Muduo 网络库使用手册

陈硕 (giantchen@gmail.com)

最后更新 2012-6-26

版权声明

内容一览

1

Muduo 网络库简介

1.1 由来

2010 年 3 月我写了一篇《学之者生，用之者死——ACE 历史与简评》[1]，其中提到“我心目中理想的网络库”的样子：

- 线程安全，原生支持多核多线程
- 不考虑可移植性，不跨平台，只支持 Linux，不支持 Windows。
- 主要支持 x86-64，兼顾 IA32。（实际上 muduo 也可以运行在 ARM 上。）
- 不支持 UDP，只支持 TCP。
- 不支持 IPv6，只支持 IPv4。
- 不考虑广域网应用，只考虑局域网。（实际上 muduo 也可以用在广域网上。）
- 不考虑公网，只考虑内网。不为安全性做特别的增强。
- 只支持一种使用模式：non-blocking IO + one event loop per thread，不支持阻塞 IO。
- API 简单易用，只暴露具体类和标准库里的类。API 不使用 non-trivial templates，也不使用虚函数。 复杂的
- 只满足常用需求的 90%，不面面俱到，必要的时候以 app 来适应 lib。
- 只做 library，不做成 framework。
- 争取全部代码在 5000 行以内（不含测试）。（目前 muduo 网络部分的核心代码约 4400 行。）
- 在不增加复杂度的前提下可以支持 FreeBSD/Darwin，方便将来用 Mac 作为开发用机，但不为它做性能优化。也就是说 IO multiplexing 使用 `poll(2)` 和 `epoll(4)`。

[1] http://blog.csdn.net/Solstice/archive/2010/03/10/5364096.aspx

- 以上条件都满足时，可以考虑搭配 Google Protocol Buffers RPC

在想清楚这些目标之后，我开始第三次尝试编写自己的 C++ 网络库。与前两次不同，这次我一开始就想好了库的名字，叫 muduo （木铎），并在 Google code 上创建了项目：http://code.google.com/p/muduo/。Muduo 以 git 为版本管理工具，托管于 https://github.com/chenshuo/muduo。muduo 的主体内容在 2010 年 5 月底已经基本完成，8 月底发布 0.1.0 版，现在（2012 年 6 月）的最新版本是 0.7.0。

1.2 安装

源文件 tar 包的下载地址：http://code.google.com/p/muduo/downloads/list，此处以 muduo-0.7.0-beta.tar.gz 为例。

Muduo 使用了 Linux 较新的系统调用 [2]，要求 Linux 的内核版本大于 2.6.28。我自己用 Debian 6.0 Squeeze / Ubuntu 10.04 LTS 作为主要开发环境（内核版本 2.6.32），以 g++ 4.4 为主要编译器版本，在 32 位和 64 位 x86 系统都编译测试通过。Muduo 在 Fedora 13 和 CentOS 6 上也能正常编译运行，还有热心网友为 Arch Linux 编写了 AUR 文件 [3]。

如果要在较旧的 Linux 2.6 内核 [4] 上使用 muduo，可以参考 backport.diff 来修改代码。不过这些系统上没有充分测试，仅仅是编译和冒烟测试通过。另外 muduo 也可以运行在嵌入式系统中，我在 Samsung S3C2440 开发板（ARM9）上成功运行了 muduo 的多个示例。当时 Linux 内核版本为 2.6.32，代码需略作改动，请参考 armlinux.diff。

Muduo 采用 CMake [5] 为 build system，安装方法：

```
$ sudo apt-get install cmake
```

Muduo 依赖 Boost [6]，很容易安装：

```
$ sudo apt-get install libboost1.40-dev
或
$ sudo apt-get install libboost1.42-dev
```

找boost包：
sudo apt-get install aptitude
aptitude search boost

查看当前安装的boost版本:
dpkg -S /usr/include/boost/version.hpp
libboost1.65-dev:amd64: /usr/include/boost/version.hpp

卸载：
sudo apt-get install autoremove libboost1.65-dev

sudo apt-get install libboost-all-dev
安装之后，库文件所在目录如下:
/usr/lib/x86_64-linux-gnu

[2] 主要是 timerfd 和 eventfd。
[3] http://aur.archlinux.org/packages.php?ID=49251
[4] 例如 Debian 5.0 Lenny、Ubuntu 8.04、CentOS 5 等等发行版。
[5] 最好不低于 2.8 版，CentOS 6 自带的 2.6 版也能用，但是无法自动识别 Protobuf 库。
[6] 核心库只依赖 TR1，示例代码用到了其他 boost 库

sudo apt-get install libboost-test-dev boost测试框架库
g++ -lboost_unit_test_framework

muduo 有三个非必须的依赖库：curl、c-ares DNS、Google Protobuf，如果安装了这三个库，cmake 会自动多编译一些示例。

```
$ sudo apt-get install libcurl4-openssl-dev libc-ares-dev
$ sudo apt-get install protobuf-compiler libprotobuf-dev
```

编译方法很简单：

```
$ tar zxf muduo-0.7.0-beta.tar.gz
$ cd muduo/

$ ./build.sh -j2
编译 muduo 库和它自带的例子，生成的可执行文件和静态库文件
分别位于 ../build/debug/{bin,lib}

$ ./build.sh install
以上命令将 muduo 头文件和库文件安装到 ../build/debug-install/{include,lib}
```

如果要编译 release 版（以 -O2 优化），可执行

```
$ BUILD_TYPE=release ./build.sh -j2
编译 muduo 库和它自带的例子，生成的可执行文件和静态库文件
分别位于 ../build/release/{bin,lib}

$ BUILD_TYPE=release ./build.sh install
以上命令将 muduo 头文件和库文件安装到 ../build/release-install/{include,lib}
```

在 muduo 1.0 正式发布之后，`BUILD_TYPE` 的默认值会改成 release。

编译完成之后请试运行其中的例子。比如 bin/inspector_test ，然后通过浏览器访问 http://10.0.0.10:12345/ 或 http://10.0.0.10:12345/proc/status，其中 10.0.0.10 替换为你的 Linux box 的 IP。

1.2.1 在自己的程序中使用 muduo

Muduo 是静态链接[7] 的 C++ 程序库，使用 muduo 库的时候，只需要设置好头文件路径（例如../build/debug-install/include）和库文件路径（例如../build/debug-install/lib）并链接相应的静态库文件（-lmuduo_net -lmuduo_base）即可。下面这个示范项目展示了如何使用 CMake 和普通 make 编译基于 muduo 的程序 https://github.com/chenshuo/muduo-tutorial。

[7]原因是在分布式系统中正确安全地发布动态库的成本很高，见第 ?? 章。

1.3 目录结构

Muduo 的目录结构如下。

```
muduo
|-- build.sh
|-- ChangeLog
|-- CMakeLists.txt
|-- License
|-- README
|-- muduo               muduo 库的主体
|   |-- base            与网络无关的基础代码，位于 ::muduo namespace，包括线程库
|   \-- net             网络库，位于 ::muduo::net namespace
|       |-- poller      poll(2) 和 epoll(4) 两种 IO multiplexing 后端
|       |-- http        一个简单的可嵌入的 web 服务器
|       |-- inspect     基于以上 web 服务器的“窥探器”，用于报告进程的状态
|       \-- protorpc    简单实现 Google Protobuf RPC，不推荐使用
|-- examples            丰富的示例
\-- TODO
```

Muduo 的源代码文件名与 class 名相同，例如 ThreadPool class 的定义是 muduo/base/ThreadPool.h，其实现位于 muduo/base/ThreadPool.cc。

基础库

muduo/base 目录是一些基础库，都是用户可见的类，内容包括

```
muduo
\-- base
    |-- AsyncLogging.{h,cc}         异步日志 backend
    |-- Atomic.h                    原子操作与原子整数
    |-- BlockingQueue.h             无界阻塞队列（消费者生产者队列）
    |-- BoundedBlockingQueue.h      有界阻塞队列
    |-- Condition.h                 条件变量，与 Mutex 一同使用
    |-- copyable.h                  一个空基类，用于标识 (tag) 值类型
    |-- CountDownLatch.{h,cc}       “倒计时门闩”同步
    |-- Date.{h,cc}                 Julian 日期库（即公历）
    |-- Exception.{h,cc}            带 stack trace 的异常基类
    |-- Logging.{h,cc}              简单的日志，可搭配 AsyncLogging 使用
    |-- Mutex.h                     互斥器
    |-- ProcessInfo.{h,cc}          进程信息
    |-- Singleton.h                 线程安全的 singleton
    |-- StringPiece.h               从 Google 开源代码借用的字符串参数传递类型
    |-- tests                       测试代码
    |-- Thread.{h,cc}               线程对象
    |-- ThreadLocal.h               线程局部数据
    |-- ThreadLocalSingleton.h      每个线程一个 singleton
    |-- ThreadPool.{h,cc}           简单的固定大小线程池
```

```
|-- Timestamp.{h,cc}          UTC 时间戳
|-- TimeZone.{h,cc}           时区与夏令时
\-- Types.h                   基本类型的声明，包括 muduo::string
```

网络核心库

Muduo 是基于 Reactor 模式的网络库，其核心是个事件循环 EventLoop，用于响应计时器和 IO 事件。Muduo 采用基于对象（object based）而非面向对象（object oriented）的设计风格，其事件回调接口多以 boost::function + boost::bind 表达，用户在使用 muduo 的时候不需要继承其中的 class。

网络库核心位于 muduo/net 和 muduo/net/poller，一共不到 4300 行代码，以下灰底表示用户不可见的内部类。

```
muduo
\-- net
    |-- Acceptor.{h,cc}              接受器，用于服务端接受连接
    |-- Buffer.{h,cc}                缓冲区，非阻塞 IO 必备
    |-- Callbacks.h
    |-- Channel.{h,cc}               用于每个 Socket 连接的事件分发
    |-- CMakeLists.txt
    |-- Connector.{h,cc}             连接器，用于客户端发起连接
    |-- Endian.h                     网络字节序与本机字节序的转换
    |-- EventLoop.{h,cc}             事件分发器
    |-- EventLoopThread.{h,cc}       新建一个专门用于 EventLoop 的线程
    |-- EventLoopThreadPool.{h,cc}   Muduo 默认多线程 IO 模型
    |-- InetAddress.{h,cc}           IP 地址的简单封装，
    |-- Poller.{h,cc}                IO multiplexing 的基类接口
    |-- poller                       IO multiplexing 的实现
    |   |-- DefaultPoller.cc         根据环境变量 MUDUO_USE_POLL 选择后端
    |   |-- EPollPoller.{h,cc}       基于 epoll(4) 的 IO multiplexing 后端
    |   \-- PollPoller.{h,cc}        基于 poll(2) 的 IO multiplexing 后端
    |-- Socket.{h,cc}                封装 Sockets 描述符，负责关闭连接
    |-- SocketsOps.{h,cc}            封装底层的 Sockets API
    |-- TcpClient.{h,cc}             TCP 客户端
    |-- TcpConnection.{h,cc}         muduo 里最大的一个类，有 300 多行
    |-- TcpServer.{h,cc}             TCP 服务端
    |-- tests                        简单测试
    |-- Timer.{h,cc}                 以下几个文件与定时器回调相关
    |-- TimerId.h
    \-- TimerQueue.{h,cc}
```

