合：

- PC 客户端网络编程，程序运行在客户的 PC 上，环境多变且不可控
- Windows 网络编程
- 面向公网的服务程序
- 高性能网络服务器

本文分两个部分：

1. 网络编程的一些胡思乱想，谈谈我对这一领域的认识
2. 几本必看的书，基本上还是 W. Richard Stevents 那几本

另外，本文没有特别说明时均暗指 TCP 协议，“连接”是“TCP 连接”，“服务端”是“TCP 服务端”。

网络编程的一些胡思乱想

以下胡乱列出我对网络编程的一些想法，前后无关联。

网络编程是什么？

网络编程是什么？是熟练使用 Sockets API 吗？说实话，在实际项目里我只用过两次 Sockets API，其他时候都是使用封装好的网络库。

第一次是 2005 年在学校做一个羽毛球赛场计分系统：我用 C# 编写运行在 PC 机上的软件，负责比分的显示；再用 C# 写了运行在 PDA 上的计分界面，记分员拿着 PDA 记录比分；这两部分程序通过 TCP 协议相互通信。这其实是个简单的分布式系统，体育馆有不止一片场地，每个场地都有一名拿 PDA 的记分员，每个场地都有两台显示比分的 PC 机（显示器是 42 吋平板电视，放在场地的对角，这样两边看台的观众都能看到比分）。这两台 PC 机功能不完全一样，一台只负责显示当前比分，另一台还要负责与 PDA 通信，并更新数据库里的比分信息。此外，还有一台 PC 机负责周期性地从数据库读出全部 7 片场地的比分，显示在体育馆墙上的大屏幕上。这台 PC 上还运行着一个程序，负责生成比分数据的静态页面，通过 FTP 上传发布到某门户网站的体育频道。系统中还有一个录入赛程（参赛队，运动员，出场

顺序等）数据库的程序，运行在数据库服务器上。算下来整个系统有十来个程序，运行在二十多台设备（PC 和 PDA）上，还要考虑可靠性。将来有机会把这个小系统仔细讲一讲，挺有意思的。

这是我第一次写实际项目中的网络程序，当时写下来的感觉是像写命令行与用户交互的程序：程序在命令行输出一句提示语，等待客户输入一句话，然后处理客户输入，再输出下一句提示语，如此循环。只不过这里的“客户”不是人，而是另一个程序。在建立好 TCP 连接之后，双方的程序都是 read/write 循环（为求简单，我用的是 blocking 读写），直到有一方断开连接。

第二次是 2010 年编写 muduo 网络库，我再次拿起了 Sockets API，写了一个基于 Reactor 模式的 C++ 网络库。写这个库的目的之一就是想让日常的网络编程从 Sockets API 的琐碎细节中解脱出来，让程序员专注于业务逻辑，把时间用在刀刃上。Muduo 网络库的示例代码包含了十几个网络程序，这些程序都没有直接使用 Sockets API。

在此之外，无论是实习还是工作，虽然我写的程序都会通过 TCP 协议与其他程序打交道，但我没有直接使用过 Sockets API。对于 TCP 网络编程，我认为核心是处理“三个半事件”，见《Muduo 网络编程示例之零：前言》中的“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“网络编程的各种任务角色”。

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。无论你是用 libevent/netty/gevent 来写网络程序，还是用前述更高层库来通信，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，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 也大大有助于分析解决线上网络服务问题。

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，事情起了变化，Linux 成为了市场份额最大的服务器操作系统[1]。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；

[1] http://en.wikipedia.org/wiki/Usage_share_of_operating_systems

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。

可移植性重要吗？

写网络程序要不要考虑移植性？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），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。

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 load balancer
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。

面向业务的网络编程的特点

跟开发通用的网络程序不同，开发面向公司业务的专用网络程序有其特点：

- 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。

- 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，如果发现目前的协议设计有问题，两边一起改了就是了。由于可以自己设计协议，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，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《多线程服务器的常用编程模型》[2]和《Muduo 多线程模型：一个 Sudoku 服务器演变》[3]。

- 性能评判的标准不同

如果开发 httpd 这样的通用网络服务，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 IO，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[4]，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 8k QPS，优化 IO 不如去优化业务计算（如果投入产出划得来的话）。

