附录 A

谈一谈网络编程学习经验

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§A.1.5 “网络编程的各种任务角色”。

受限于本人的经历和经验，本附录的适应范围是：

- 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”，以自问自答的形式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 的那几本。

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”。

A.1 网络编程的一些“胡思乱想”

以下大致列出我对网络编程的一些想法，前后无关联。

A.1.1 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 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，体育馆有几片场地，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 （显示器是 42 寸平板电视，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 的功能不完全一样，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 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，避免 single point of failure。

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几十个网络程序，这些示例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 §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[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；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、

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

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 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。

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

[2] 必要时甚至要修改 Linux 内核（http://linux.dell.com/files/presentations/Linux_Plumbers_Conf_2010/Scaling_techniques_for_servers_with_high_connection%20rates.pdf）。

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 8 000 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——

[3] http://blog.csdn.net/solstice/article/details/5334243，收入本书第 3 章。

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。

A.1.8 7×24 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 7×24 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 7×24 运行通常是系统设计上考虑不周。具体见本书 §9.2 “分布式系统的可靠性浅说”。重要的不是 7×24，而是在程序不必做到 7×24 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 7×24，那么也不必害怕内存碎片[5][6]，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。有没有谁能够故意制造内存碎片（不是内存泄漏）使得服务程序失去响应？
- 现在的内存分配器（`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（§12.2.8）。

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

[5] http://stackoverflow.com/questions/3770457/what-is-memory-fragmentation

[6] http://stackoverflow.com/questions/60871/how-to-solve-memory-fragmentation

- 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% ~ 5%，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？

有人为了避免内存碎片，不使用 STL 容器，也不敢 new/delete，这算是 premature optimization 还是因噎废食呢？

A.1.9 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，在一段时间之内持有（hold）一些内核资源。如果并发访问量很高，就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 (s)。服务端有一个定时器，如果客户端若干秒之内没有主动断开，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，这样可避免频繁创建、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 p. 348 举的例子）外部事件发生时，网络消息应该发 snapshot 还是 delta？新增功能时，各个组件如何平滑升级？

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。

A.1.10 网络编程的三个层次

侯捷先生在《漫谈程序员与编程》[7] 中讲到 STL 运用的三个档次：“会用 STL，是一种档次。对 STL 原理有所了解，又是一个档次。追踪过 STL 源码，又是一个档次。第三种档次的人用起 STL 来，虎虎生风之势绝非第一档次的人能够望其项背。”

我认为网络编程也可以分为三个层次：

1. 读过教程和文档，做过练习；
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》并基本理解 TCP/IP 协议，读过本系统的 manpage。在这个层次，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：

1. 有可能出现 TCP 自连接（self-connection）[8]，程序应该有所准备。
2. 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。

[7] http://jjhou.boolan.com/programmer-5-talk.htm

[8] 见 §8.11 和《学之者生，用之者死——ACE 历史与简评》举的三个硬伤（http://blog.csdn.net/solstice/article/details/5364096）。

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（见本书附录 D），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)。写完这一版协议栈，还可以继续发挥，用 FTDI 的 USB-SPI 接口芯片连接 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，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。

A.1.11 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了更高的要求：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（§7.13）。

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。

A.1.12 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[9]，方便试验各个 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

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

In [15]: client.recv(1024)      # 再去读数据，立刻返回结果
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：

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

```
$ nc localhost 5000
Hello <enter>
World <enter>
Bye! <enter>
```

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，采用非阻塞 IO，支持高并发和多线程，核心代码量不大（4000 多行），示例丰富，可供网络编程的学习者参考。

A.1.13 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 §7.5 “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 中有 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 checksum 是一种非常弱的 16-bit check sum 算法，其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（sum）”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，checksum 不变）。以太网的 CRC32 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？

图 A-1 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 A-1 中的消息 a）。router 收到 ethernet frame b，转发到另一个网段（消息 c），最后 server 收到 d，通知应用程序。以太网 CRC 能保证 a 和 b 相同，c 和 d 相同；TCP header checksum 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 自己会构造消息 c（替换掉源地址），这时候 a 和 d 的 payload 不能用 TCP header checksum 校验。

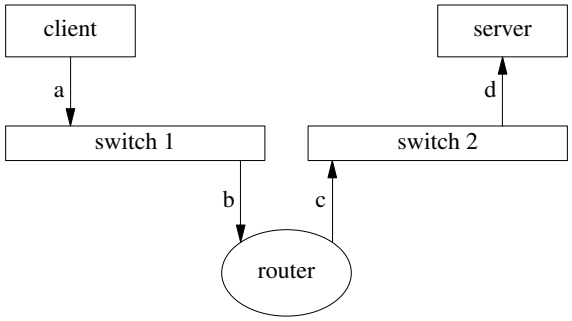

图 A-1

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》[10] 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 [11]，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。另外见 Google 工程师的经验分享 [12]。

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。

A.2 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，即 [APUE]、两卷《UNIX 网络编程》、三卷《TCP/IP 详解》。其中四本与

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

[11] http://status.aws.amazon.com/s3-20080720.html

[12] http://www.ukuug.org/events/spring2007/programme/ThatCouldntHappenToUs.pdf 第 14 页起。

网络编程直接相关。[UNPv2] 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 详解》一二三卷作为整体推荐，其实这三本书的用处不同，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。

第一本：《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。这本书迄今至少被三百多篇学术论文引用过 [13]。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，保存着自己的状态变量；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得太早了，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

[13] http://portal.acm.org/citation.cfm?id=161724

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 月推出第 2 版，经典能否重现？

第二本：《Unix Network Programming, Vol. 1: Networking API》第 2 版或第 3 版（这两版的副标题稍有不同，第 3 版去掉了 XTI），以下统称 UNP。W. Richard Stevens 在 UNP 第 2 版出版之后就不幸去世了，UNP 第 3 版是由他人续写的。

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 深刻地认识到了这一点，他在 UNP 第 2 版的前言中写道：[14]

> 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 算法对收发包延时的影响），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 UNP 第 3 版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《UNIX 网络编程》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：[15]

[14] http://www.kohala.com/start/preface.unpv12e.html

[15] http://blog.csdn.net/myan/archive/2010/09/11/5877305.aspx

> （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，而其他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的那部分，把其他东西留给实践。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，只有边干边学才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、思想性的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，一下子打通了读者的“任督二脉”。可惜这样的书太少了，就算是已故的 W. 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了。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。如果代码中确实要用到 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。在读第一遍的时候，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？见仁见智吧。我自己是先看的 TCPv1，花了大约两个月时间，然后再读 UNP 和 APUE。

第三本：《Effective TCP/IP Programming》

关于第三本书，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”。我手头这本中国电力出版社 2001 年的中文版翻译尚可，但是却把参考文献去掉了，正文中引用的文章资料根本查不到名字。人民邮电出版社 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。

这本书的第一作者是 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 的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，大大减轻了阅读的负担。

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，与操作系统的其他部分紧密关联。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，把持不住就会迷失主题。其三，一些历史包袱让代码变得复杂晦涩。比如 BSD 在 20 世纪 80 年代初需要在只有 4MiB 内存的 VAX 小型机上实现 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 为讲解对象，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。（《TCP/IP 详解（第 3 卷）》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。

附录 B
从《C++ Primer（第 4 版）》入手学习 C++

这是我为《C++ Primer（第 4 版）（评注版）》写的序言，文中“本书”指的是这本评注版（脚注 34 除外）。

B.1 为什么要学习 C++

2009 年本书作者 Stanley Lippman 先生应邀来华参加上海祝成科技举办的 C++ 技术大会，他表示人们现在还用 C++ 的唯一理由是其性能。相比之下，Java、C#、Python 等语言更加易学易用并且开发工具丰富，它们的开发效率都高于 C++。但 C++ 目前仍然是运行最快的语言[1]，如果你的应用领域确实在乎这个性能，那么 C++ 是不二之选。