网络附属库

网络库有一些附属模块，它们不是核心内容，在使用的时候需要链接相应的库，例如 -lmuduo_http, -lmuduo_inspect 等等。HttpServer 和 Inspector 暴露出一个 http 界面，用于监控进程的状态，类似于 Java JMX。见第 **??** 节。

附属模块位于 muduo/net/{http,inspect,protorpc} 等处。

```
muduo
\-- net
    |-- http   不打算做成通用的 http 服务器，这只是简陋而不完整 http 协议实现
    |   |-- CMakeLists.txt
    |   |-- HttpContext.h
    |   |-- HttpRequest.h
    |   |-- HttpResponse.{h,cc}
    |   |-- HttpServer.{h,cc}
    |   \-- tests/HttpServer_test.cc  示范如何在程序中嵌入 http 服务器
    |-- inspect                       基于 http 协议的窥探器，用于报告进程的状态
    |   |-- CMakeLists.txt
    |   |-- Inspector.{h,cc}
    |   |-- ProcessInspector.{h,cc}
    |   \-- tests/Inspector_test.cc   示范暴露程序状态，包括内存使用和文件描述符
    \-- protorpc                      简单实现 Google Protobuf RPC
        |-- CMakeLists.txt
        |-- google-inl.h
        |-- RpcChannel.{h,cc}
        |-- RpcCodec.{h,cc}
        |-- rpc.proto
        \-- RpcServer.{h,cc}
```

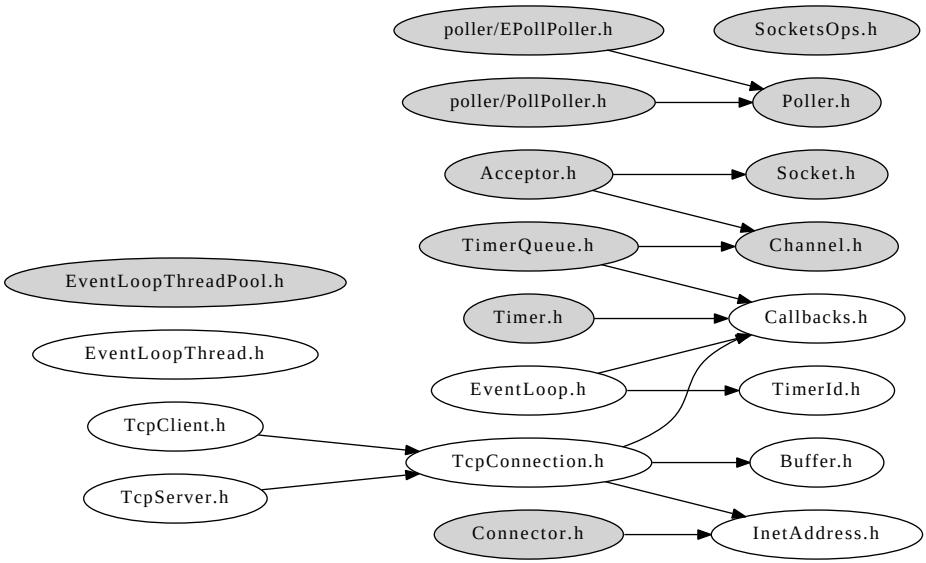
1.3.1 代码结构

Muduo 的头文件明确分为客户可见和客户不可见两类。以下是安装之后暴露的头文件和库文件。对于使用 muduo 库而言，只需要掌握 5 个关键类：Buffer、EventLoop、TcpConnection、TcpClient、TcpServer。

```
|-- include                         头文件
|   \-- muduo
|       |-- base                    基础库，同前，略
|       \-- net                     网络核心库
|           |-- Buffer.h
|           |-- Callbacks.h
|           |-- Endian.h
|           |-- EventLoop.h
|           |-- EventLoopThread.h
|           |-- InetAddress.h
|           |-- TcpClient.h
|           |-- TcpConnection.h
|           |-- TcpServer.h
|           |-- TimerId.h
|                                   以下为网络附属库的头文件
|           |-- http
|           |   |-- HttpRequest.h
|           |   |-- HttpResponse.h
|           |   \-- HttpServer.h
```

```
|           |-- inspect
|           |   |-- Inspector.h
|           |   \-- ProcessInspector.h
|           \-- protorpc
|               |-- RpcChannel.h
|               |-- RpcCodec.h
|               \-- RpcServer.h
\-- lib                                 静态库文件
    |-- libmuduo_base.a
    |-- libmuduo_net.a
    |-- libmuduo_http.a
    |-- libmuduo_inspect.a
    \-- libmuduo_protorpc.a
```

下图是 muduo 的网络核心库的头文件包含关系，用户可见的为白底，用户不可见的为灰底。



muduo 头文件中使用了前向声明 (forward declaration)，大大简化了头文件之间的依赖关系。例如 Acceptor.h、Channel.h、Connector.h、TcpConnection.h 都前向声明了 EventLoop class，从而避免包含 EventLoop.h。另外 TcpClient.h 前向声明了 Connector class，从而避免将内部类暴露给用户，类似的做法还有 TcpServer.h 用到的 Acceptor 和 EventLoopThreadPool、EventLoop.h 用到的 Poller 和 TimerQueue、TcpConnection.h 用到的 Channel 和 Socket 等等。

这里简单介绍各个 class 的作用，详细的介绍留给后文。

公开接口

- Buffer 仿 Netty ChannelBuffer 的 buffer class，数据的读写透过 buffer 进行。用户代码不需要调用 read(2)/write(2)，只需要处理收到的数据和准备好要发送的数据。见第 2.4 节。
- InetAddress 封装 IPv4 地址 (end point)，注意，muduo 目前不能解析域名[8]，只认 IP 地址。
- EventLoop 事件循环（反应器 Reactor），每个线程只能有一个 EventLoop 实体，它负责 IO 和定时器事件的分派。它用 eventfd(2) 来异步唤醒，这有别于传统的用一对 pipe(2) 的办法。它用 TimerQueue 作为计时器管理，用 Poller 作为 IO Multiplexing。
- EventLoopThread 启动一个线程，在其中运行 EventLoop::loop()
- TcpConnection 整个网络库的核心，封装一次 TCP 连接
- TcpClient 用于编写网络客户端，能发起连接，并且有重试功能
- TcpServer 用于编写网络服务器，接受客户的连接

在这些类中，TcpConnection 的生命期依靠 shared_ptr 管理（即用户和库共同控制）。Buffer 的生命期由 TcpConnection 控制。其余类的生命期由用户控制。Buffer 和 InetAddress 具有值语义，可以拷贝；其他 class 都是对象语义，不可以拷贝。

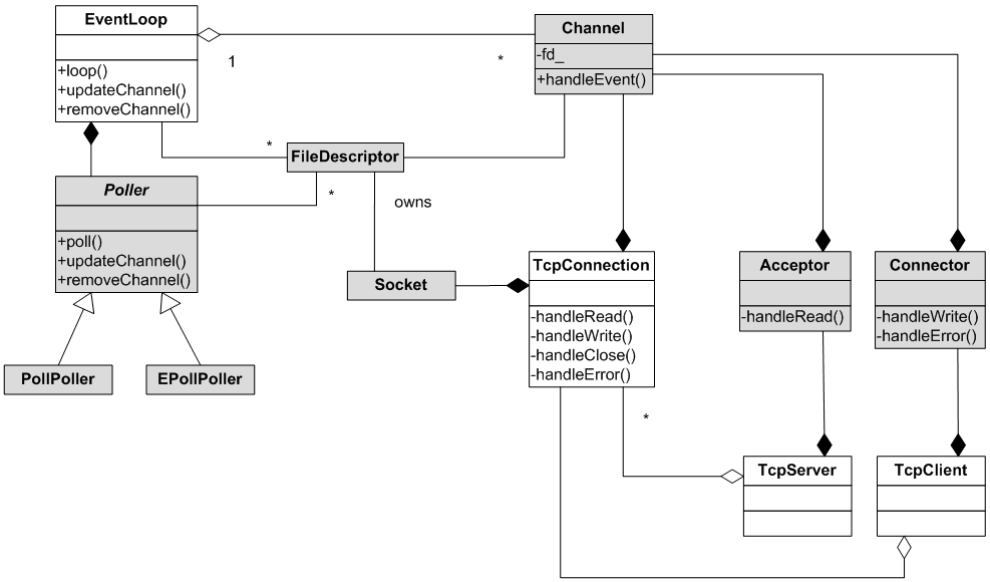
内部实现

- Channel 是 selectable IO channel，负责注册与响应 IO 事件，它不拥有 file descriptor。它是 Acceptor、Connector、EventLoop、TimerQueue、TcpConnection 的成员，生命期由后者控制。
- Socket 封装一个 file descriptor，并在析构时关闭 fd。它是 Acceptor、TcpConnection 的成员，生命期由后者控制。EventLoop、TimerQueue 也拥有 fd，但是不封装为 Socket。
- SocketsOps 封装各种 sockets 系统调用。
- Poller 是 PollPoller 和 EPollPoller 的基类，采用“电平触发”的语意。它是 EventLoop 的成员，生命期由后者控制。
- PollPoller 和 EPollPoller 封装 poll(2) 和 epoll(4) 两种 IO Multiplexing 后端。Poll 的存在价值是便于调试，因为 poll(2) 调用是上下文无关的，用 strace(1) 很容易知道库的行为是否正确。

[8]因为 gethostbyname() 域名解析是个阻塞操作，用户可以自己调用它。

- Connector 用于发起 TCP 连接，它是 TcpClient 的成员，生命期由后者控制。
- Acceptor 用于接受 TCP 连接，它是 TcpServer 的成员，生命期由后者控制。
- TimerQueue 用 timerfd 实现定时，这有别于传统的设置 poll/epoll_wait 的等待时长的办法。TimerQueue 用 std::map 来管理 Timer，常用操作的复杂度是 $O(\ln N)$，N 为定时器数目。它是 EventLoop 的成员，生命期由后者控制。
- EventLoopThreadPool 用于创建 IO 线程池，用于把 TcpConnection 分派到某个 EventLoop 线程上。它是 TcpServer 的成员，生命期由后者控制。