[2] https://github.com/downloads/chenshuo/documents/multithreaded_server.pdf
[3] http://www.cnblogs.com/Solstice/archive/2011/06/16/2082590.html
[4] 我写的基于 Netty 和 Protobuf 的简易 RPC 经 bluedavy 测试是 14.9 万 QPS，他自己的更高 http://blog.bluedavy.com/?p=334 https://github.com/chenshuo/recipes/tree/master/protorpc

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，比我写的《多线程服务器的适用场合》就曾经人被说是“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 网络编程里边，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。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几万、几十万、上百万并发连接？也许是，也许不是。比如云风在一篇介绍网游服务器的博客[5]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。

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。服务端网络编程有一些通用的模式，可参考前面提到的两篇文章。

7x24 重要吗？内存碎片可怕吗？

一谈到服务端网络编程，有人立刻会提出 7x24 运行的要求。对于某些网络设备而言，这是合理的需求，比如交换机、路由器。对于开发商业系统，我认为要求程序 7x24 运行通常是系统设计上考虑不

[5] http://blog.codingnow.com/2006/04/iocp_kqueue_epoll.html

周。具体见《分布式系统的工程化开发方法》第 20 页起。重要的不是 7x24，而是在程序不必做到 7x24 的情况下也能达到足够高的可用性。一个考虑周到的系统应该允许每个进程都能随时重启，这样才能在廉价的服务器硬件上做到高可用性。

既然不要求 7x24，那么也不必害怕内存碎片，理由如下：

- 64-bit 系统的地址空间足够大，不会出现没有足够的连续空间这种情况。有没有谁能故意制造内存碎片（不是内存泄露）使得服务程序失去响应？
- 现代的内存分配器（malloc 及其第三方实现）今非昔比，除了 memcached 这种纯以内存为卖点的程序需要自己设计分配器之外，其他网络程序大可使用系统自带的 malloc 或者某个第三方实现。（重新发明 memory pool 似乎已经不流行了。）
- Linux Kernel 也大量用到了动态内存分配。既然操作系统内核都不怕动态分配内存造成碎片，应用程序为什么要害怕？应用程序的可靠性只要不低于硬件和操作系统的可靠性就行，普通 PC 服务器的年故障率约为 3%，算一算你的服务程序一年要被意外重启多少次。
- 内存碎片如何度量？有没有什么工具能为当前进程的内存碎片状况评个分？如果不能比较两种方案的内存碎片程度，谈何优化？

有人为了避免所谓的内存碎片，害怕使用 STL 容器，也不敢 new/delete，这算是 premature optimization 还是因噎废食呢？

协议设计是网络编程的核心

对于专用的业务系统，协议设计是核心任务，决定了系统的开发难度与可靠性，但是这个领域还没有形成大家公认的设计流程。

系统中哪个程序发起连接，哪个程序接受连接？如果写标准的网络服务，那么这不是问题，按 RFC 来就行了。自己设计业务系统，有没有章法可循？以网游为例，到底是连接服务器主动连接逻辑服务器，还是逻辑服务器主动连接“连接服务器”？似乎没有定论，两种做法都行。一般可以按照“依赖->被依赖”的关系来设计发起连接的方向。

比新建连接难的是关闭连接。在传统的网络服务中（特别是短连接服务），不少是服务端主动关闭连接，比如 daytime、HTTP/1.0。也有少部分是客户端主动关闭连接，通常是些长连接服务，比如 echo、chargen 等。我们自己的业务系统该如何设计连接关闭协议呢？

服务端主动关闭连接的缺点之一是会多占用服务器资源。服务端主动关闭连接之后会进入 TIME_WAIT 状态，在一段时间之内 hold 住一些内核资源。如果并发访问量很高，这会影响服务端的处理能力。这似乎暗示我们应该把协议设计为客户端主动关闭，让 TIME_WAIT 状态分散到多台客户机器上，化整为零。

这又有另外的问题：客户端赖着不走怎么办？会不会造成拒绝服务攻击？或许有一个二者结合的方案：客户端在收到响应之后就应该主动关闭，这样把 TIME_WAIT 留在客户端。服务端有一个定时器，如果客户端若干秒钟之内没有主动断开，就踢掉它。这样善意的客户端会把 TIME_WAIT 留给自己，buggy 的客户端会把 TIME_WAIT 留给服务端。或者干脆使用长连接协议，这样避免频繁创建销毁连接。