这里略举几个例子[2]。对于手持设备而言，提高运行效率意味着完成相同的任务需要更少的电能，从而延长设备的操作时间，增强用户体验。对于嵌入式[3]设备而言，提高运行效率意味着：实现相同的功能可以选用较低档的处理器和较少的存储器，降低单个设备的成本；如果设备销量大到一定的规模，可以弥补 C++ 开发的成本。对于分布式系统而言，提高 10% 的性能就意味着节约 10% 的机器和能源。如果系统大到一定的规模（数千台服务器），值得用程序员的时间去换取机器的时间和数量，可以降低总体成本。另外，对于某些延迟敏感的应用（游戏[4]，金融交易），通常不能

[1] 见编程语言性能对比网站（http://shootout.alioth.debian.org/）和 Google 员工写的语言性能对比论文（https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf）。

[2] C++ 之父 Bjarne Stroustrup 维护的 C++ 用户列表：http://www2.research.att.com/~bs/applications.html。

[3] 初窥 C++ 在嵌入式系统中的应用，参见 http://aristeia.com/TalkNotes/MISRA_Day_2010.pdf。

[4] Milo Yip 在《C++ 强大背后》提到大部分游戏引擎（如 Unreal/Source）及中间件（如 Havok/FMOD）是 C++ 实现的（http://www.cnblogs.com/miloyip/archive/2010/09/17/behind_cplusplus.html）。

容忍垃圾收集（GC）带来的不确定延时，而 C++ 可以自动并精确地控制对象销毁和内存释放时机[5]。我曾经不止一次见到，出于性能（特别是及时性方面的）原因，用 C++ 重写现有的 Java 或 C# 程序。

C++ 之父 Bjarne Stroustrup 把 C++ 定位于偏重系统编程（system programming）[6] 的通用程序设计语言，开发信息基础架构（infrastructure）是 C++ 的重要用途之一[7]。Herb Sutter 总结道[8]，C++ 注重运行效率（efficiency）、灵活性（flexibility）[9] 和抽象能力（abstraction），并为此付出了生产力（productivity）方面的代价[10]。用本书作者的话来说，就是“C++ is about *efficient programming with abstractions*”（C++ 的核心价值在于能写出“运行效率不打折扣的抽象”）[11]。

要想发挥 C++ 的性能优势，程序员需要对语言本身及各种操作的代价有深入的了解[12]，特别要避免不必要的对象创建[13]。例如下面这个函数如果漏写了 `&`，功能还是正确的，但性能将会大打折扣。编译器和单元测试都无法帮我们查出此类错误，程序员自己在编码时须得小心在意。

```
inline int find_longest(const std::vector<std::string>& words)
{
  // std::max_element(words.begin(), words.end(), LengthCompare());
}
```

在现代 CPU 体系结构下，C++ 的性能优势很大程度上得益于对内存布局（memory layout）的精确控制，从而优化内存访问的局部性（locality of reference）

[5] 参见孟岩的《垃圾收集机制批判》：“C++ 利用智能指针达成的效果是，一旦某对象不再被引用，系统刻不容缓，立刻回收内存。这通常发生在关键任务完成后的清理（clean up）时期，不会影响关键任务的实时性，同时，内存里所有的对象都是有用的，绝对没有垃圾空占内存。”（http://blog.csdn.net/myan/article/details/1906）

[6] 有人半开玩笑地说：“所谓系统编程，就是那些 CPU 时间比程序员的时间更重要的工作。”

[7]《Software Development for Infrastructure》（http://www2.research.att.com/~bs/Computer-Jan12.pdf）。

[8] Herb Sutter 在 C++ and Beyond 2011 会议上的开场演讲：《Why C++?》（http://channel9.msdn.com/posts/C-and-Beyond-2011-Herb-Sutter-Why-C）。

[9] 这里的灵活性指的是编译器不阻止你干你想干的事情，比如为了追求运行效率而实现即时编译（just-in-time compilation）。

[10] 我曾向 Stanley Lippman 介绍目前我在 Linux 下的工作环境（编辑器、编译器、调试器），他表示这跟他在 1970 年代的工作环境相差无几，可见 C++ 在开发工具方面的落后。另外 C++ 的编译运行调试周期也比现代的语言长，这多少影响了工作效率。

[11] 可参考 Ulrich Drepper 在《Stop Underutilizing Your Computer》中举的 SIMD 例子（http://www.redhat.com/f/pdf/summit/udrepper_945_stop_underutilizing.pdf）。

[12]《Technical Report on C++ Performance》（http://www.open-std.org/jtc1/sc22/wg21/docs/18015.html）。

[13] 可参考 Scott Meyers 的《Effective C++ in an Embedded Environment》讲义（http://www.artima.com/shop/effective_cpp_in_an_embedded_environment）。

并充分利用内存阶层（memory hierarchy）提速[14]。可参考 Scott Meyers 的讲义《CPU Caches and Why You Care》[15]、Herb Sutter 的讲义《Machine Architecture》[16]和任何一本现代的计算机体系结构教材（《计算机体系结构：量化研究方法》、《计算机组成与设计：硬件/软件接口》、《深入理解计算机系统》等）。这一点优势在近期内不会被基于 GC 的语言赶上[17]。

C++ 的协作性不如 C、Java、Python，开源项目也比这几个语言少得多，因此在 TIOBE 语言流行榜中节节下滑。但是据我所知，很多企业内部使用 C++ 来构建自己的分布式系统基础架构，并且有替换 Java 开源实现的趋势。

B.2 学习 C++ 只需要读一本大部头

C++ 不是特性（features）最丰富的语言，却是最复杂的语言，诸多语言特性相互干扰，使其复杂度成倍增加。鉴于其学习难度和知识点之间的关联性，恐怕不能用“粗粗看看语法，就撸起袖子开干，边查 Google 边学习”[18] 这种方式来学习 C++，那样很容易掉到陷阱里或养成坏的编程习惯。如果想成为专业 C++ 开发者，全面而深入地了解这门复杂语言及其标准库，你需要一本系统而权威[19] 的书，这样的书必定会是一本八九百页的大部头[20]。

兼具系统性和权威性的 C++ 教材有两本，C++ 之父 Bjarne Stroustrup 的代表作《The C++ Programming Language》和 Stanley Lippman 的这本《C++ Primer》。侯捷先生评价道：“泰山北斗已现，又何必案牍劳形于墨瀚书海之中！这两本书都从 C++ 盘古开天以来，一路改版，斩将擎旗，追奔逐北，成就一生荣光。[21]”

从实用的角度，这两本书读一本即可，因为它们覆盖的 C++ 知识点相差无几。就我个人的阅读体验而言，Primer 更易读一些，我 10 年前深入学习 C++ 正是用的

[14] 我们知道 `std::list` 的任一位置插入是 $O(1)$ 操作，而 `std::vector` 的任一位置插入是 $O(N)$ 操作，但由于 `vector` 的元素布局更加紧凑（compact），很多时候 `vector` 的随机插入性能甚至会高于 `list`。参见 http://ecn.channel9.msdn.com/events/GoingNative12/GN12Cpp11Style.pdf，这也佐证 `vector` 是首选容器。