类图 以下是 muduo 的类图，Buffer 是 TcpConnection 的成员。



1.3.2 例子

Muduo 附带了十几个示例程序，编译出来有近百个可执行文件。这些例子位于 examples 目录，其中包括从 Boost.Asio、Java Netty、Python Twisted 等处移植过来的例子。从这些例子可以充分学习非阻塞网络编程。

```
examples
|-- asio            从 Boost.Asio 移植的例子
|   |-- chat           多人聊天的服务端和客户端，示范打包和拆包 (codec)
|   \-- tutorial       一系列 timers
|-- cdns            基于 c-ares 的异步 DNS 解析
|-- curl            基于 curl 的异步 HTTP 客户端
|-- filetransfer    简单的文件传输，示范完整发送 TCP 数据
```

```
|-- hub             一个简单的 pub/sub/hub 服务，演示应用级的广播
|-- idleconnection  踢掉空闲连接
|-- maxconnection   控制最大连接数
|-- multiplexer     1:n 串并转换服务
|-- netty           从 JBoss Netty 移植的例子
|   |-- discard        可用于测试带宽，服务器可多线程运行
|   |-- echo           可用于测试带宽，服务器可多线程运行
|   \-- uptime         带自动重连的 TCP 长连接客户端
|-- pingpong        pingpong 协议，用于测试消息吞吐量
|-- protobuf        Google Protobuf 的网络传输示例
|   |-- codec          自动反射消息类型的传输方案
|   |-- rpc            RPC 示例，实现 sudoku 服务
|   \-- rpcbench       RPC 性能测试示例
|-- roundtrip       测试两台机器的网络延时与时间差
|-- shorturl        简单的短址服务
|-- simple          五个简单网络协议的实现
|   |-- allinone       在一个程序里同时实现下面 5 个协议
|   |-- chargen        RFC 864，可测试带宽
|   |-- chargenclient  chargen 的客户端
|   |-- daytime        RFC 867
|   |-- discard        RFC 863
|   |-- echo           RFC 862
|   |-- time           RFC 868
|   \-- timeclient     time 协议的客户端
|-- socks4a         Socks4a 代理服务器，示范动态创建 TcpClient
|-- sudoku          数独求解器，示范 muduo 的多线程模型
|-- twisted         从 Python Twisted 移植的例子
|   \-- finger         finger01 ~ 07
\-- zeromq          从 ZeroMQ 移植的性能（消息延迟）测试
```

另外还有几个基于 muduo 的示例项目，由于 License 等原因没有放到 muduo 发行版中，可以单独下载。

- http://github.com/chenshuo/muduo-udns 基于 UDNS 的异步 DNS 解析
- http://github.com/chenshuo/muduo-protorpc 新的 RPC 实现，自动管理对象生命期

1.3.3 线程模型

Muduo 的线程模型符合我主张的 one loop per thread + thread pool 模型。每个线程最多有一个 EventLoop。每个 TcpConnection 必须归某个 EventLoop 管理，所有的 IO 会转移到这个线程，换句话说一个 file descriptor 只能由一个线程读写。TcpConnection 所在的线程由其所属的 EventLoop 决定，这样我们可以很方便地把不同的 TCP 连接放到不同的线程去，也可以把一些 TCP 连接放到一个线程里。TcpConnection 和 EventLoop 是线程安全的，可以跨线程调用。

TcpServer 直接支持多线程，它有两种模式：

- 单线程，accept 与 TcpConnection 用同一个线程做 IO。
- 多线程，accept 与 EventLoop 在同一个线程，另外创建一个 EventLoop-ThreadPool，新到的连接会按 round-robin 方式分配到线程池中。

后文第 1.6 节还会以 sudoku 服务器为例再次介绍 muduo 的多线程模型。

结语

Muduo 是我对常见网络编程任务的总结，用它我能很容易地编写多线程的 TCP 服务器和客户端。Muduo 是我业余时间的作品，代码估计还有一些 bug，功能也不完善（例如不支持 signal 处理 [9]），待日后慢慢改进吧。

1.4 使用教程

本文主要介绍 muduo 网络库的使用，其设计与实现将有另文讲解。

Muduo 只支持 Linux 2.6.x 下的并发非阻塞 TCP 网络编程，它的核心是每个 IO 线程一个事件循环，把 IO 事件分发到回调函数上。

我编写 Muduo 网络库的目的之一就是简化日常的 TCP 网络编程，让程序员能把精力集中在业务逻辑的实现上，而不要天天和 Sockets API 较劲。借用 Brooks 的话说，我希望 Muduo 能减少网络编程中的偶发复杂性 (accidental complexity)。

1.4.1 TCP 非阻塞网络编程本质论

基于事件的非阻塞网络编程是编写高性能并发网络服务程序的主流模式，头一次使用这种方式编程通常需要转换思维模式。把原来“主动调用 `recv(2)` 来接收数据，主动调用 `accept(2)` 来接受新连接，主动调用 `send(2)` 来发送数据”的思路换成“注册一个收数据的回调，网络库收到数据会调用我，直接把数据提供给我，供我消费。注册一个接受连接的回调，网络库接受了新连接会回调我，直接把新的连接对象传给我，供我使用。需要发送数据的时候，只管往连接中写，网络库会负责无阻塞地发送。”这种编程方式有点像 Win32 的消息循环，消息循环中的代码应该避免阻塞，否则会让整个窗口失去响应，同理，事件处理函数也应该避免阻塞，否则会让网络服务失去响应。

[9] Signal 也可以通过 signalfd(2) 融入到 EventLoop 中，见 github.com/chenshuo/muduo-protorpc 中的 zurg slave 例子

我认为，TCP 网络编程最本质的是处理三个半事件：

1. 连接的建立，包括服务端接受 (accept) 新连接和客户端成功发起 (connect) 连接。TCP 连接一旦建立，客户端和服务端是平等的，可以各自收发数据。
2. 连接的断开，包括主动断开 (close 或 shutdown) 和被动断开 (read(2) 返回 0)。
3. 消息到达，文件描述符可读。这是最为重要的一个事件，对它的处理方式决定了网络编程的风格（阻塞还是非阻塞，如何处理分包，应用层的缓冲如何设计等等）。

3.5 消息发送完毕，这算半个。对于低流量的服务，可以不必关心这个事件；另外，这里“发送完毕”是指将数据写入操作系统的缓冲区，将由 TCP 协议栈负责数据的发送与重传，不代表对方已经收到数据。

这其中有很多难点，也有很多细节需要注意，比方说：

如果要主动关闭连接，如何保证对方已经收到全部数据？如果应用层有缓冲（这在非阻塞网络编程中是必须的，见下文），那么如何保证先发送完缓冲区中的数据，然后再断开连接。直接调用 close(2) 恐怕是不行的。

如果主动发起连接，但是对方主动拒绝，如何定期 (带 back-off) 重试？

非阻塞网络编程该用边沿触发 (edge trigger) 还是电平触发 (level trigger)？[10] 如果是电平触发，那么什么时候关注 EPOLLOUT 事件？会不会造成 busy-loop？如果是边沿触发，如何防止漏读造成的饥饿？epoll(4) 一定比 poll(2) 快吗？

在非阻塞网络编程中，为什么要使用应用层发送缓冲区？假设应用程序需要发送 40k 数据，但是操作系统的 TCP 发送缓冲区只有 25k 剩余空间，那么剩下的 15k 数据怎么办？如果等待 OS 缓冲区可用，会阻塞当前线程，因为不知道对方什么时候收到并读取数据。因此网络库应该把这 15k 数据缓存起来，放到这个 TCP 链接的应用层发送缓冲区中，等 socket 变得可写的时候立刻发送数据，这样“发送”操作不会阻塞。如果在此期间应用程序又要发送 50k 数据，此时发送缓冲区中尚有 5k 数据，那么网络库应该将这 50k 数据追加到发送缓冲区的末尾，而不能立刻尝试 write，因为这样有可能打乱数据的顺序。

在非阻塞网络编程中，为什么要使用应用层接收缓冲区？假如一次读到的数据不够一个完整的数据包，那么这些已经读到的数据是不是应该先暂存在某个地方，等剩余的数据收到之后再一并处理？见 lighttpd 关于 \r\n\r\n 分包的 bug [11]。假如

[10] 这两个中文术语有其他译法，我选择了一个电子工程师熟悉的说法。

[11] http://redmine.lighttpd.net/issues/show/2105

数据是一个字节一个字节地到达，间隔 10ms，每个字节触发一次文件描述符可读 (readable) 事件，程序是否还能正常工作？lighttpd 在这个问题上出过安全漏洞 [12]。

在非阻塞网络编程中，如何设计并使用缓冲区？一方面我们希望减少系统调用，一次读的数据越多越划算，那么似乎应该准备一个大的缓冲区。另一方面，我们系统减少内存占用。如果有 10k 个连接，每个连接一建立就分配 64k 的读缓冲的话，将占用 640M 内存，而大多数时候这些缓冲区的使用率很低。muduo 用 readv 结合栈上空间巧妙地解决了这个问题。

如果使用发送缓冲区，万一接收方处理缓慢，数据会不会一直堆积在发送方，造成内存暴涨？如何做应用层的流量控制？

如何设计并实现定时器？并使之与网络 IO 共用一个线程，以避免锁。

这些问题在 muduo 的代码中可以找到答案。

1.4.2 Echo 服务的实现

Muduo 的使用非常简单，不需要从指定的类派生，也不用覆写虚函数，只需要注册几个回调函数去处理前面提到的三个半事件就行了。

以经典的 echo 回显服务为例：

1. 定义 EchoServer class，不需要派生自任何基类：

examples/simple/echo/echo.h

```
4  #include <muduo/net/TcpServer.h>
5
6  // RFC 862
7  class EchoServer
8  {
9   public:
10   EchoServer(muduo::net::EventLoop* loop,
11              const muduo::net::InetAddress& listenAddr);
12
13   void start();  // calls server_.start();
14
15  private:
16   void onConnection(const muduo::net::TcpConnectionPtr& conn);
17
18   void onMessage(const muduo::net::TcpConnectionPtr& conn,
19                  muduo::net::Buffer* buf,
20                  muduo::Timestamp time);
21
22   muduo::net::EventLoop* loop_;
```

[12] http://download.lighttpd.net/lighttpd/security/lighttpd_sa_2010_01.txt

```
23   muduo::net::TcpServer server_;
24 };
```
examples/simple/echo/echo.h

在构造函数里注册回调函数：

```
10 EchoServer::EchoServer(muduo::net::EventLoop* loop,
11                        const muduo::net::InetAddress& listenAddr)
12   : loop_(loop),
13     server_(loop, listenAddr, "EchoServer")
14 {
15   server_.setConnectionCallback(
16       boost::bind(&EchoServer::onConnection, this, _1));
17   server_.setMessageCallback(
18       boost::bind(&EchoServer::onMessage, this, _1, _2, _3));
19 }
```
examples/simple/echo/echo.cc