比连接的建立与断开更重要的是设计消息协议。消息格式很好办，XML、JSON、Protobuf 都是很好的选择；难的是消息内容。一个消息应该包含哪些内容？多个程序相互通信如何避免 race condition（见《分布式系统的工程化开发方法》p.16 的例子）？系统的全局状态该如何跃迁？可惜这方面可供参考的

例子不多，也没有太多通用的指导原则，我知道的只有 30 年前提出的 end-to-end principle 和 happens-before relationship。只能从实践中慢慢积累了。

网络编程的三个层次

侯捷先生在《漫談程序員與編程》中讲到 STL 运用的三个档次：“**會用 STL，是一種檔次。對 STL 原理有所了解，又是一個檔次。追蹤過 STL 源碼，又是一個檔次。第三種檔次的人用起 STL 來，虎虎生風之勢絕非第一檔次的人能夠望其項背。**”

我认为网络编程也可以分为三个层次：

1. 读过教程和文档
2. 熟悉本系统 TCP/IP 协议栈的脾气
3. 自己写过一个简单的 TCP/IP stack

第一个层次是基本要求，读过《Unix 网络编程》这样的编程教材，读过《TCP/IP 详解》基本理解 TCP/IP 协议，读过本系统的 manpage。这个层次可以编写一些基本的网络程序，完成常见的任务。但网络编程不是照猫画虎这么简单，若是按照 manpage 的功能描述就能编写产品级的网络程序，那人生就太幸福了。

第二个层次，熟悉本系统的 TCP/IP 协议栈参数设置与优化是开发高性能网络程序的必备条件。摸透协议栈的脾气还能解决工作中遇到的比较复杂的网络问题。拿 Linux 的 TCP/IP 协议栈来说：

- 有可能出现**自连接**（见《学之者生，用之者死——ACE 历史与简评》举的三个硬伤），程序应该有所准备。
- Linux 的内核会有 bug，比如某种 TCP 拥塞控制算法曾经出现 TCP window clamping（窗口箝位）bug，导致吞吐量暴跌，可以选用其他拥塞控制算法来绕开(work around)这个问题。

这些阴暗角落在 manpage 里没有描述，要通过其他渠道了解。

编写可靠的网络程序的关键是熟悉各种场景下的 error code（文件描述符用完了如何？本地 ephemeral port 暂时用完，不能发起新连接怎么办？服务端新建并发连接太快，backlog 用完了，客户端 connect 会返回什么错误？），有的在 manpage 里有描述，有的要通过实践或阅读源码获得。

第三个层次，通过自己写一个简单的 TCP/IP 协议栈，能大大加深对 TCP/IP 的理解，更能明白 TCP 为什么要这么设计，有哪些因素制约，每一步操作的代价是什么，写起网络程序来更是成竹在胸。

其实实现 TCP/IP 只需要操作系统提供三个接口函数：一个函数，两个回调函数。分别是：send_packet()、on_receive_packet()、on_timer()。多年前有一篇文章《使用 libnet 与 libpcap 构造 TCP/IP 协议软件》介绍了在用户态实现 TCP/IP 的方法。lwIP 也是很好的借鉴对象。

如果有时间，我打算自己写一个 Mini/Tiny/Toy/Trivial/Yet-Another TCP/IP。我准备换一个思路，用 TUN/TAP 设备在用户态实现一个能与本机点对点通信的 TCP/IP 协议栈[6]，这样那三个接口函数就表现为我最熟悉的文件读写。在用户态实现的好处是便于调试，协议栈做成静态库，与应用程序链接到一起（库的接口不必是标准的 Sockets API）。做完这一版，还可以继续发挥，用 FTDI 的 USB-SPI 接口芯片连

[6] 《关于 TCP 并发连接的几个思考题与试验》 http://blog.csdn.net/solstice/article/details/6579232

接 ENC28J60 适配器，做一个真正独立于操作系统的 TCP/IP stack。如果只实现最基本的 IP、ICMP Echo、TCP 的话，代码应能控制在 3000 行以内；也可以实现 UDP，如果应用程序需要用到 DNS 的话。