[15] http://aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf

[16] http://www.nwcpp.org/Downloads/2007/Machine_Architecture_-_NWCPP.pdf

[17] Bjarne Stroustrup 有一篇论文《Abstraction and the C++ machine model》对比了 C++ 和 Java 的对象内存布局（http://www2.research.att.com/ bs/abstraction-and-machine.pdf）。

[18] 语出孟岩《快速掌握一个语言最常用的 50%》（http://blog.csdn.net/myan/article/details/3144661）。

[19] “权威”的意思是说你不用担心作者讲错了，能达到这个水准的 C++ 图书作者全世界也屈指可数。

[20] 同样篇幅的 Java、C#、Python 教材可以从语言、标准库一路讲到多线程、网络编程、图形编程。

[21] 侯捷《大道之行也——C++ Primer 3/e 译序》（http://jjhou.boolan.com/cpp-primer-foreword.pdf）。

《C++ Primer（第 3 版）》。这次借评注的机会仔细阅读了《C++ Primer（第 4 版）》，感觉像在读一本完全不同的新书。第 4 版内容组织及文字表达比第 3 版进步很多[22]，第 3 版可谓“事无巨细、面面俱到”，第 4 版则重点突出、详略得当，甚至篇幅也缩短了，这多半归功于新加盟的作者 Barbara Moo。

《C++ Primer （第 4 版）》讲什么？适合谁读？

这是一本 C++ 语言的教程，不是编程教程。本书不讲八皇后问题、Huffman 编码、汉诺塔、约瑟夫环、大整数运算等经典编程例题，本书的例子和习题往往都跟 C++ 本身直接相关。本书的主要内容是精解 C++ 语法（syntax）与语意（semantics），并介绍 C++ 标准库的大部分内容（含 STL）。“这本书在全世界 C++ 教学领域的突出和重要，已经无须我再赘言[23]。”

本书适合 C++ 语言的初学者，但不适合编程初学者。换言之，这本书可以是你的第一本 C++ 书，但恐怕不能作为第一本编程书。如果你不知道什么是变量、赋值、分支、条件、循环、函数，你需要一本更加初级的书[24]，本书第 1 章可用做自测题。

如果你已经学过一门编程语言，并且打算成为专业 C++ 开发者，从《C++ Primer (第 4 版)》入手不会让你走弯路。值得特别说明的是，学习本书不需要事先具备 C 语言知识。相反，这本书教你编写真正的 C++ 程序，而不是披着 C++ 外衣的 C 程序。

《C++ Primer （第 4 版)》的定位是语言教材，不是语言规格书，它并没有面面俱到地谈到 C++ 的每一个角落，而是重点讲解 C++ 程序员日常工作中真正有用的、必须掌握的语言设施和标准库[25]。本书的作者一点也不炫耀自己的知识和技巧，虽然他们有十足的资本[26]。这本书用语非常严谨（没有那些似是而非的比喻），用词平和，讲解细致，读起来并不枯燥。特别是如果你已经有一定的编程经验，在阅读时不妨思考如何用 C++ 来更好地完成以往的编程任务。

尽管本书篇幅近 900 页，但其内容还是十分紧凑的，很多地方读一个句子就值得写一小段代码去验证。为了节省篇幅，本书经常修改前文代码中的一两行，来说明新的知识点，值得把每一行代码敲到机器中去验证。习题当然也不能轻易放过。

[22] Bjarne Stroustrup 在《Programming — Principles and Practice Using C++》的参考文献中引用了本书，并特别注明“use only the 4th edition”。

[23] 侯捷《C++ Primer 4/e 译序》。

[24] 如果没有时间精读脚注 22 中提到的那本大部头，短小精干的《Accelerated C++》亦是上佳之选。另外如果想从 C 语言入手，我推荐裘宗燕老师的《从问题到程序: 程序设计与 C 语言引论》(用最新版)。

[25] 本书把 `iostream` 的格式化输出放到附录，彻底不谈 `locale/facet`，可谓匠心独运。

[26] Stanley Lippman 曾说：Virtual base class support wanders off into the Byzantine... The material is simply too esoteric to warrant discussion...

《C++ Primer （第 4 版）》体现了现代 C++ 教学与编程理念：在现成的高质量类库上构建自己的程序，而不是什么都从头自己写。这本书在第 3 章介绍了 `string` 和 `vector` 这两个常用的 class，立刻就能写出很多有用的程序。但作者不是一次性把 `string` 的上百个成员函数一一列举，而是有选择地先讲解了最常用的那几个函数，充分体现了本书作为教材而不是手册的定位。

《C++ Primer （第 4 版）》的代码示例质量很高，不是那种随手写的玩具代码。第 10.4.2 节实现了带禁用词的单词计数，第 10.6 利用标准库容器简洁地实现了基于倒排索引思路的文本检索，第 15.9 节又用面向对象方法扩充了文本检索的功能，支持布尔查询。值得一提的是，这本书讲解继承和多态时举的例子符合 Liskov 替换原则，是正宗的面向对象。相反，某些教材以复用基类代码为目的，常以“人、学生、老师、教授”或“雇员、经理、销售、合同工”为例，这是误用了面向对象的“复用”。

《C++ Primer （第 4 版）》出版于 2005 年，遵循 2003 年的 C++ 语言标准 [27]。C++ 新标准已于 2011 年定案（称为 C++11），本书不涉及 TR1 [28] 和 C++11，这并不意味着这本书过时了 [29]。相反，这本书里沉淀的都是当前广泛使用的 C++ 编程实践，学习它可谓正当时。评注版也不会越俎代庖地介绍这些新内容，但是会指出哪些语言设施已在新标准中废弃，避免读者浪费精力。

《C++ Primer （第 4 版）》是平台中立的，并不针对特定的编译器或操作系统。目前最主流的 C++ 编译器有两个，GNU G++ 和微软 Visual C++。实际上，这两个编译器阵营基本上“模塑 [30]”了 C++ 语言的行为。理论上讲，C++ 语言的行为是由 C++ 标准规定的。但是 C++ 不像其他很多语言有“官方参考实现 [31]”，因此 C++ 的行为实际上是由语言标准、几大主流编译器、现有不计其数的 C++ 产品代码共同确定的，三者相互制约。C++ 编译器不光要尽可能符合标准，同时也要遵循目标平台的成文或不成文规范和约定，例如高效地利用硬件资源、兼容操作系统提供的 C 语言接口等等。在 C++ 标准没有明文规定的地方，C++ 编译器也不能随心所欲地自由发挥。学习 C++ 的要点之一是明白哪些行为是由标准保证的，哪些是由实现（软硬件平台和编译器）保证的 [32]，哪些是编译器自由实现，没有保证的；换言之，明白哪些程序行为是可依赖的。从学习的角度，我建议如果有条件不妨两个编译器都用，相互

[27] 基本等同于 1998 年的初版 C++ 标准，修正了编译器作者关心的一些问题，与普通程序员基本无关。

[28] TR1 是 2005 年 C++ 标准库的一次扩充，增加了智能指针、bind/function、哈希表、正则表达式等。

[29] 作者正在编写《C++ Primer （第 5 版）》，会包含 C++11 的内容。

[30] G++ 统治了 Linux，并且能用在很多 Unix 系统上；Visual C++ 统治了 Windows。其他 C++ 编译器的行为通常要向它们靠拢，例如 Intel C++ 在 Linux 上要兼容 G++，而在 Windows 上要兼容 Visual C++。

[31] 曾经是 Cfront，本书作者正是其主要开发者（http://www.softwarepreservation.org/projects/c_plus_plus）。