2. 实现 `EchoServer::onConnection()` 和 `EchoServer::onMessage()`：

```
26 void EchoServer::onConnection(const muduo::net::TcpConnectionPtr& conn)
27 {
28   LOG_INFO << "EchoServer - " << conn->peerAddress().toIpPort() << " -> "
29            << conn->localAddress().toIpPort() << " is "
30            << (conn->connected() ? "UP" : "DOWN");
31 }
32
33 void EchoServer::onMessage(const muduo::net::TcpConnectionPtr& conn,
34                            muduo::net::Buffer* buf,
35                            muduo::Timestamp time)
36 {
37   muduo::string msg(buf->retrieveAsString());
38   LOG_INFO << conn->name() << " echo " << msg.size() << " bytes, "
39            << "data received at " << time.toString();
40   conn->send(msg);
41 }
```
examples/simple/echo/echo.cc

第 37 行和第 40 行代码是 echo 服务的“业务逻辑”：把收到的数据原封不动地发回客户端。注意我们不用担心第 40 行 `send(msg)` 是否完整地发送了数据，因为 muduo 网络库会帮我们管理发送缓冲区。

这两个函数体现了“基于事件编程”的典型做法，即程序主体是被动等待事件发生，事件发生之后网络库会调用（回调）事先注册的事件处理函数 (event handler)。

在 `onConnection()` 函数中，`conn` 参数是 `TcpConnection` 对象的 `shared_ptr`，`TcpConnection::connected()` 返回一个 `bool` 值，表明目前连接是建立还是断开，

TcpConnection 的 peerAddress() 和 localAddress() 成员函数分别返回对方和本地的地址（以 InetAddress 对象表示的 IP 和 port）。

在 onMessage() 函数中，conn 参数是收到数据的那个 TCP 连接；buf 是已经收到的数据（buf的数据会累积，直到用户从中取走 (retrieve) 数据），注意 buf 是指针，表明用户代码可以修改（消费）buffer；time是收到数据的确切时间，即 epoll_wait(2)返回的时间，注意这个时间通常比 read(2) 发生的时间略早，可以用于正确测量程序的消息处理延迟。另外 Timestamp 对象采用 pass-by-value，而不是 pass-by-(const)reference，这是有意的，因为在 x86-64 上可以直接通过寄存器传参。

3. 在 main() 里用 EventLoop 让整个程序跑起来：

examples/simple/echo/main.cc

```
#include "echo.h"

#include <muduo/base/Logging.h>
#include <muduo/net/EventLoop.h>

// using namespace muduo;
// using namespace muduo::net;

int main()
{
  LOG_INFO << "pid = " << getpid();
  muduo::net::EventLoop loop;
  muduo::net::InetAddress listenAddr(2007);
  EchoServer server(&loop, listenAddr);
  server.start();
  loop.loop();
}
```

examples/simple/echo/main.cc

完整的代码见 muduo/examples/simple/echo。这个几十行的小程序实现了一个单线程并发的 echo 服务程序，可以同时处理多个连接。

这个程序用到了 TcpServer、EventLoop、TcpConnection、Buffer 这几个 class，也大致反映了这几个 class 的典型用法，后文还会详细介绍这几个 class。注意，以后的代码大多会省略 namespace。

1.4.3 七步实现 Finger 服务

Python Twisted 是一款非常好的网络库，它也采用 Reactor 作为网络编程的基本模型，所以从使用上与 muduo 颇有相似之处。（当然，muduo 没有 deferreds）

Finger 是 twisted 文档的一个经典例子，本文展示如何用 muduo 来实现最简单的 finger 服务端。限于篇幅，只实现 finger01~finger07。代码位于 examples/twisted/finger 。

1 拒绝连接 什么都不做，程序空等。

examples/twisted/finger/finger01.cc

```
#include <muduo/net/EventLoop.h>

using namespace muduo;
using namespace muduo::net;

int main()
{
  EventLoop loop;
  loop.loop();
}
```

examples/twisted/finger/finger01.cc

2 接受新连接 在 1079 端口侦听新连接，接受连接之后什么都不做，程序空等。muduo 会自动丢弃收到的数据。

examples/twisted/finger/finger02.cc

```
#include <muduo/net/EventLoop.h>
#include <muduo/net/TcpServer.h>

using namespace muduo;
using namespace muduo::net;

int main()
{
  EventLoop loop;
  TcpServer server(&loop, InetAddress(1079), "Finger");
  server.start();
  loop.loop();
}
```

examples/twisted/finger/finger02.cc

3 主动断开连接 接受新连接之后主动断开。以下省略头文件和 namespace。

examples/twisted/finger/finger03.cc

```
void onConnection(const TcpConnectionPtr& conn)
{
  if (conn->connected())
  {
    conn->shutdown();
  }
}
```

```
14
15 int main()
16 {
17   EventLoop loop;
18   TcpServer server(&loop, InetAddress(1079), "Finger");
19   server.setConnectionCallback(onConnection);
20   server.start();
21   loop.loop();
22 }
```

examples/twisted/finger/finger03.cc

4 读取用户名，然后断开连接 如果读到一行以 \r\n 结尾的消息，就断开连接。finger04.cc 注意这段代码有安全问题，如果恶意客户端不断发送数据而不换行，会撑爆服务端的内存。另外，Buffer::findCRLF() 是线性查找，如果客户端每次发一个字节，服务端的时间复杂度为 $O(N^2)$，会消耗 CPU 资源。

examples/twisted/finger/finger04.cc

```
7  void onMessage(const TcpConnectionPtr& conn,
8                 Buffer* buf,
9                 Timestamp receiveTime)
10 {
11   if (buf->findCRLF())
12   {
13     conn->shutdown();
14   }
15 }
16
17 int main()
18 {
19   EventLoop loop;
20   TcpServer server(&loop, InetAddress(1079), "Finger");
21   server.setMessageCallback(onMessage);
22   server.start();
23   loop.loop();
24 }
```

examples/twisted/finger/finger04.cc

5 读取用户名、输出错误信息、然后断开连接 如果读到一行以 \r\n 结尾的消息，就发送一条出错信息，然后断开连接。安全问题同上。

```
--- examples/twisted/finger/finger04.cc 2010-08-29 00:03:14 +0800
+++ examples/twisted/finger/finger05.cc 2010-08-29 00:06:05 +0800
@@ -7,12 +7,13 @@
 void onMessage(const TcpConnectionPtr& conn,
                Buffer* buf,
                Timestamp receiveTime)
 {
   if (buf->findCRLF())
   {
```

```
+    conn->send("No such user\r\n");
     conn->shutdown();
   }
 }
```

6 从空的 UserMap 里查找用户 从一行消息中拿到用户名（第 30 行），在 UserMap 里查找，然后返回结果。安全问题同上。

examples/twisted/finger/finger06.cc

```
9  typedef std::map<string, string> UserMap;
10 UserMap users;
11
12 string getUser(const string& user)
13 {
14   string result = "No such user";
15   UserMap::iterator it = users.find(user);
16   if (it != users.end())
17   {
18     result = it->second;
19   }
20   return result;
21 }
22
23 void onMessage(const TcpConnectionPtr& conn,
24                Buffer* buf,
25                Timestamp receiveTime)
26 {
27   const char* crlf = buf->findCRLF();
28   if (crlf)
29   {
30     string user(buf->peek(), crlf);
31     conn->send(getUser(user) + "\r\n");
32     buf->retrieveUntil(crlf + 2);
33     conn->shutdown();
34   }
35 }
36
37 int main()
38 {
39   EventLoop loop;
40   TcpServer server(&loop, InetAddress(1079), "Finger");
41   server.setMessageCallback(onMessage);
42   server.start();
43   loop.loop();
44 }
```

examples/twisted/finger/finger06.cc

7. 往 UserMap 里添加一个用户 与前面几乎完全一样，只多了第 39 行。

```
--- examples/twisted/finger/finger06.cc 2010-08-29 00:14:33 +0800
+++ examples/twisted/finger/finger07.cc 2010-08-29 00:15:22 +0800
```

```
@@ -36,6 +36,7 @@

 int main()
 {
+  users["schen"] = "Happy and well";
   EventLoop loop;
   TcpServer server(&loop, InetAddress(1079), "Finger");
   server.setMessageCallback(onMessage);
   server.start();
   loop.loop();
 }
```

以上就是全部内容，可以用 telnet 扮演客户端来测试我们的简单 finger 服务端。

Telnet 测试

在一个命令行窗口运行

```
$ ./bin/twisted_finger07
```

另一个命令行运行

```
$ telnet localhost 1079
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
muduo
No such user
Connection closed by foreign host.
```

再试一次

```
$ telnet localhost 1079
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
schen
Happy and well
Connection closed by foreign host.
```

冒烟测试过关。

1.5 性能评测

我在一开始编写 muduo 的时候并没有以高性能为首先目标。在 2010 年 8 月发布之后，有网友询问其性能与其他常见网络库相比如何，因此我才加入了一些性能对比的示例代码。我很惊奇的发现，在 muduo 擅长的领域（TCP 长连接），其性能不比任何开源网络库差。

性能对比原则：采用对方的性能测试方案，用 muduo 实现功能相同或类似的程序，然后放到相同的软硬件环境中对比。

1.5.1 muduo 与 boost asio、libevent2 吞吐量对比

我在编写 muduo 的时候并没有以高并发高吞吐为主要目标。但出乎我的意料，ping pong 测试表明，muduo 吞吐量比 boost asio 高 15% 以上；比 libevent2 高 18% 以上，个别情况达到 70%。

测试对象

- boost 1.40 中的 asio 1.4.3
- asio 1.4.5 (http://think-async.com/Asio/Download)
- libevent 2.0.6-rc (http://monkey.org/ provos/libevent-2.0.6-rc.tar.gz)
- muduo 0.1.1

测试代码

- asio 的测试代码取自 http://asio.cvs.sourceforge.net/viewvc/asio/asio/src/tests/performance/ ，未作更改。
- 我自己编写了 libevent2 的 ping pong 测试代码，地址在 http://github.com/chenshuo/recipes/tree/master/pingpong/libevent/ 。由于这个测试代码没有使用多线程，所以只对比 muduo 和 libevent2 在单线程下的性能。
- muduo 的测试代码位于 examples/pingpong/，代码如 gist[13]所示。

muduo 和 asio 的优化编译参数均为 -O2 -finline-limit=1000

```
$ BUILD_TYPE=release ./build.sh  # 编译 muduo 的优化版本
```