最主要的三个例子

我认为 TCP 网络编程有三个例子最值得学习研究，分别是 echo、chat、proxy，都是长连接协议。

Echo 的作用：熟悉服务端被动接受新连接、收发数据、被动处理连接断开。每个连接是独立服务的，连接之间没有关联。在消息内容方面 Echo 有一些变种：比如做成一问一答的方式，收到的请求和发送响应的内容不一样，这时候要考虑打包与拆包格式的设计，进一步还可以写简单的 HTTP 服务。

Chat 的作用：连接之间的数据有交流，从 a 收到的数据要发给 b。这样对连接管理提出的更高的要求：如何用一个程序同时处理多个连接？fork() per connection 似乎是不行的。如何防止串话？b 有可能随时断开连接，而新建立的连接 c 可能恰好复用了 b 的文件描述符，那么 a 会不会错误地把消息发给 c？

Proxy 的作用：连接的管理更加复杂：既要被动接受连接，也要主动发起连接，既要主动关闭连接，也要被动关闭连接。还要考虑两边速度不匹配，见《Muduo 网络编程示例之十：socks4a 代理服务器》。

这三个例子功能简单，突出了 TCP 网络编程中的重点问题，挨着做一遍基本就能达到层次一的要求。

学习 Sockets API 的利器：IPython

我在编写 muduo 网络库的时候，写了一个命令行交互式的调试工具[7]，方便我试验各个 Sockets API 的返回时机和返回值。后来发现其实可以用 IPython 达到相同的效果，不必自己编程。用交互式工具很快就能摸清各种 IO 事件的发生条件，比反复编译 C 代码高效得多。比方说想简单试验一下 TCP 服务器和 epoll，可以这么写：

```
$ ipython
In [1]: import socket, select
In [2]: s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
In [3]: s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
In [4]: s.bind(('', 5000))
In [5]: s.listen(5)
In [6]: client, address = s.accept()  # client.fileno() == 4

In [7]: client.recv(1024)  # 此处会阻塞
Out[7]: 'Hello\n'

In [8]: epoll = select.epoll()
In [9]: epoll.register(client.fileno(), select.EPOLLIN)  # 试试省略第二个参数

In [10]: epoll.poll(60)    # 此处会阻塞
Out[10]: [(4, 1)]          # 表示第 4 号文件可读（select.EPOLLIN == 1）

In [11]: client.recv(1024) # 已经有数据可读，不会阻塞了
Out[11]: 'World\n'

In [12]: client.setblocking(0)  # 改为非阻塞方式
In [13]: client.recv(1024)      # 没有数据可读，立刻返回，错误码 EAGAIN == 11
error: [Errno 11] Resource temporarily unavailable
```

[7] http://blog.csdn.net/Solstice/article/details/5497814

```
In [14]: epoll.poll(60)          # epoll_wait() 一下
Out[14]: [(4, 1)]

In [15]: client.recv(1024)       # 再去读数据，有了
Out[15]: 'Bye!\n'

In [16]: client.close()
```

同时在另一个命令行窗口用 nc 发送数据。

```
$ nc localhost 5000
Hello
World
Bye!
```

在编写 muduo 的时候，我一般会开四个命令行窗口，其一看 log，其二看 strace，其三用 netcat/tempest/ipython 充作通信对方，其四看 tcpdump。各个工具的输出相互验证，很快就摸清了门道。muduo 是一个基于 Reactor 模式的 Linux C++网络库[8]，采用非阻塞 IO，支持高并发和多线程，核心代码量不大（3000 多行），示例丰富，可供网络编程的学习者参考。

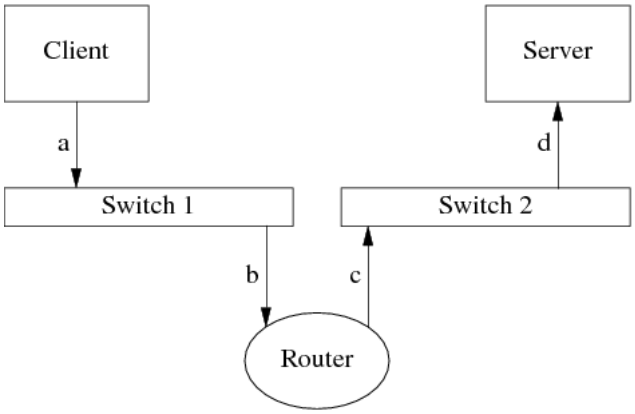
TCP 的可靠性有多高？

TCP 是“面向连接的、可靠的、字节流传输协议”，这里的“可靠”究竟是什么意思？《Effective TCP/IP Programming》第 9 条说：Realize That TCP Is a Reliable Protocol, Not an Infallible Protocol，那么 TCP 在哪种情况下会出错？这里说的“出错”指的是收到的数据与发送的数据不一致，而不是数据不可达。