[32] 包括 C++ 标准有规定，但编译器拒绝遵循的（http://stackoverflow.com/questions/3931312）。

比照，避免把编译器和平台特定的行为误解为 C++ 语言规定的行为 [33]。尽管不是每个人都需要写跨平台的代码，但也大可不必自我限定在编译器的某个特定版本，毕竟编译器是会升级的。

本着“练从难处练，用从易处用”的精神，我建议在命令行下编译运行本书的示例代码，并尽量少用调试器。另外，值得了解 C++ 的编译链接模型 [34]，这样才能不被实际开发中遇到的编译错误或链接错误绊住手脚。（C++ 不像现代语言那样有完善的模块（module）和包（package）设施，它从 C 语言继承了头文件、源文件、库文件等古老的模块化机制，这套机制相对较为脆弱，需要花一定时间学习规范的做法，避免误用。）

就学习 C++ 语言本身而言，我认为有几个练习非常值得一做。这不是“重复发明轮子”，而是必要的编程练习，帮助你熟悉、掌握这门语言。一是写一个复数类或者大整数类 [35]，实现基本的加减乘运算，熟悉封装与数据抽象。二是写一个字符串类，熟悉内存管理与拷贝控制。三是写一个简化的 `vector<T>` 类模板，熟悉基本的模板编程，你的这个 `vector` 应该能放入 `int` 和 `std::string` 等元素类型。四是写一个表达式计算器，实现一个节点类的继承体系（图 B-1 右），体会面向对象编程。前三个练习是写独立的值语义的类，第四个练习是对象语义，同时要考虑类与类之间的关系。

表达式计算器能把四则运算式 $3 + 2 \times 4$ 解析为图 B-1 左图的表达式树 [36]，对根节点调用 `calculate()` 虚函数就能算出表达式的值。做完之后还可以再扩充功能，比如支持三角函数和变量。

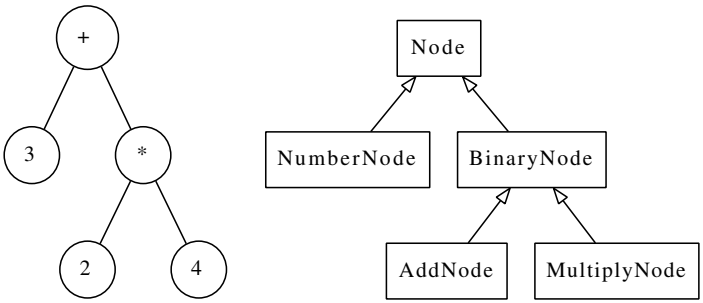

图 B-1

[33] G++ 是免费的，可使用较新的 4.x 版，最好 32-bit 和 64-bit 一起用，因为服务端已经普及 64-bit 编程。微软也有免费的 C++ 编译器，可考虑用 Visual C++ 2010 Express，建议不要用老掉牙的 Visual C++ 6.0 作为学习平台。

[34] 可参考笔者写的《C++ 工程实践经验谈》中的“C++ 编译模型精要”一节（本书第 10 章）。

[35] 大整数类可以以 `std::vector<int>` 为成员变量，避免手动资源管理。

[36] “解析”可以用数据结构课程介绍的逆波兰表达式方法，也可以用编译原理中介绍的递归下降法，还可以用专门的 Packrat 算法。程序结构可参考 http://www.relisoft.com/book/lang/poly/3tree.html。

在写完面向对象版的表达式树之后，还可以略微尝试泛型编程。比如把类的继承体系简化为图 B-2，然后用 `BinaryNode<std::plus<double> >` 和 `BinaryNode<std::multiplies<double> >` 来具现化 `BinaryNode<T>` 类模板，通过控制模板参数的类型来实现不同的运算。

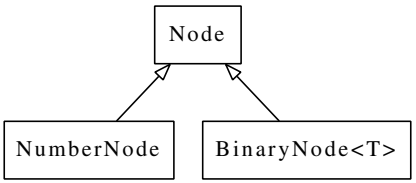

图 B-2

在表达式树这个例子中，节点对象是动态创建的，值得思考：如何才能安全地、不重不漏地释放内存。本书第 15.8 节的 `Handle` 可供参考。（C++ 的面向对象基础设施相对于现代的语言而言显得很简陋，现在 C++ 也不再以“支持面向对象”为卖点了。）

C++ 难学吗？“能够靠读书、看文章、读代码、做练习学会的东西没什么门槛，智力正常的人只要愿意花工夫，都不难达到（不错）的程度。[37]”C++ 好书很多，不过优秀的 C++ 开源代码很少，而且风格迥异[38]。我这里按个人口味和经验列几个供读者参考阅读：Google 的 Protobuf、leveldb、PCRE 的 C++ 封装，我自己写的 muduo 网络库。这些代码都不长，功能明确，阅读难度不大。如果有时间，还可以读一读 Chromium 中的基础库源码。在读 Google 开源的 C++ 代码时要连注释一起细读。我不建议一开始就读 STL 或 Boost 的源码，因为编写通用 C++ 模板库和编写 C++ 应用程序的知识体系相差很大。另外可以考虑读一些优秀的 C 或 Java 开源项目，并思考是否可以用 C++ 更好地实现或封装之（特别是资源管理方面能否避免手动清理）。

B.3 继续前进

我能够随手列出十几本 C++ 好书，但是从实用角度出发，这里只举两三本必读的书。读过《C++ Primer》和这几本书之后，想必读者已能自行识别 C++ 图书的优劣，可以根据项目需要加以钻研。

第一本是《Effective C++ 中文版（第 3 版）》[39] [EC3]。学习语法是一回事，高效

[37] 孟岩《技术路线的选择重要但不具有决定性》（http://blog.csdn.net/myan/article/details/3247071）。

[38] 从代码风格上往往能判断项目成型的时代。

[39] Scott Meyers 著，侯捷译，电子工业出版社出版。

地运用这门语言是另一回事。C++ 是一个遍布陷阱的语言，吸取专家经验尤为重要，既能快速提高眼界，又能避免重蹈覆辙。《C++ Primer》加上这本书包含的 C++ 知识足以应付日常应用程序开发。

我假定读者一定会阅读这本书，因此在评注中不引用《Effective C++ 中文版（第 3 版）》的任何章节。

《Effective C++ 中文版（第 3 版）》的内容也反映了 C++ 用法的进步。第 2 版建议“总是让基类拥有虚析构函数”，第 3 版改为“为多态基类声明虚析构函数”。因为在 C++ 中，“继承”不光只有面向对象这一种用途，即 C++ 的继承不一定是为了覆写（override）基类的虚函数。第 2 版花了很多笔墨介绍浅拷贝与深拷贝，以及对指针成员变量的处理 [40]。第 3 版则提议，对于多数 class 而言，要么直接禁用拷贝构造函数和赋值操作符，要么通过选用合适的成员变量类型 [41]，使得编译器默认生成的这两个成员函数就能正常工作。

什么是 C++ 编程中最重要的编程技法（idiom）？我认为是“用对象来管理资源”，即 RAII。资源包括动态分配的内存 [42]，也包括打开的文件、TCP 网络连接、数据库连接、互斥锁等等。借助 RAII，我们可以把资源管理和对象生命期管理等同起来，而对象生命期管理在现代 C++ 里根本不困难（见注 5），只需要花几天时间熟悉几个智能指针 [43] 的基本用法即可。学会了这三招两式，现代的 C++ 程序中可以完全不写 delete，也不必为指针或内存错误操心。现代 C++ 程序里出现资源和内存泄漏的唯一可能是循环引用，一旦发现，也很容易修正设计和代码。这方面的详细内容请参考《Effective C++ 中文版（第 3 版）》的第 3 章“资源管理”。