[13]http://gist.github.com/564985

测试环境

硬件：DELL 490 工作站，双路 Intel 四核 Xeon E5320 CPU，共 8 核，主频 1.86GHz，内存 16GB

操作系统：Ubuntu Linux Server 10.04.1 LTS x86_64

编译器：g++ 4.4.3

测试方法

依据 asio 性能测试[14] 的办法，用 ping pong 协议来测试 muduo、asio、libevent2 在单机上的吞吐量。

简单地说，ping pong 协议是客户端和服务器都实现 echo 协议。当 TCP 连接建立时，客户端向服务器发送一些数据，服务器会 echo 回这些数据，然后客户端再 echo 回服务器。这些数据就会像乒乓球一样在客户端和服务器之间来回传送，直到有一方断开连接为止。这是用来测试吞吐量的常用办法。注意数据是无格式的，双方都是收到多少数据就反射回去多少数据，并不拆包，这与后面的 ZeroMQ 延迟测试不同。

我主要做了两项测试：

- 单线程测试。客户端与服务器运行在同一台机器，均为单线程，测试并发连接数为 1/10/100/1000/10000 时的吞吐量。
- 多线程测试。并发连接数为 100 或 1000，服务器和客户端的线程数同时设为 1/2/3/4。（由于我家里只有一台 8 核机器，而且服务器和客户端运行在同一台机器上，线程数大于 4 没有意义。）

所有测试中，ping pong 消息的大小均为 16k bytes。测试用的 shell 脚本可从 gist.github.com/564985 下载。

在同一台机器测试吞吐量的原因：

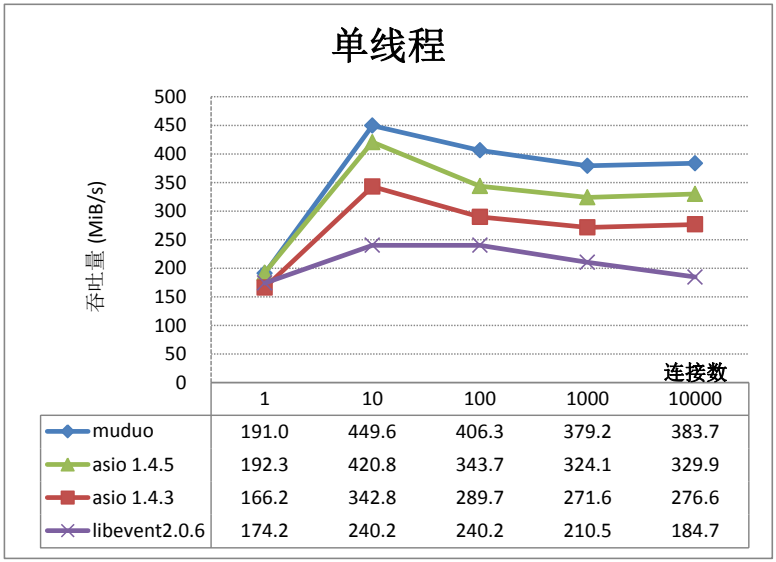
现在的 CPU 很快，即便是单线程单 TCP 连接也能把 Gigabit 以太网的带宽跑满。如果用两台机器，所有的吞吐量测试结果都将是 110 MiB/s，失去了对比的意义。（用 Python 也能跑出同样的吞吐量，或许可以对比哪个库占的 CPU 少。）

在同一台机器上测试，可以在 CPU 资源相同的情况下，单纯对比网络库的效率。也就是说单线程下，服务端和客户端各占满 1 个 CPU，比较哪个库的吞吐量高。

[14] http://think-async.com/Asio/LinuxPerformanceImprovements

测试结果

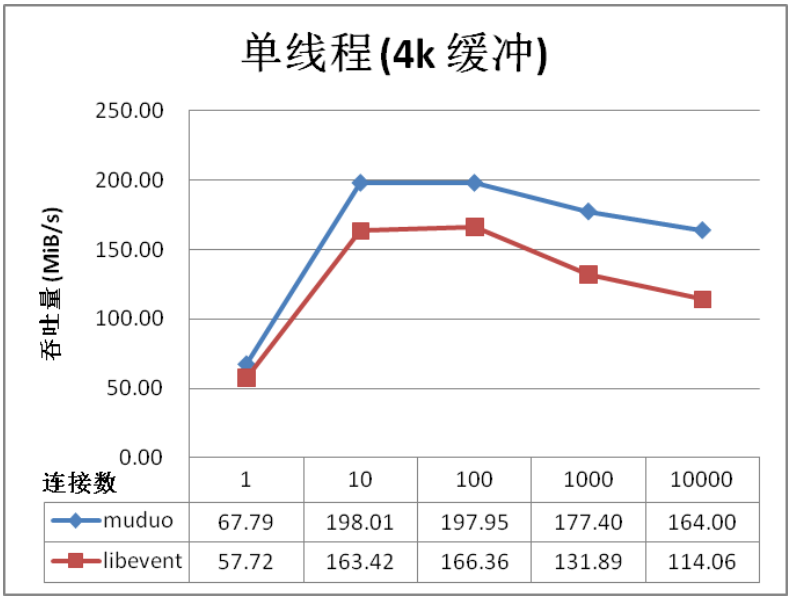
单线程测试的结果，数字越大越好：



	1	10	100	1000	10000
muduo	191.0	449.6	406.3	379.2	383.7
asio 1.4.5	192.3	420.8	343.7	324.1	329.9
asio 1.4.3	166.2	342.8	289.7	271.6	276.6
libevent2.0.6	174.2	240.2	240.2	210.5	184.7

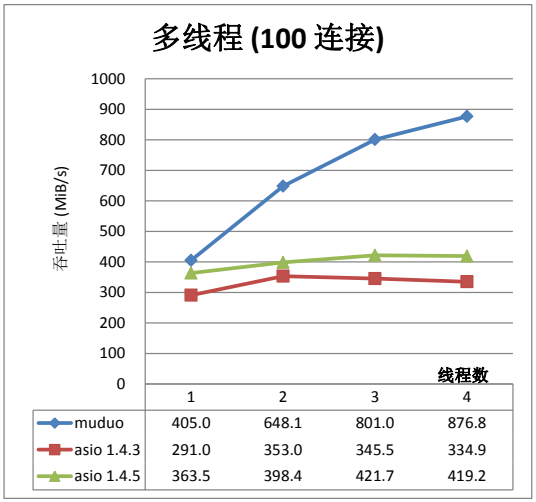
以上结果让人大跌眼镜，muduo 居然比 libevent2 快 70%！跟踪 libevent2 的源代码发现，它每次最多从 socket 读取 4096 字节的数据 (证据在 buffer.c 的 evbuffer_-read() 函数)，怪不得吞吐量比 muduo 小很多。因为在这一测试中，muduo 每次读取 16384 字节，系统调用的性价比较高。

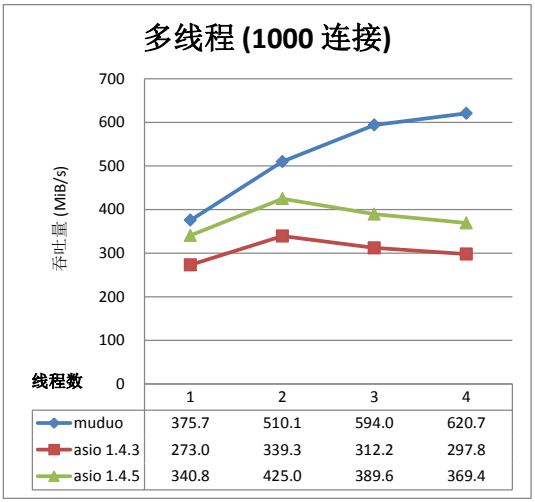
为了公平起见，我再测了一次，这回两个库都发送 4096 字节的消息。



测试结果表明 muduo 吞吐量平均比 libevent2 高 18% 以上。

多线程测试的结果，数字越大越好：





测试结果表明 muduo 吞吐量平均比 asio 高 15% 以上。

讨论

muduo 出乎意料地比 asio 性能优越，我想主要得益于其简单的设计和简洁的代码。asio 在多线程测试中表现不佳，我猜测其主要原因是测试代码只使用了一个 io_service，如果改用“io_service per CPU”的话，性能应该有所提高。我对 asio 的了解程度仅限于能读懂其代码，希望能有 asio 高手编写“io_service per CPU”的 ping pong 测试，以便与 muduo 做一个公平的比较。

由于 libevent2 每次最多从网络读取 4096 字节，大大限制了它的吞吐量。

ping pong 测试很容易实现，欢迎其他网络库（ACE、POCO、libevent 等）也能加入到对比中来，期待这些库的高手出马。

1.5.2 击鼓传花：对比 muduo 与 libevent2 的事件处理效率

前面我们比较了 muduo 和 libevent2 的吞吐量，得到的结论是 muduo 比 libevent2 快 18%。有人会说，libevent2 并不是为高吞吐的应用场景而设计的，这样的比较不公平，胜之不武。为了公平起见，这回我们用 libevent2 自带的性能测试程序（击鼓传花）来对比 muduo 和 libevent2 在高并发情况下的 IO 事件处理效率。

测试用的软硬件环境与前一小节相同，另外我还在自己的 DELL E6400 笔记本上运行了测试，结果也附在后面。

测试的场景是：有 1000 个人围成一圈，玩击鼓传花的游戏，一开始第 1 个人手里有花，他把花传给右手边的人，那个人再继续把花传给右手边的人，当花转手 100 次之后游戏停止，记录从开始到结束的时间。

用程序表达是，有 1000 个网络连接 (`socketpair(2)`s 或 `pipe(2)`s)，数据在这些连接中顺次传递，一开始往第 1 个连接里写 1 个字节，然后从这个连接的另一头读出这 1 个字节，再写入第 2 个连接，然后读出来继续写到第 3 个连接，直到一共写了 100 次之后程序停止，记录所用的时间。

以上是只有一个活动连接的场景，我们实际测试的是 100 个或 1000 个活动连接 (即 100 朵花或 1000 朵花，均匀分散在人群手中)，而连接总数（即并发数）从 100 到 100 000 (十万)。注意每个连接是两个文件描述符，为了运行测试，需要调高每个进程能打开的文件数，比如设为 256000。

libevent2 的测试代码位于 test/bench.c，我修复了 2.0.6-rc 版里的一个小 bug。修正后的代码见已经提交给 libevent2 作者，现在下载的最新版本是正确的。