我在《一种自动反射消息类型的 Google Protobuf 网络传输方案》中设计了带 check sum 的消息格式，很多人表示不理解，认为是多余的。IP header 里边有 check sum，TCP header 也有 check sum，链路层以太网还有 CRC32 校验，那么为什么还需要在应用层做校验？什么情况下 TCP 传送的数据会出错？

IP header 和 TCP header 的 check sum 是一种非常弱的 16-bit check sum 算法，把数据当成反码表示的 16-bit integers，再加到一起。这种 checksum 算法能检出一些简单的错误，而对某些错误无能为力，由于是简单的加法，遇到“和”不变的情况就无法检查出错误（比如交换两个 16-bit 整数，加法满足交换律，结果不变）。以太网的 CRC32 比较强，但它只能保证同一个网段上的通信不会出错（两台机器的网线插到同一个交换机上，这时候以太网的 CRC 是有用的）。但是，如果两台机器之间经过了多级路由器呢？

[8] http://blog.csdn.net/solstice/article/category/779646



上图中 Client 向 Server 发了一个 TCP segment，这个 segment 先被封装成一个 IP packet，再被封装成 ethernet frame，发送到路由器（图中消息 a）。Router 收到 ethernet frame (b)，转发到另一个网段(c)，最后 Server 收到 d，通知应用程序。Ethernet CRC 能保证 a 和 b 相同，c 和 d 相同；TCP header check sum 的强度不足以保证收发 payload 的内容一样。另外，如果把 Router 换成 NAT，那么 NAT 自己会构造 c（替换掉源地址），这时候 a 和 d 的 payload 不能用 tcp header checksum 校验。

路由器可能出现硬件故障，比方说它的内存故障（或偶然错误）导致收发 IP 报文出现多 bit 的反转或双字节交换，这个反转如果发生在 payload 区，那么无法用链路层、网络层、传输层的 check sum 查出来，只能通过应用层的 check sum 来检测。这个现象在开发的时候不会遇到，因为开发用的几台机器很可能都连到同一个交换机，ethernet CRC 能防止错误。开发和测试的时候数据量不大，错误很难发生。之后大规模部署到生产环境，网络环境复杂，这时候出个错就让人措手不及。有一篇论文《When the CRC and TCP checksum disagree》分析了这个问题。另外《The Limitations of the Ethernet CRC and TCP/IP checksums for error detection》[9]也值得一读。

这个情况真的会发生吗？会的，Amazon S3 在 2008 年 7 月就遇到过，单 bit 反转导致了一次严重线上事故，所以他们吸取教训加了 check sum。见 http://status.aws.amazon.com/s3-20080720.html

另外一个例证：下载大文件的时候一般都会附上 MD5，这除了有安全方面的考虑（防止篡改），也说明应用层应该自己设法校验数据的正确性。这是 end-to-end principle 的一个例证。

三本必看的书

谈到 Unix 编程和网络编程，W. Richard Stevens 是个绕不开的人物，他生前写了 6 本书，APUE、两卷 UNP、三卷 TCP/IP。有四本与网络编程直接相关。UNP 第二卷其实跟网络编程关系不大，是 APUE 在多线程和进程间通信(IPC)方面的补充。很多人把 TCP/IP 一二三卷作为整体推荐，其实这三本书用处不同，应该区别对待。

这里谈到的几本书都没有超出孟岩在《TCP/IP 网络编程之四书五经》中的推荐，说明网络编程这一领域已经相对成熟稳定。

- 《*TCP/IP Illustrated, Vol. 1: The Protocols*》中文名《TCP/IP 详解》，以下简称 TCPv1。

  TCPv1 是一本奇书。

[9] http://noahdavids.org/self_published/CRC_and_checksum.html

这本书迄今至少被三百多篇学术论文引用过 http://portal.acm.org/citation.cfm?id=161724。一本学术专著被论文引用算不上出奇，难得的是一本写给程序员看的技术书能被学术论文引用几百次，我不知道还有哪本技术书能做到这一点。