C++ 是目前唯一能实现自动化资源管理的语言，C 语言完全靠手工释放资源，而其他基于垃圾收集的语言只能自动清理内存，而不能自动清理其他资源 [44]（网络连接，数据库连接等）。

除了智能指针，TR1 中的 `bind/function` 也十分值得投入精力去学一学 [45]。让你从一个崭新的视角，重新审视类与类之间的关系。Stephan T. Lavavej 有一套 PPT 介

[40] Andrew Koenig 的《Teaching C++ Badly: Introduce Constructors and Destructors at the Same Time》（http://drdobbs.com/blogs/cpp/229500116）。

[41] 能自动管理资源的 `std::string`、`std::vector`、`boost::shared_ptr` 等等，这样多数 class 连析构函数都不必写。

[42] “分配内存”包括在堆（heap）上创建对象。

[43] 包括 TR1 中的 `shared_ptr`、`weak_ptr`，还有更简单的 `boost::scoped_ptr`。

[44] Java 7 有 `try-with-resources` 语句，Python 有 `with` 语句，C# 有 `using` 语句，可以自动清理栈上的资源，但对生命期大于局部作用域的资源无能为力，需要程序员手工管理。

[45] 孟岩的《function/bind 的救赎（上）》（http://blog.csdn.net/myan/article/details/5928531）。

绍 TR1 的这几个主要部件[46]。

第二本书，如果读者还是在校学生，已经学过数据结构课程[47]的话，可以考虑读一读《泛型编程与 STL》[48]；如果已经工作，学完《C++ Primer》立刻就要参加 C++ 项目开发，那么我推荐阅读《C++ 编程规范》[49] [CCS]。

泛型编程有一套自己的术语，如 concept、model、refinement 等等，理解这套术语才能阅读泛型程序库的文档。即便不掌握泛型编程作为一种程序设计方法，也要掌握 C++ 中以泛型思维设计出来的标准容器库和算法库（STL）。坊间面向对象的书琳琅满目，学习机会也很多，而泛型编程只有这么一本，读之可以开阔视野，并且加深对 STL 的理解（特别是迭代器[50]）和应用。

C++ 模板是一种强大的抽象手段，我不赞同每个人都把精力花在钻研艰深的模板语法和技巧上。从实用角度，能在应用程序中写写简单的函数模板和类模板即可（以 type traits 为限），并非每个人都要去写公用的模板库。

由于 C++ 语言过于庞大复杂，我见过的开发团队都对其剪裁使用[51]。往往团队越大，项目成立时间越早，剪裁得越厉害，也越接近 C。制订一份好的编程规范相当不容易。若规范定得太紧（比如定为团队成员知识能力的交集），程序员束手束脚，限制了生产力，对程序员个人发展也不利[52]。若规范定得太松（定为团队成员知识能力的并集），项目内代码风格迥异，学习交流协作成本上升，恐怕对生产力也不利。由两位顶级专家合写的《C++ 编程规范》一书可谓是现代 C++ 编程规范的范本。

《C++ 编程规范》同时也是专家经验一类的书，这本书篇幅比《Effective C++ 中文版（第 3 版）》短小，条款数目却多了近一倍，可谓言简意赅。有的条款看了就明白，照做即可：

- 第 1 条，以高警告级别编译代码，确保编译器无警告。
- 第 31 条，避免写出依赖于函数实参求值顺序的代码。C++ 操作符的优先级、结合性与表达式的求值顺序是无关的。裘宗燕老师写的《C/C++ 语言中表达式的求值》[53] 一文对此有明确的说明。

[46] http://blogs.msdn.com/b/vcblog/archive/2008/02/22/tr1-slide-decks.aspx

[47] 最好再学一点基础的离散数学。

[48] Matthew Austern 著，侯捷译，中国电力出版社。

[49] Herb Sutter 等著，刘基诚译，人民邮电出版社出版。（这本书的繁体版由侯捷先生和我翻译。）

[50] 侯捷先生的《芝麻开门：从 Iterator 谈起》（http://jjhou.boolan.com/programmer-3-traits.pdf）。

[51] 孟岩的《编程语言的层次观点——兼谈 C++ 的剪裁方案》（http://blog.csdn.net/myan/article/details/1920）。

[52] 一个人通常不会在一个团队工作一辈子，其他团队可能有不同的 C++ 剪裁使用方式，程序员要有“一桶水”的本事，才能应付不同形状大小的水碗。

[53] http://www.math.pku.edu.cn/teachers/qiuzy/technotes/expression2009.pdf

- 第 35 条，避免继承“并非设计作为基类使用”的 class。
- 第 43 条，明智地使用 pimpl。这是编写 C++ 动态链接库的必备手法，可以最大限度地提高二进制兼容性。
- 第 56 条，尽量提供不会失败的 swap() 函数。有了 swap() 函数，我们在自定义赋值操作符时就不必检查自赋值了。
- 第 59 条，不要在头文件中或 #include 之前写 using。
- 第 73 条，以 by value 方式抛出异常，以 by reference 方式捕捉异常。
- 第 76 条，优先考虑 vector，其次再选择适当的容器。
- 第 79 条，容器内只可存放 value 和 smart pointer。

有的条款则需要相当的设计与编码经验才能解其中三昧：

- 第 5 条，为每个物体（entity）分配一个内聚任务。
- 第 6 条，正确性、简单性、清晰性居首。
- 第 8、9 条，不要过早优化；不要过早劣化。
- 第 22 条，将依赖关系最小化。避免循环依赖。
- 第 32 条，搞清楚你写的是哪一种 class。明白 value class、base class、trait class、policy class、exception class 各有其作用，写法也不尽相同。
- 第 33 条，尽可能写小型 class，避免写出“大怪兽（monolithic class）”。
- 第 37 条，public 继承意味着可替换性。继承非为复用，乃为被复用。
- 第 57 条，将 class 类型及其非成员函数接口放入同一个 namespace。

值得一提的是，《C++ 编程规范》是出发点，但不是一份终极规范。例如 Google 的 C++ 编程规范 [54] 和 LLVM 编程规范 [55] 都明确禁用异常，这跟这本书的推荐做法正好相反。

B.4 评注版使用说明

评注版采用大 16 开印刷，在保留原书版式的前提下，对其进行了重新分页，评注的文字与正文左右分栏并列排版。另外，本书已依据原书 2010 年第 11 次印刷的版本进行了全面修订。为了节省篇幅，原书每章末尾的小结、术语表及书末的索引都没

[54] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Exceptions

[55] http://llvm.org/docs/CodingStandards.html#ci_rtti_exceptions

有印在评注版中，而是做成 PDF 供读者下载，这也方便读者检索。评注的目的是帮助初次学习 C++ 的读者快速深入掌握这门语言的核心知识，澄清一些概念、比较与其他语言的不同、补充实践中的注意事项等。评注的内容约占全书篇幅的 15%，大致比例是三分评、七分注，并有一些补白的内容 [56]。如果读者拿不定主意是否购买，可以先翻一翻第 5 章。我在评注中不谈 C++11 [57]，但会略微涉及 TR1，因为 TR1 已经投入实用。

为了不打断读者阅读的思路，评注中不会给 URL 链接，评注中偶尔会引用《C++ 编程规范》的条款，以 [CCS] 标明，这些条款的标题已在前文列出。另外评注中出现的 so*XXXXXX* 表示 http://stackoverflow.com/questions/*XXXXXX* 网址。