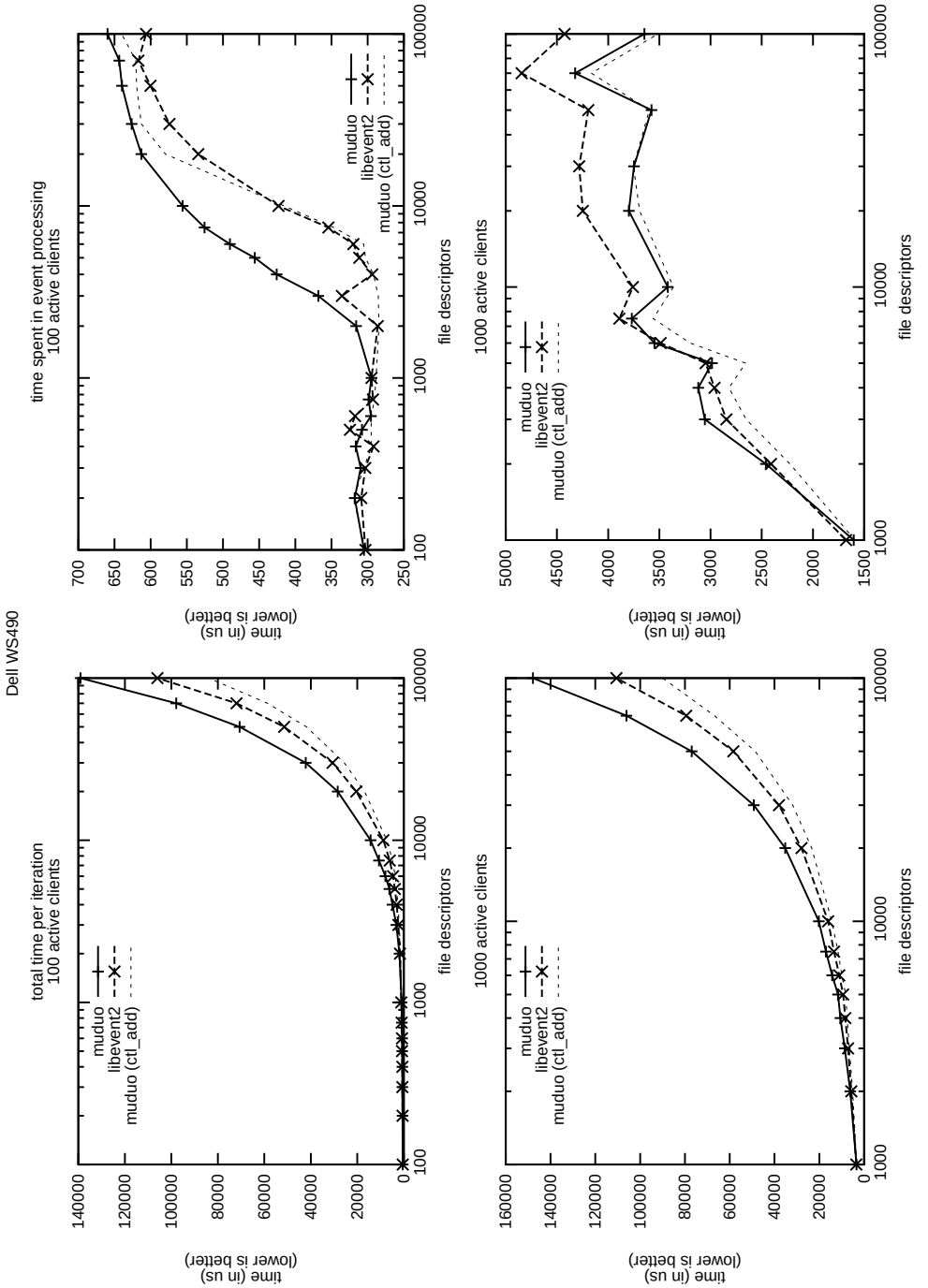
muduo 的测试代码位于 examples/pingpong/bench.cc。

测试结果与讨论

第一轮，分别用 100 个活动连接和 1000 个活动连接，无超时，读写 100 次，测试一次游戏的总时间（包含初始化）和事件处理的时间（不包含注册 event watcher）随连接数（并发数）变化的情况。具体解释见 libev 的性能测试文档 [15]，不同之处在于我们不比较 timer event 的性能，只比较 IO event 的性能。对每个并发数，程序循环 25 次，刨去第一次的热身数据，后 24 次算平均值。测试用的脚本[16] 是 libev 的作者 Marc Lehmann 写的，我略作改用，用于测试 muduo 和 libevent2。

第一轮的结果，请先只看 '+' 线和 '×' 线。'×' 线是 libevent2 用的时间，'+' 线是 muduo 用的时间。数字越小越好。注意这个图的横坐标是对数的，每一个数量级的取值点为 1, 2, 3, 4, 5, 6, 7.5, 10。

[15] http://libev.schmorp.de/bench.html
[16] http://github.com/chenshuo/recipes/blob/master/pingpong/libevent/run_bench.sh


Dell WS490
total time per iteration
100 active clients
time spent in event processing
100 active clients
1000 active clients
1000 active clients
time (in us)
(lower is better)
file descriptors
muduo
libevent2
muduo (ctl_add)

从两条线对比可以看出：

1. libevent2 在初始化 event watcher 方面比 muduo 快 20%（左边的两个图）
2. 在事件处理方面（右边的两个图）：
   (a) 在 100 个活动连接的情况下，
   当总连接数（并发数）小于 1000 或大于 30000 时时，二者性能差不多；
   当总连接数大于 1000 小于 30000 时，libevent2 明显领先。
   (b) 在 1000 个活动连接的情况下，
   当并发数小于 10000 时，libevent2 和 muduo 得分接近；
   当并发数大于 10000 时，muduo 明显占优。

这里我们有两个问题：

1. 为什么 muduo 花在初始化上的时间比较多？
2. 为什么在一些情况下它比 libevent2 慢很多。

我仔细分析了其中的原因，并参考了 libev 的作者 Marc Lehmann 的观点[17]，结论是：在第一轮初始化时，libevent2 和 muduo 都是用 `epoll_ctl(fd, EPOLL_CTL_ADD, ...)` 来添加文件描述符的 event watcher。不同之处在于，在后面 24 轮中，muduo 使用了 `epoll_ctl(fd, EPOLL_CTL_MOD, ...)` 来更新已有的 event watcher；然而 libevent2 继续调用 `epoll_ctl(fd, EPOLL_CTL_ADD, ...)` 来重复添加 fd，并忽略返回的错误码 EEXIST (File exists)。在这种重复添加的情况下，`EPOLL_CTL_ADD` 将会快速地返回错误，而 `EPOLL_CTL_MOD` 会做更多的工作，花的时间也更长。于是 libevent2 捡了个便宜。

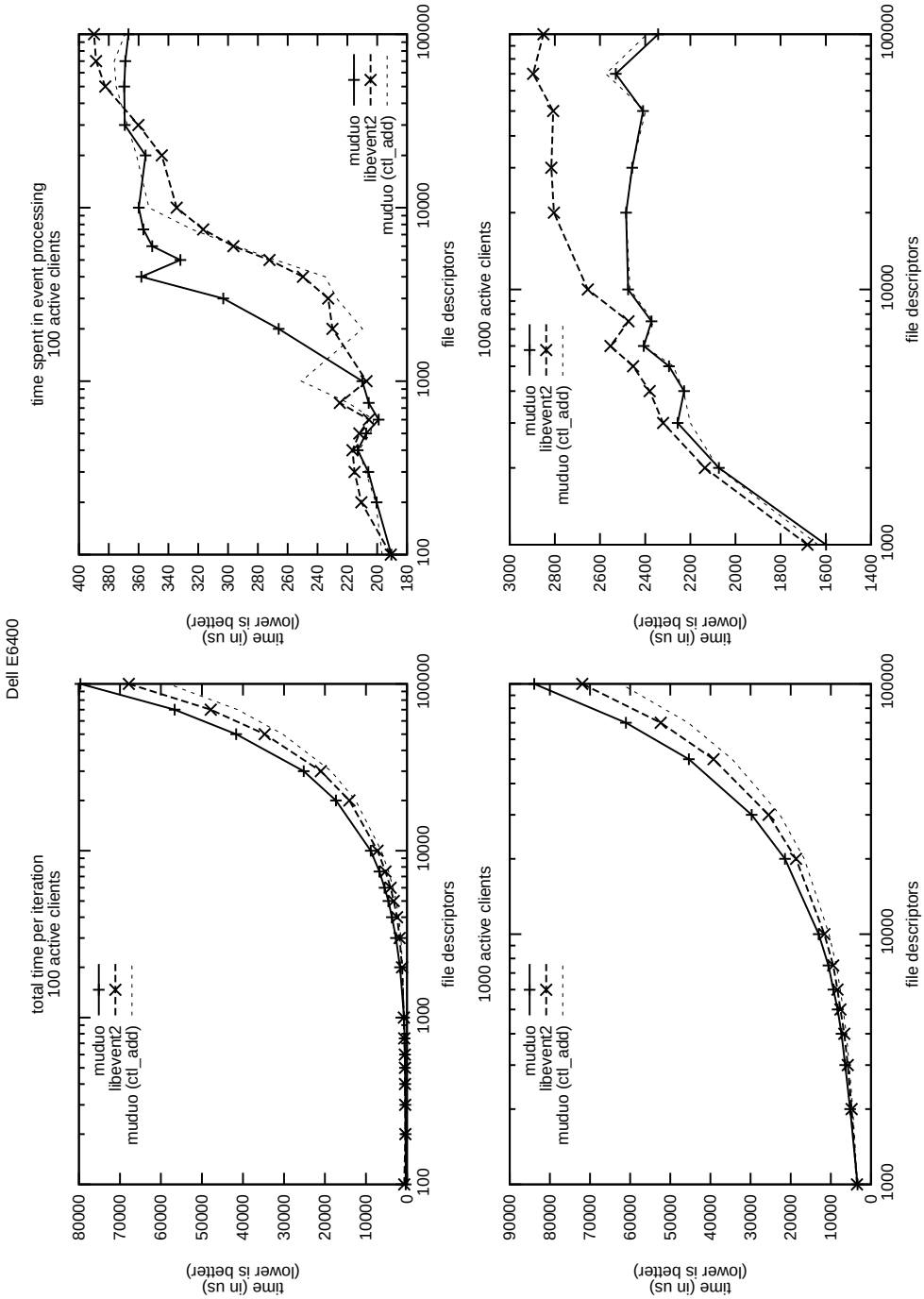
为了验证这个结论，我改动了 muduo，让它每次都用 `EPOLL_CTL_ADD` 方式初始化和更新 event watcher，并忽略返回的错误。

第二轮测试结果见上图的虚线，可见改动之后的 muduo 的初始化性能比 libevent2 更好，事件处理的耗时也有所降低（我推测是 kernel 内部的原因）。

这个改动只是为了验证想法，我并没有把它放到 muduo 最终的代码中去，这或许可以留作日后优化的余地。（具体的改动是 muduo/net/poller/EPollPoller.cc 第 138 行和 173 行，读者可自行验证。）