TCPv1 堪称 TCP/IP 领域的圣经。作者 W. Richard Stevens 不是 TCP/IP 协议的发明人，他从使用者（程序员）的角度，以 tcpdump 为工具，对 TCP 协议抽丝剥茧娓娓道来（第 17~24 章），让人叹服。恐怕 TCP 协议的设计者也难以讲解得如此出色，至少不会像他这么耐心细致地画几百幅收发 package 的时序图。

TCP 作为一个可靠的传输层协议，其核心有三点：

1. Positive acknowledgement with retransmission
2. Flow control using sliding window（包括 Nagle 算法等）
3. Congestion control（包括 slow start、congestion avoidance、fast retransmit 等）

第一点已经足以满足“可靠性”要求（为什么？）；第二点是为了提高吞吐量，充分利用链路层带宽；第三点是防止过载造成丢包。换言之，第二点是避免发得太慢，第三点是避免发得太快，二者相互制约。从反馈控制的角度看，TCP 像是一个自适应的节流阀，根据管道的拥堵情况自动调整阀门的流量。

TCP 的 flow control 有一个问题，每个 TCP connection 是彼此独立的，保存有自己的状态变量；一个程序如果同时开启多个连接，或者操作系统中运行多个网络程序，这些连接似乎不知道他人的存在，缺少对网卡带宽的统筹安排。（或许现代的操作系统已经解决了这个问题？）

TCPv1 唯一的不足是它出版太早了，1993 年至今网络技术发展了几代。**链路层**方面，当年主流的 10Mbit 网卡和集线器早已经被淘汰；100Mbit 以太网也没什么企业在用了，交换机(switch)也已经全面取代了集线器(hub)；服务器机房以 1Gbit 网络为主，有些场合甚至用上了 10Gbit 以太网。另外，无线网的普及也让 TCP flow control 面临新挑战；原来设计 TCP 的时候，人们认为丢包通常是拥塞造成的，这时应该放慢发送速度，减轻拥塞；而在无线网中，丢包可能是信号太弱造成的，这时反而应该快速重试，以保证性能。**网络层**方面变化不大，IPv6 雷声大雨点小。**传输层**方面，由于链路层带宽大增，TCP window scale option 被普遍使用，另外 TCP timestamps option 和 TCP selective ack option 也很常用。由于这些因素，在现在的 Linux 机器上运行 tcpdump 观察 TCP 协议，程序输出会与原书有些不同。

一个好消息：TCPv1 已于 2011 年 10 月由人续写了第二版，不知能否延续辉煌？
http://www.amazon.com/gp/product/0321336313

- 《Unix Network Programming, Vol. 1: Networking API》第二版或第三版（这两版的副标题稍有不同，第三版去掉了 XTI），以下统称 UNP，如果需要会以 UNP2e、UNP3e 细分。

UNP 是 Sockets API 的权威指南，但是网络编程远不是使用那十几个 Sockets API 那么简单，作者 W. Richard Stevens 深刻地认识到这一点，他在 UNP2e 的前言中写到：
http://www.kohala.com/start/preface.unpv12e.html

> *I have found when teaching network programming* ***that about 80% of all network programming problems have nothing to do with network programming****, per se. That is, the problems are not with the API functions such as* `accept` *and* `select`*, but* ***the problems arise from a lack of understanding of the underlying network protocols.*** *For example, I have found that once a student understands TCP's three-way handshake and four-packet connection termination, many network programming problems are immediately understood.*

搞网络编程，一定要熟悉 TCP/IP 协议及其外在表现（比如打开和关闭 Nagle 算法对收发包的影响），不然出点意料之外的情况就摸不着头脑了。我不知道为什么 UNP3e 在前言中去掉了这段至关重要的话。