网上资源

代码下载：http://www.informit.com/store/product.aspx?isbn=0201721481
豆瓣页面：http://book.douban.com/subject/10944985/
术语表与索引 PDF 下载：http://chenshuo.com/cp4/ （本序的电子版也发布于此，方便读者访问脚注中的网站）。

我的联系方式：giantchen@gmail.com　http://weibo.com/giantchen

陈硕
2012 年 5 月
中国 · 香港

[56] 第 10 章绘制了数据结构示意图，第 11 章补充 lower_bound 和 upper_bound 的示例。
[57] 从 Scott Meyers 的讲义可以快速学习 C++11 （http://www.artima.com/shop/overview_of_the_new_cpp）。

附录 C
关于 Boost 的看法

这是我为电子工业出版社出版的《Boost 程序库完全开发指南》写的推荐序，此处节选了我对在 C++ 工程项目中使用 Boost 的看法。

最近一年[1]我电话面试了数十位 C++ 应聘者。惯用的暖场问题是“工作中使用过 STL 的哪些组件？使用过 Boost 的哪些组件?”。得到的答案大多集中在 vector、map、shared_ptr。如果对方是在校学生，我一般会问问 vector 或 map 的内部实现、各种操作的复杂度以及迭代器失效的可能场景。如果是有经验的程序员，我还会追问 shared_ptr 的线程安全性、循环引用的后果及如何避免、weak_ptr 的作用等。如果这些都回答得不错，进一步还可以问问如何实现线程安全的引用计数，如何定制删除动作等等。这些问题让我能迅速辨别对方的 C++ 水平。

我之所以在面试时问到 Boost，是因为其中的某些组件确实可以用于编写可维护的产品代码。Boost 包含近百个程序库，其中不乏具有工程实用价值的佳品。每个人的口味与技术背景不一样，对 Boost 的取舍也不一样。就我的个人经验而言，首先可以使用绝对无害的库，例如 noncopyable、scoped_ptr、static_assert 等，这些库的学习和使用都比较简单，容易入手。其次，有些功能自己实现起来并不困难，正好 Boost 里提供了现成的代码，那就不妨一用，比如 date_time[2] 和 circular_buffer 等等。然后，在新项目中，对于消息传递和资源管理可以考虑采用更加现代的方式，例如用 function/bind 在某些情况下代替虚函数作为库的回调接口、借助 shared_ptr 实现线程安全的对象回调等等。这二者会影响整个程序的设计思路与风格，需要通盘考虑，如果正确使用智能指针，在现代 C++ 程序里一般不需要出现 delete 语句。最后，对某些性能不佳的库保持警惕，比如 lexical_cast。总之，在项目组成员人人都能理解并运用的基础上，适当引入现成的 Boost 组件，以减少重复劳动，提高生产力。

Boost 是一个宝库，其中既有可以直接拿来用的代码，也有值得借鉴的设计思路。

[1] 这篇文章写于 2010 年 8 月。

[2] 注意 boost::date_time 处理时区和夏令时采用的方法不够灵活，可以考虑使用 muduo::TimeZone。

试举一例：正则表达式库 regex 对线程安全的处理。早期的 RegEx class 不是线程安全的，它把“正则表达式”和“匹配动作”放到了一个 class 里边。由于有可变数据，RegEx 的对象不能跨线程使用。如今的 regex 明确地区分了不可变（immutable）与可变（mutable）的数据，前者可以安全地跨线程共享，后者则不行。比如正则表达式本身（basic_regex）与一次匹配的结果（match_results）是不可变的；而匹配动作本身（match_regex）涉及状态更新，是可变的，于是用可重入的函数将其封装起来，不让这些数据泄露给别的线程。正是由于做了这样合理的区分，regex 在正常使用时就不必加锁。

Donald Knuth 在《Coders at Work》一书里表达了这样一个观点：如果程序员的工作就是摆弄参数去调用现成的库，而不知道这些库是如何实现的，那么这份职业就没啥乐趣可言。换句话说，固然我们强调工作中不要重新发明轮子，但是作为一个合格的程序员，应该具备自制轮子的能力。非不能也，是不为也。

C/C++ 语言的一大特点是其标准库可以用语言自身实现。C 标准库的 strlen、strcpy、strcmp 系列函数是教学与练习的好题材，C++ 标准库的 complex、string、vector 则是 class、资源管理、模板编程的绝佳示范。在深入了解 STL 的实现之后，运用 STL 自然手到擒来，并能自动避免一些错误和低效的用法。

对于 Boost 也是如此，为了消除使用时的疑虑，为了用得更顺手，有时我们需要适当了解其内部实现，甚至编写简化版用作对比验证。但是由于 Boost 代码用到了日常应用程序开发中不常见的高级语法和技巧，并且为了跨多个平台和编译器而大量使用了预处理宏，阅读 Boost 源码并不轻松惬意，需要下一番工夫。另一方面，如果沉迷于这些有趣的底层细节而忘了原本要解决什么问题，恐怕就舍本逐末了。

Boost 中的很多库是按泛型编程（generic programming）的范式来设计的，对于熟悉面向对象编程的人而言，或许面临一个思路的转变。比如，你得熟悉泛型编程的那套术语，如 concept、model、refinement，才容易读懂 Boost.Threads 的文档中关于各种锁的描述。我想，对于熟悉 STL 设计理念的人而言，这不是什么大问题。

在某些领域，Boost 不是唯一的选择，也不一定是最好的选择。比如，要生成公式化的源代码，我宁愿用脚本语言写一小段代码生成程序，而不用 Boost.Preprocessor；要在 C++ 程序中嵌入领域特定语言，我宁愿用 Lua 或其他语言解释器，而不用 Boost.Proto；要用 C++ 程序解析上下文无关文法，我宁愿用 ANTLR 来定义词法与语法规则并生成解析器（parser），而不用 Boost.Spirit。总之，使用 Boost 时心态要平和，别较劲去改造 C++ 语言。把它有助于提高生产力的那部分功能充分发挥出来，让项目从中受益才是关键。

（后略）

附录 D
关于 TCP 并发连接的几个思考题与试验

前几天我在新浪微博上出了两道有关 TCP 的思考题，引发了一场讨论[1]。

第一道初级题目是：有一台机器，它有一个 IP，上面运行了一个 TCP 服务程序，程序只侦听一个端口，问：从理论上讲（只考虑 TCP/IP 这一层面，不考虑 IPv6）这个服务程序可以支持多少并发 TCP 连接？（答 65536 上下的直接出局。）

具体来说，这个问题等价于：有一个 TCP 服务程序的地址是 1.2.3.4:8765，问它从理论上能接受多少个并发连接？

第二道进阶题目是：一台被测机器 A，功能同上，同一交换机上还接有一台机器 B，如果允许 B 的程序直接收发以太网 frame，问：让 A 承担 10 万个并发 TCP 连接需要用多少 B 的资源？100 万个呢？

从讨论的结果看，很多人做出了第一道题，而第二道题则几乎无人问津。这里先不公布答案（第一题答案见文末），让我们继续思考一个本质的问题：一个 TCP 连接要占用多少系统资源？

在现在的 Linux 操作系统上，如果用 `socket(2)` 或 `accept(2)` 来创建 TCP 连接，那么每个连接至少要占用一个文件描述符（file descriptor）。为什么说“至少”？因为文件描述符可以复制，比如 `dup()`；也可以被继承，比如 `fork()`；这样可能出现系统中同一个 TCP 连接有多个文件描述符与之对应。据此，很多人给出的第一题答案是：并发连接数受限于系统能同时打开的文件数目的最大值。这个答案在实践中是正确的，却不符合原题意。