同样的测试在双核笔记本电脑上运行了一次，结果如下：（我的笔记本的 CPU 主频是 2.4GHz，高于台式机的 1.86GHz，所以用时较少。）

[17] http://lists.schmorp.de/pipermail/libev/2010q2/001041.html


Dell E6400
time spent in event processing
100 active clients
1000 active clients
total time per iteration
100 active clients
1000 active clients
time (in us)
(lower is better)
file descriptors
muduo
libevent2
muduo (ctl_add)

结论：在事件处理效率方面，muduo 与 libevent2 总体比较接近，各擅胜场。在并发量特别大的情况下（大于 10k），muduo 略微占优。

1.5.3 muduo 与 nginx 吞吐量对比

本节简单对比了 nginx 1.0.12 和 muduo 0.3.1 内置的简陋 http 服务器的长连接性能。其中 muduo 的 http 实现和测试代码位于 muduo/net/http/。

测试环境

- 服务端，运行 http server，8 核 DELL 490 工作站，Xeon E5320 CPU
- 客户端，运行 ab [18] 和 weighttp [19]，4 核 i5-2500 CPU
- 网络：普通家用千兆网

测试方法 为了公平起见，nginx 和 muduo 都没有访问文件，而是直接返回内存中的数据。毕竟我们想比较的是程序的网络性能，而不是机器的磁盘性能。另外客户机的性能优于服务机，因为我们要给服务端 http server 施压，试图使其饱和，而不是测试 http client 的性能。

Muduo http 测试服务器的主要代码：

muduo/net/http/tests/HttpServer_test.cc

```
void onRequest(const HttpRequest& req, HttpResponse* resp)
{
  if (req.path() == "/") {
    // ...
  } else if (req.path() == "/hello") {
    resp->setStatusCode(HttpResponse::k200Ok);
    resp->setStatusMessage("OK");
    resp->setContentType("text/plain");
    resp->addHeader("Server", "Muduo");
    resp->setBody("hello, world!\n");
  } else {
    resp->setStatusCode(HttpResponse::k404NotFound);
    resp->setStatusMessage("Not Found");
    resp->setCloseConnection(true);
  }
}

int main(int argc, char* argv[])
{
  int numThreads = 0;
```

[18] http://httpd.apache.org/docs/2.4/programs/ab.html
[19] http://redmine.lighttpd.net/projects/weighttp/wiki

```
  if (argc > 1)
  {
    benchmark = true;
    Logger::setLogLevel(Logger::WARN);
    numThreads = atoi(argv[1]);
  }
  EventLoop loop;
  HttpServer server(&loop, InetAddress(8000), "dummy");
  server.setHttpCallback(onRequest);
  server.setThreadNum(numThreads);
  server.start();
  loop.loop();
}
```

muduo/net/http/tests/HttpServer_test.cc

Nginx 使用了章亦春的 http echo 模块 [20] 来实现直接返回内存数据。配置文件如下

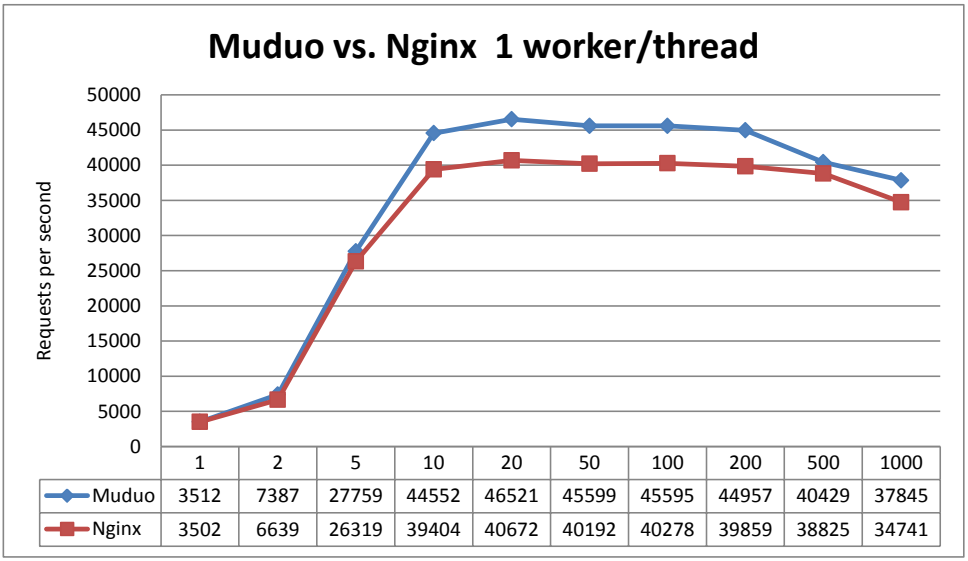
```
#user  nobody;
worker_processes  4;

events {
    worker_connections  10240;
}

http {
    include       mime.types;
    default_type  application/octet-stream;

    access_log  off;

    sendfile       on;
    tcp_nopush     on;

    keepalive_timeout  65;

    server {
        listen       8080;
        server_name  localhost;

        location / {
            root   html;
            index  index.html index.htm;
        }

        location /hello {
          default_type text/plain;
          echo "hello, world!";
        }
    }
}
```

[20] http://wiki.nginx.org/HttpEchoModule

客户端运行以下命令来获取 /hello 的内容，服务端返回字符串 "hello, world!"。

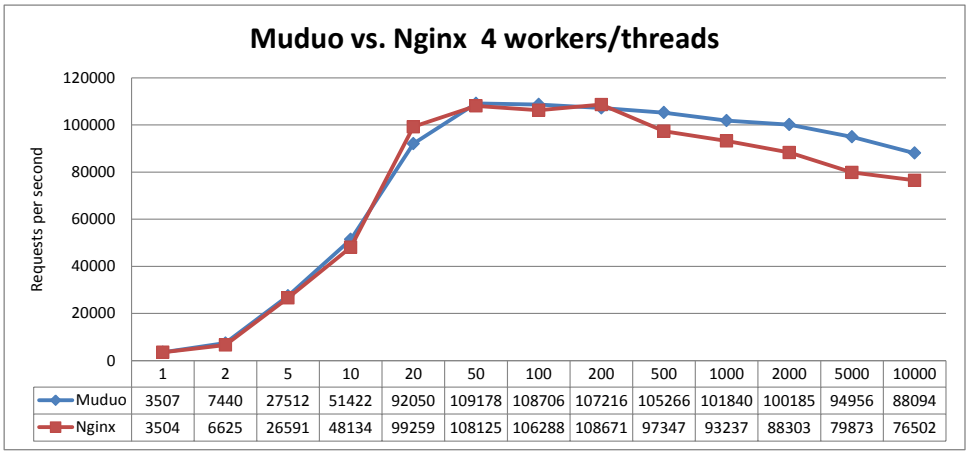
```
./ab -n 100000 -k -r -c 1000 10.0.0.9:8080/hello
```

先测试单线程的性能，横轴是并发连接数，纵轴为每秒完成的 http 请求响应数目，下同。在测试期间，ab 的 CPU 使用率低于 70%，客户端游刃有余。



	1	2	5	10	20	50	100	200	500	1000
Muduo	3512	7387	27759	44552	46521	45599	45595	44957	40429	37845
Nginx	3502	6639	26319	39404	40672	40192	40278	39859	38825	34741

再对比 muduo 4 线程和 nginx 4 工作进程的性能。当连接数大于 20 时，top(1) 显示 ab 的 CPU 使用率达到 85%，已经饱和，因此换用 weighttp （双线程）来完成其余测试。



	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000
Muduo	3507	7440	27512	51422	92050	109178	108706	107216	105266	101840	100185	94956	88094
Nginx	3504	6625	26591	48134	99259	108125	106288	108671	97347	93237	88303	79873	76502

CPU 使用率对比（百分比是 `top(1)` 显示的数值）：

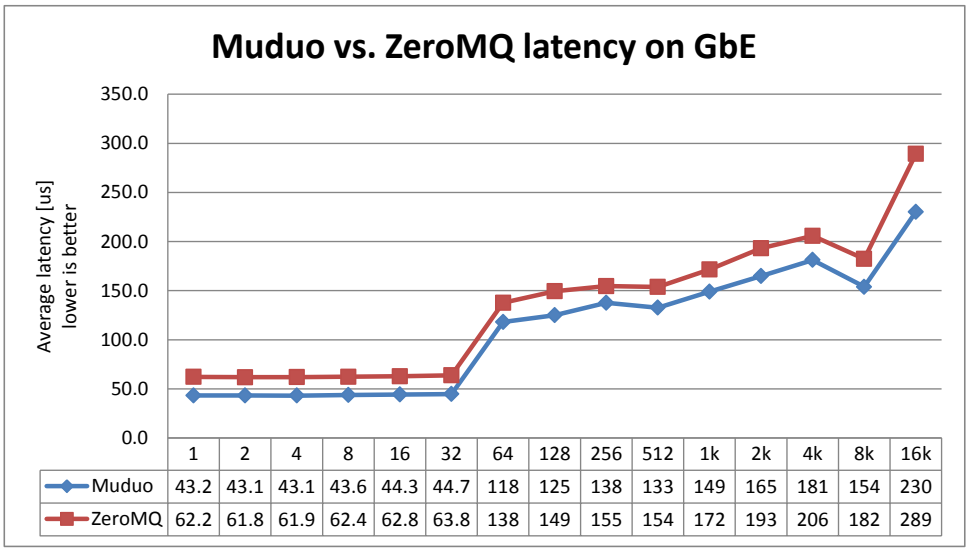
- 10k 并发连接，4 workers/threads，muduo 是 $4 \times 83\%$，nginx 是 $4 \times 75\%$
- 1k 并发连接，4 workers/threads，muduo 是 $4 \times 85\%$，nginx 是 $4 \times 78\%$

初看起来 nginx 的 CPU 使用率略低，但是实际上二者都已经把 CPU 资源耗尽了。与 CPU benchmark 不同，涉及 IO 的 benchmark 在满负载下的 CPU 使用率不会达到 100%，因为内核要占用一部分时间处理 IO。这里的数值差异说明 muduo 和 nginx 在满负荷的情况下，用户态和内核态的比重略有区别。

测试结果显示 muduo 多数情况下略快，nginx 和 muduo 在合适的条件下 QPS 都能超过 10 万。值得说明的是，Muduo 没有实现完整的 http 服务器，而只是实现了满足最基本要求的 http 协议，因此这个测试结果并不是说明 muduo 比 nginx 更适合用做 httpd，而是说明 muduo 在性能方面没有犯低级错误。

1.5.4 muduo 与 ZeroMQ 的延迟对比

本节我们用 ZeroMQ 自带的延迟与吞吐量测试[21]与 muduo 做一对比，muduo 代码位于 examples/zeromq/。测试的内容很简单，可以认为是第 1.5.1 节 ping pong 测试的翻版，不同之处在于这里的消息的长度是固定的，收到完整的消息再 echo 回发送方，如此往复。测试结果如下，横轴为消息的长度，纵轴为单程延迟（微秒）。可见在消息长度小于 16kB 的时，muduo 的延迟稳定地低于 ZeroMQ。