另外值得一提的是，UNP 中文版翻译得相当好，译者杨继张先生是真懂网络编程的。

UNP 很详细，面面俱到，UDP、TCP、IPv4、IPv6 都讲到了。要说有什么缺点的话，就是太详细了，重点不够突出。我十分赞同孟岩说的

> “（孟岩）我主张，在具备基础之后，学习任何新东西，都要抓住主线，突出重点。对于关键理论的学习，要集中精力，速战速决。而旁枝末节和非本质性的知识内容，完全可以留给实践去零敲碎打。
>
> “原因是这样的，任何一个高级的知识内容，其中都只有一小部分是有思想创新、有重大影响的，而其它很多东西都是琐碎的、非本质的。因此，集中学习时必须把握住真正重要那部分，把其它东西留给实践。对于重点知识，只有集中学习其理论，才能确保体系性、连贯性、正确性，而对于那些旁枝末节，只有边干边学能够让你了解它们的真实价值是大是小，才能让你留下更生动的印象。如果你把精力用错了地方，比如用集中大块的时间来学习那些本来只需要查查手册就可以明白的小技巧，而对于真正重要的、思想性东西放在平时零敲碎打，那么肯定是事倍功半，甚至适得其反。
>
> “因此我对于市面上绝大部分开发类图书都不满——它们基本上都是面向知识体系本身的，而不是面向读者的。总是把相关的所有知识细节都放在一堆，然后一堆一堆攒起来变成一本书。反映在内容上，就是毫无重点地平铺直叙，不分轻重地陈述细节，往往在第三章以前就用无聊的细节谋杀了读者的热情。为什么当年侯捷先生的《深入浅出 MFC》和 Scott Meyers 的 *Effective C++* 能够成为经典？就在于这两本书抓住了各自领域中的主干，提纲挈领，纲举目张，一下子打通读者的任督二脉。可惜这样的书太少，就算是已故 Richard Stevens 和当今 Jeffrey Richter 的书，也只是在体系性和深入性上高人一头，并不是面向读者的书。”

什么是旁枝末节呢？拿以太网来说，CRC32 如何计算就是“旁枝末节”。网络程序员要明白 check sum 的作用，知道为什么需要 check sum，至于具体怎么算 CRC 就不需要程序员操心。这部分通常是由网卡硬件完成的，在发包的时候由硬件填充 CRC，在收包的时候网卡自动丢弃 CRC 不合格的包。如果代码里边确实要用到 CRC 计算，调用通用的 zlib 就行，也不用自己实现。

UNP 就像给了你一堆做菜的原料（各种 Sockets 函数的用法），常用和不常用的都给了（Out-of-Band Data、Signal-Driven IO 等等），要靠读者自己设法取舍组合，做出一盘大菜来。在第一遍读的时候，我建议只读那些基本且重要的章节；另外那些次要的内容可略作了解，即便跳过不读也无妨。UNP 是一本操作性很强的书，读这本这本书一定要上机练习。

另外，UNP 举的两个例子（菜谱）太简单，daytime 和 echo 一个是短连接协议，一个是长连接无格式协议，不足以覆盖基本的网络开发场景（比如 TCP 封包与拆包、多连接之间交换数据）。我估计 W. Richard Stevens 原打算在 UNP 第三卷中讲解一些实际的例子，只可惜他英年早逝，我等无福阅读。

UNP 是一本偏重 Unix 传统的书，这本书写作的时候服务端还不需要处理成千上万的连接，也没有现在那么多网络攻击。书中重点介绍的以 accept()+fork()来处理并发连接的方式在现在看来已经有点吃力，这本书的代码也没有特别防范恶意攻击。如果工作涉及这些方面，需要再进一步学习专门的知识（C10k 问题，安全编程）。

TCPv1 和 UNP 应该先看哪本？我不知道。我自己是先看的 TCPv1，花了大约半学期时间，然后再读 UNP2e 和 APUE。

- 《*Effective TCP/IP Programming*》

第三本书我犹豫了很久，不知道该推荐哪本，还有哪本书能与 W. Richard Stevens 的这两本比肩吗？W. Richard Stevens 为技术书籍的写作树立了难以逾越的标杆，他是一位伟大的技术作家。没能看到他写完 UNP 第三卷实在是人生的遗憾。

《*Effective TCP/IP Programming*》这本书属于专家经验总结类，初看时觉得收获很大，工作一段时间再看也能有新的发现。比如第 6 条“TCP 是一个字节流协议”，看过这一条就不会去研究所谓的“TCP 粘包问题”。我手头这本电力社 2001 年的中文版翻译尚可，但是很狗血的是把参考文献去掉了，正文中引用的文章资料根本查不到名字。人邮 2011 年重新翻译出版的版本有参考文献。