如果抛开操作系统层面，只考虑 TCP/IP 层面，建立一个 TCP 连接有哪些开销？理论上最小的开销是多少？考虑两个场景：

1. 假设有一个 TCP 服务程序，向这个程序成功发起连接需要做哪些事情？换句话说，如何才能让这个 TCP 服务程序认为有客户连接到了它（让它的 `accept(2)` 调用正常返回）？

[1] http://weibo.com/1701018393/eCuxDrta0Nn

2. 假设有一个 TCP 客户端程序，让这个程序成功建立到服务器的连接需要做哪些事情？换句话说，如何才能让这个 TCP 客户端程序认为它自己已经连接到服务器了（让它的 `connect(2)` 调用正常返回）？

以上这两个问题问的不是如何编程，如何调用 Sockets API，而是问如何让操作系统的 TCP/IP 协议栈认为任务已经成功完成，连接已经成功建立。

学过 TCP/IP 协议，理解三路握手的读者想必明白，TCP 连接是虚拟的连接，不是电路连接。维持 TCP 连接理论上不占用网络资源（会占用两头程序的系统资源）。只要连接的双方认为 TCP 连接存在，并且可以互相发送 IP packet，那么 TCP 连接就一直存在。

对于问题 1，向一个 TCP 服务程序发起一个连接，客户端（为明白起见，以下称为 faketcp 客户端）只需要做三件事情（三路握手）：

1a. 向 TCP 服务程序发一个 IP packet，包含 SYN 的 TCP segment；
1b. 等待对方返回一个包含 SYN 和 ACK 的 TCP segment；
1c. 向对方发送一个包含 ACK 的 segment。

faketcp 客户端在做完这三件事情之后，TCP 服务器程序会认为连接已建立。而做这三件事情并不占用客户端的资源（为什么?），如果 faketcp 客户端程序可以绕开操作系统的 TCP/IP 协议栈，自己直接发送并接收 IP packet 或 Ethernet frame 的话。换句话说，faketcp 客户端可以一直重复做这三件事件，每次用一个不同的 IP:PORT，在服务端创建不计其数的 TCP 连接，而 faketcp 客户端自己毫发无损。我们很快将看到如何用程序来实现这一点。

对于问题 2，为了让一个 TCP 客户端程序认为连接已建立，faketcp 服务端也只需要做三件事情：

2a. 等待客户端发来的 SYN TCP segment；
2b. 发送一个包含 SYN 和 ACK 的 TCP segment；
2c. 忽视对方发来的包含 ACK 的 segment。

faketcp 服务端在做完头两件事情（收一个 SYN、发一个 SYN+ACK）之后，TCP 客户端程序会认为连接已建立。而做这三件事情并不占用 faketcp 服务端的资源（为什么?）。换句话说，faketcp 服务端可以一直重复做这三件事，接受不计其数的 TCP 连接，而 faketcp 服务端自己毫发无损。我们很快将看到如何用程序来实现这一点。

基于对以上两个问题的分析，说明单独谈论“TCP 并发连接数”是没有意义的，因为连接数基本上是要多少有多少。更有意义的性能指标或许是：“每秒收发多少条消息”、“每秒收发多少字节的数据”、“支持多少个活动的并发客户”等等。

faketcp 的程序实现

为了验证我上面的说法，我写了几个小程序来实现 faketcp，这几个程序可以发起或接受不计其数的 TCP 并发连接，并且不消耗操作系统资源，连动态内存分配都不会用到。代码见 recipes/faketcp，可以直接用 make 编译。

我家里有一台运行 Ubuntu Linux 10.04 的 PC，hostname 是 atom，所有的试验都在这上面进行。家里试验环境的网络配置如图 D-1 所示。

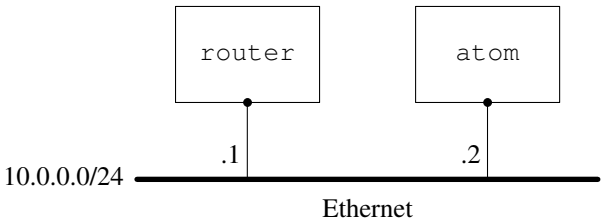

图 D-1

我在附录 A 中曾提到“可以用 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”，这次的试验正好可以用上这个办法。试验的网络配置如图 D-2 所示。

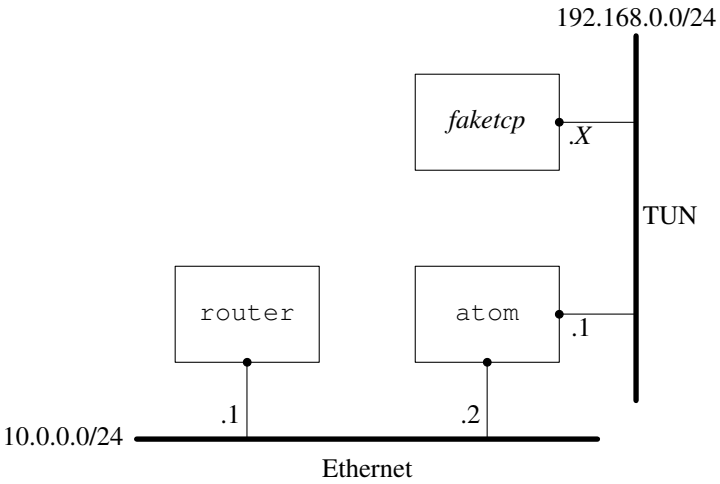

图 D-2

具体做法是：在 atom 上通过打开 /dev/net/tun 设备来创建一个 tun0 虚拟网卡，然后把这个网卡的地址设为 192.168.0.1/24，这样 faketcp 程序就扮演了 192.168.0.0/24 这个网段上的所有机器。atom 发给 192.168.0.2~192.168.0.254 的 IP packet 都会发给 faketcp 程序，faketcp 程序可以模拟其中任何一个 IP 给 atom 发 IP packet。

程序分成几步来实现。

第一步：实现 ICMP echo 协议，这样就能 ping 通 faketcp 了。代码见 recipes/faketcp/icmpecho.cc。

其中响应 ICMP echo request 的函数是 `icmp_input()`，位于 recipes/faketcp/faketcp.cc。这个函数在后面的程序中也会用到。

运行方法，打开 3 个命令行窗口：

1. 在第 1 个窗口运行 `sudo ./icmpecho`，程序显示

```
allocted tunnel interface tun0
```

2. 在第 2 个窗口运行

```
$ sudo ifconfig tun0 192.168.0.1/24
$ sudo tcpdump -i tun0
```

3. 在第 3 个窗口运行

```
$ ping 192.168.0.2
$ ping 192.168.0.3
$ ping 192.168.0.234
```

注意到每个 192.168.0.X 的 IP 都能 ping 通。

第二步：实现拒绝 TCP 连接的功能，即在收到 SYN TCP segment 的时候发送 RST segment。代码见 recipes/faketcp/rejectall.cc。

运行方法，打开 3 个命令行窗口，头两个窗口的操作与前面相同，运行的 faketcp 程序是 `./rejectall`。在第 3 个窗口运行

```
$ nc 192.168.0.2 2000
$ nc 192.168.0.2 3333
$ nc 192.168.0.7 5555
```

注意到向其中任意一个 IP 发起的 TCP 连接都被拒接了。

第三步：实现接受 TCP 连接的功能，即在收到 SYN TCP segment 的时候发回 SYN+ACK。这个程序同时处理了连接断开的情况，即在收到 FIN segment 的时候发回 FIN+ACK。代码见 recipes/faketcp/acceptall.cc。