其他值得一看的书

以下两本都不易读，需要相当的基础。

- 《*TCP/IP* Illustrated, *Vol. 2: The Implementation*》以下简称 TCPv2

1200 页的大部头，详细讲解了 4.4BSD 的完整 TCP/IP 协议栈，注释了 15,000 行 C 源码。这本书啃下来不容易，如果时间不充裕，我认为没必要啃完，应用层的网络程序员选其中与工作相关的部分来阅读即可。

这本书第一作者是 Gary Wright，从叙述风格和内容组织上是典型的“面向知识体系本身”，先讲 mbuf，再从链路层一路往上、以太网、IP 网络层、ICMP、IP 多播、IGMP、IP 路由、多播路由、Sockets 系统调用、ARP 等等。到了正文内容 3/4 的地方才开始讲 TCP。面面俱到、主次不明。

对于主要使用 TCP 的程序员，我认为 TCPv2 一大半内容可以跳过不看，比如路由表、IGMP 等等（开发网络设备的人可能更关心这些内容）。在工作中大可以把 IP 视为 host-to-host 的协议，把“IP packet 如何送达对方机器”的细节视为黑盒子，这不会影响对 TCP 的理解和运用，因为网络协议是分层的。这样精简下来，需要看的只有三四百页，四五千行代码，大大减轻了负担。

这本书直接呈现高质量的工业级操作系统源码，读起来有难度，读懂它甚至要有“不求甚解的能力”。其一，代码只能看，不能上机运行，也不能改动试验。其二，与操作系统其他部分紧密关联。比如 TCP/IP stack 下接网卡驱动、软中断；上承 inode 转发来的系统调用操作；中间还要与平级的进程文件描述符管理子系统打交道；如果要把每一部分都弄清楚，把持不住就迷失主题了。其三，一些历史包袱让代码变复杂晦涩。比如 BSD 在 80 年代初需要在只有 4M 内存的 VAX 上实现 TCP/IP，内存方面捉襟见肘，这才发明了 mbuf 结构，代码也增加了不少偶发复杂度（buffer 不连续的处理）。

读这套 TCP/IP 书切忌胶柱鼓瑟，这套书以 4.4BSD 为底，其描述的行为（特别是与 timer 相关的行为）与现在的 Linux TCP/IP 有不小的出入，用书本上的知识直接套用到生产环境的 Linux 系统可能会造成不小的误解和困扰。（TCPv3 不重要，可以成套买来收藏，不读亦可。）

- 《*Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*》以下简称 POSA2

这本书总结了开发并发网络服务程序的模式，是对 UNP 很好的补充。UNP 中的代码往往把业务逻辑和 Sockets API 调用混在一起，代码固然短小精悍，但是这种编码风格恐怕不适合开发大型的网络程序。POSA2 强调模块化，网络通信交给 library/framework 去做，程序员写代码只关注业务逻辑，这是非常重要的思想。阅读这本书对于深入理解常用的 event-driven 网络库（libevent、Java Netty、Java Mina、Perl POE、Python Twisted 等等）也很有帮助，因为这些库都是依照这本书的思想编写的。

POSA2 的代码是示意性的，思想很好，细节不佳。其 C++ 代码没有充分考虑资源的自动化管理(RAII)，如果直接按照书中介绍的方式去实现网络库，那么会给使用者造成不小的负担与陷阱。换言之，照他说的做，而不是照他做的学。

不值一看的书

Douglas Comer 教授名气很大，著作等身，但是他写的网络方面的书不值一读，味同嚼蜡。网络编程与 TCP/IP 方面，有 W. Richard Stevens 的书扛鼎；计算机网络原理方面，有 Kurose 的“自顶向下”和 Peterson 的“系统”打旗，没其他人什么事儿。顺便一提，Tanenbaum 的操作系统教材是最好的之一（嗯，之二，因为他写了两本：“现代”和“设计与实现”），不过他的计算机网络和体系结构教材的地位比不上他的操作系统书的地位。体系结构方面，Patterson 和 Hennessy 二人合作的两本书是最好的，近年来崭露头角的《深入理解计算机系统》也非常好；当然，侧重点不同。

(完)

2011-06-06: 初版。

2011-06-08: 修正 TCP checksum 表述错误。

2012-02-13: 增加 ipython 小节。