运行方法，打开 3 个命令行窗口，步骤与前面相同，运行的 faketcp 程序是 `./acceptall`。这次会发现 nc 能和 192.168.0.X 中的每一个 IP 每一个 port 都能连通。还可以在第 4 个窗口中运行 `netstat -tpn`，以确认连接确实建立起来了。如果在 nc 中输入数据，数据会堆积在操作系统中，表现为 `netstat` 显示的发送队列（Send-Q）的长度增加。

第四步：在第三步接受 TCP 连接的基础上，实现接收数据，即在收到包含 payload 数据的 TCP segment 时发回 ACK。代码见 recipes/faketcp/discardall.cc。

运行方法，打开 3 个命令行窗口，步骤与前面相同，运行的 faketcp 程序是 `./discardall`。这次会发现 nc 能和 192.168.0.X 中的每一个 IP 每一个 port 都能连通，数据也能发出去。还可以在第 4 个窗口中运行 `netstat -tpn`，以确认连接确实建立起来了，并且发送队列的长度为 0。

这一步已经解决了前面的问题 2，扮演任意 TCP 服务端。

第五步：解决前面的问题 1，扮演客户端向 `atom` 发起任意多的连接。代码见 recipes/faketcp/connectmany.cc。

这一步的运行方法与前面不同，打开 4 个命令行窗口：

1. 在第 1 个窗口运行 `sudo ./connectmany 192.168.0.1 2007 1000`，表示将向 192.168.0.1:2007 发起 1000 个并发连接。程序显示
   ```
   allocted tunnel interface tun0
   press enter key to start connecting 192.168.0.1:2007
   ```
2. 在第 2 个窗口运行
   ```
   $ sudo ifconfig tun0 192.168.0.1/24
   $ sudo tcpdump -i tun0
   ```
3. 在第 3 个窗口运行一个能接收并发 TCP 连接的服务程序，可以是 httpd，也可以是 muduo 的 echo 或 discard 示例，程序应 listen 2007 端口。
4. 在第 1 个窗口中按回车键，再在第 4 个窗口中用 `netstat -tpn` 命令来观察并发连接。

有兴趣的话，还可以继续扩展，做更多的有关 TCP 的试验，以进一步加深理解，验证操作系统的 TCP/IP 协议栈面对不同输入的行为。甚至可以按我在附录 A 中提议的那样，实现完整的 TCP 状态机，做出一个简单的 mini tcp stack。

第一道题的答案：

在只考虑 IPv4 的情况下，并发数的理论上限是 2^{48}。考虑某些 IP 段被保留了，这个上界可适当缩小，但数量级不变。实际的限制是操作系统全局文件描述符的数量，以及内存大小。

一个 TCP 连接有两个 end points，每个 end point 是 {ip, port}，题目说其中一个 end point 已经固定，那么留下一个 end point 的自由度，即 2^{48}。客户端 IP 的上限是 2^{32} 个，每个客户端 IP 发起连接的上限是 2^{16}，乘到一起得到理论上限。

即便客户端使用 NAT，也不影响这个理论上限。(为什么?)

在真实的 Linux 系统中，可以通过调整内核参数来支持上百万并发连接，具体做法见：

- http://urbanairship.com/blog/2010/09/29/linux-kernel-tuning-for-c500k/
- http://www.metabrew.com/article/a-million-user-comet-application-with-mochiweb-part-3
- http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf

参考文献

[JCP] Brian Goetz. Java Concurrency in Practice. Addison-Wesley，2006

[RWC] Bryan Cantrill and Jeff Bonwick. Real-World Concurrency. ACM Queue，2008，9. http://queue.acm.org/detail.cfm?id=1454462

[APUE] W. Richard Stevens and Stephen A. Rago. Advanced Programming in the UNIX Environment，2nd ed. Addison-Wesley，2005 （影印版：UNIX 环境高级编程（第 2 版）. 北京：人民邮电出版社，2006）

[UNP] W. Richard Stevens. UNIX 网络编程——第 1 卷：套接口 API（第 3 版）. 杨继张译. 北京：清华大学出版社，2006 （原书名 Unix Network Programming，vol. 1，The Sockets Networking API，3rd ed；影印版：UNIX 网络编程卷 1. 北京：机械工业出版社，2004）

[UNPv2] W. Richard Stevens. Unix Network Programming，vol. 2，Interprocess Communications，2nd ed. Prentice Hall，1999 （影印版：UNIX 网络编程卷 2：进程间通信（第 2 版）. 北京：清华大学出版社，2002）

[TCPv1] W. Richard Stevens. TCP/IP Illustrated，vol. 1: The Protocols. Addison-Wesley，1994 （影印版：TCP/IP 详解卷 1：协议. 北京：人民邮电出版社，2010）

[TCPv2] W. Richard Stevens. TCP/IP Illustrated，vol. 2: The Implementation. Addison-Wesley，1995 （影印版：TCP/IP 详解卷 2：实现. 北京：人民邮电出版社，2010）

[CC2e] Steve McConnell. 代码大全（第 2 版）. 金戈，汤凌，陈硕等译. 北京：电子工业出版社，2006 （原书名 Code Complete，2nd ed）

[EC3] Scott Meyers. Effective C++ 中文版（第 3 版）. 侯捷译. 北京：电子工业出版社，2006

[ESTL] Scott Meyers. Effective STL. Addison-Wesley，2001

[CCS] Herb Sutter and Andrei Alexandrescu. C++ 编程规范. 侯捷，陈硕译. 碁峰出版社，2008 （原书名 C++ Coding Standards: 101 Rules，Guidelines，and Best Practices）

[LLL] 俞甲子，石凡，潘爱民. 程序员的自我修养——链接、装载与库. 北京：电子工业出版社，2009

[WELC] Michael Feathers. 修改代码的艺术. 刘未鹏译. 北京：人民邮电出版社，2007 （原书名 Working Effectively with Legacy Code）

[TPoP] Brian W. Kernihgan and Rob Pike. 程序设计实践. 裘宗燕译. 北京：机械工业出版社，2000 （原书名 The Practice of Programming）

[K&R] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language，2nd ed. Prentice Hall，1988 （影印版：C 程序设计语言（第 2 版）. 北京：清华大学出版社，2000）

[ExpC] Peter van der Linden. Expert C Programming: Deep C Secrets. Prentice Hall，1994

[CS:APP] Randal E. Bryant and David R. O'Hallaron. 深入理解计算机系统（第 2 版）. 龚奕利，雷迎春译. 北京：机械工业出版社，2011（原书名 Computer Systems: A Programmer's Perspective）

[D&E] Bjarne Stroustrup. C++ 语言的设计和演化. 裘宗燕译. 北京：机械工业出版社，2002 （原书名 The Design and Evolution of C++）

[ERL] Joe Armstrong. Erlang 程序设计. 赵东炜，金尹译. 北京：人民邮电出版社，2008 （原书名 Programming Erlang）

[DCC] Luiz A. Barroso and Urs Hölzle. The Datacenter as a Computer. Morgan and Claypool Publishers，2009
http://www.morganclaypool.com/doi/abs/10.2200/S00193ED1V01Y200905CAC006

[Gr00] Jeff Grossman. A Technique for Safe Deletion with Object Locking. More C++ Gems. Robert C. Martin (ed.). Cambridge University Press，2000

[jjhou02] 侯捷. 池内春秋：Memory Pool 的设计哲学和无痛运用. 程序员，2002，9.
http://jjhou.boolan.com/programmer-13-memory-pool.pdf

[Alex10] Andrei Alexandrescu. Scalable Use of the STL. C++ and Beyond 2010.
http://www.artima.com/shop/cpp_and_beyond_2010