	1	2	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k
Muduo	43.2	43.1	43.1	43.6	44.3	44.7	118	125	138	133	149	165	181	154	230
ZeroMQ	62.2	61.8	61.9	62.4	62.8	63.8	138	149	155	154	172	193	206	182	289

[21] http://www.zeromq.org/results:perf-howto

1.6 详解 Muduo 多线程模型

本文以一个 Sudoku Solver 为例，回顾了并发网络服务程序的多种设计方案，并介绍了使用 muduo 网络库编写多线程服务器的两种最常用手法。下一章的例子展现了 Muduo 在编写单线程并发网络服务程序方面的能力与便捷性，今天我们先看一看它在多线程方面的表现。本文代码见：examples/sudoku/

1.6.1 数独求解服务器

假设有这么一个网络编程任务：写一个求解数独的程序 (Sudoku Solver)，并把它做成一个网络服务。

Sudoku Solver 是我喜爱的网络编程例子，它曾经出现在《分布式系统部署、监控与进程管理的几重境界》、《Muduo 设计与实现之一：Buffer 类的设计》、《〈多线程服务器的适用场合〉例释与答疑》等文中，它也可以看成是 echo 服务的一个变种（《谈一谈网络编程学习经验》把 echo 列为三大 TCP 网络编程案例之一）。

写这么一个程序在网络编程方面的难度不高，跟写 echo 服务差不多（从网络连接读入一个 Sudoku 题目，算出答案，再发回给客户），挑战在于怎样做才能发挥现在多核硬件的能力？在谈这个问题之前，让我们先写一个基本的单线程版。

协议

一个简单的以 `\r\n` 分隔的文本行协议，使用 TCP 长连接，客户端在不需要服务时主动断开连接。

请求：`[id:]<81digits>\r\n`

响应：`[id:]<81digits>\r\n`

或者：`[id:]NoSolution\r\n`

其中 `[id:]` 表示可选的 id，用于区分先后的请求，以支持 Parallel Pipelining，响应中会回显请求中的 id。Parallel Pipelining 的意义见赖勇浩的《以小见大——那些基于 protobuf 的五花八门的 RPC（2）》[22]，或者见我写的《分布式系统的工程化开发方法》第 54 页关于 out-of-order RPC 的介绍。

`<81digits>` 是 Sudoku 的棋盘，9×9 个数字，从左上角到右下角按行扫描，未知数字以 0 表示。如果 Sudoku 有解，那么响应是填满数字的棋盘；如果无解，则返回 NoSolution。

[22] http://blog.csdn.net/lanphaday/archive/2011/04/11/6316099.aspx

例子 1 请求：

```
000000010400000000020000000000050407008000300001090000300400200050100000000806000\r\n
```

响应：

```
693784512487512936125963874932651487568247391741398625319475268856129743274836159\r\n
```

例子 2 请求：

```
a:000000010400000000020000000000050407008000300001090000300400200050100000000806000\r\n
```

响应：

```
a:693784512487512936125963874932651487568247391741398625319475268856129743274836159\r\n
```

例子 3 请求：

```
b:000000010400000000020000000000050407008000300001090000300400200050100000000806005\r\n
```

响应：`b:NoSolution\r\n`

基于这个文本协议，我们可以用 telnet 模拟客户端来测试 sudoku solver，不需要单独编写 sudoku client。SudokuSolver 的默认端口号是 9981，因为它有 $9 \times 9 = 81$ 个格子。

基本实现

Sudoku 的求解算法见《谈谈数独 (Sudoku)》[23]一文，这不是本文的重点。假设我们已经有一个函数能求解 Sudoku，它的原型如下

```
string solveSudoku(const string& puzzle);
```

函数的输入是上文的"<81digits>"，输出是"<81digits>" 或"NoSolution"。这个函数是个 pure function，同时也是线程安全的。

有了这个函数，我们以《Muduo 网络编程示例之零：前言》中的 EchoServer 为蓝本，稍作修改就能得到 SudokuServer。这里只列出最关键的 onMessage() 函数，完整的代码见 examples/sudoku/server_basic.cc 。onMessage() 的主要功能是处理协议格式，并调用 solveSudoku() 求解问题。

examples/sudoku/server_basic.cc

```
const int kCells = 81;

void onMessage(const TcpConnectionPtr& conn, Buffer* buf, Timestamp)
{
```

[23] http://blog.csdn.net/Solstice/archive/2008/02/15/2096209.aspx

```
    LOG_DEBUG << conn->name();
    size_t len = buf->readableBytes();
    while (len >= kCells + 2)
    {
      const char* crlf = buf->findCRLF();
      if (crlf)
      {
        string request(buf->peek(), crlf);
        string id;
        buf->retrieveUntil(crlf + 2);
        string::iterator colon = find(request.begin(), request.end(), ':');
        if (colon != request.end())
        {
          id.assign(request.begin(), colon);
          request.erase(request.begin(), colon+1);
        }
        if (request.size() == implicit_cast<size_t>(kCells))
        {
          string result = solveSudoku(request);
          if (id.empty())
          {
            conn->send(result+"\r\n");
          }
          else
          {
            conn->send(id+":"+result+"\r\n");
          }
        }
        else
        {
          conn->send("Bad Request!\r\n");
          conn->shutdown();
        }
      }
      else
      {
        break;
      }
    }
  }
```

examples/sudoku/server_basic.cc

server_basic.cc 是一个并发服务器，可以同时服务多个客户连接。但是它是单线程的，无法发挥多核硬件的能力。

Sudoku 是一个计算密集型的任务（见《Muduo 设计与实现之一：Buffer 类的设计》中关于其性能的分析），其瓶颈在 CPU。为了让这个单线程 server_basic 程序充分利用 CPU 资源，一个简单的办法是在同一台机器上部署多个 server_basic 进程，让每个进程占用不同的端口，比如在一台 8 核机器上部署 8 个 server_basic 进程，分别占用 9981、9982、……、9988 端口。这样做其实是把难题推给了客户端，因为客

户端 (s) 要自己做负载均衡。再想得远一点，在 8 个 server_basic 前面部署一个 load balancer？似乎小题大做了。

能不能在一个端口上提供服务，并且又能发挥多核处理器的计算能力呢？当然可以，办法不止一种。

1.6.2 常见的并发网络服务程序设计方案

W. Richard Stevens 的 UNP2e 第 27 章 Client-Server Design Alternatives 介绍了十来种当时（90 年代末）流行的编写并发网络程序的方案。UNP3e 第 30 章，内容未变，还是这几种。以下简称 UNP CSDA 方案。UNP 这本书主要讲解阻塞式网络编程，在非阻塞方面着墨不多，仅有一章。正确使用 non-blocking IO 需要考虑的问题很多，不适宜直接调用 Sockets API，而需要一个功能完善的网络库支撑。

随着 2000 年前后第一次互联网浪潮的兴起，业界对高并发 http 服务器的强烈需求大大推动了这一领域的研究，目前高性能 httpd 普遍采用的是单线程 reactor 方式。另外一个说法是 IBM Lotus 使用 TCP 长连接协议，而把 Lotus 服务端移植到 Linux 的过程中 IBM 的工程师们大大提高了 Linux 内核在处理并发连接方面的可伸缩性，因为一个公司可能有上万人同时上线，连接到同一台跑着 Lotus server 的 Linux 服务器。

可伸缩网络编程这个领域其实近十年来没什么新东西，POSA2 已经作了相当全面的总结，另外以下几篇文章也值得参考。

- http://bulk.fefe.de/scalable-networking.pdf
- http://www.kegel.com/c10k.html
- http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf

下表是陈硕总结的 12 种常见方案。其中“多连接互通”指的是如果开发 chat 服务，多个客户连接之间是否能方便地交换数据（chat 也是《谈一谈网络编程学习经验》中举的三大 TCP 网络编程案例之一）。对于 echo/http/sudoku 这类“连接相互独立”的服务程序，这个功能无足轻重，但是对于 chat 类服务至关重要。“顺序性”指的是在 http/sudoku 这类请求 -响应服务中，如果客户连接顺序发送多个请求，那么计算得到的多个响应是否按相同的顺序发还给客户（这里指的是在自然条件下，不含刻意同步）。

方案	并发模型	UNP 对应	多进程	多线程	阻塞 IO	IO 复用	长连接	并发性	多核	开销	互通	顺序性	线程数量确定	特点
0	accept+read/write	0	no	no	Y	no	no	无	no	低	no	Y	Y	一次服务一个客户
1	accept+fork	1	Y	no	Y	no	Y	低	Y	高	no	Y	no	process-per-connection
2	accept+thread	6	no	Y	Y	no	Y	中	Y	中	Y	Y	no	thread-per-connection
3	prefork	2/3/4/5	Y	no	Y	no	Y	低	Y	高	no	Y	no	见 UNP
4	pre threaded	7/8	no	Y	Y	no	Y	中	Y	中	Y	Y	no	见 UNP
5	poll (reactor)	sec6.8	no	no	no	Y	Y	高	no	低	Y	Y	Y	单线程 reactor
6	reactor +thread-per-task	无	no	Y	no	Y	Y	中	Y	中	Y	no	no	thread-per-request
7	reactor +worker thread	无	no	Y	no	Y	Y	中	Y	中	Y	Y	no	worker-thread-per-connection
8	reactor +thread poll	无	no	Y	no	Y	Y	高	Y	低	Y	no	Y	主线程 IO，工作线程计算
9	reactors in threads	无	no	Y	no	Y	Y	高	Y	低	Y	Y	Y	one loop per thread
10	reactors in processes	无	Y	no	no	Y	Y	高	Y	低	no	Y	Y	Nginx
11	reactors + thread pool	无	no	Y	no	Y	Y	高	Y	低	Y	no	Y	最灵活的 IO 与 CPU 配置

UNP CSDA 方案归入 0 ～ 5。方案 5 也是目前用得很多的单线程 reactor 方案，muduo 对此提供了很好的支持。6 和 7 其实不是实用的方案，只是作为过渡品。8 和 9 是本文重点介绍的方案，其实这两个方案已经在《多线程服务器的常用编程模型》一文中提到过，只不过当时我还没有写 muduo，无法用具体的代码示例来说明。

在对比各方案之前，我们先看看基本的 micro benchmark 数据（前两项由 Thread_bench.cc 测得，第三项由 BlockingQueue_bench.cc 测得，硬件为 E5320，内核 Linux 2.6.32。）：

- fork()+exit(): 534.7us
- pthread_create()+pthread_join(): 42.5us, 其中创建线程用 26.1us
- push/pop a blocking queue : 11.5us
- sudoku resolve: 100us (根据题目难度不同，浮动范围 20~200us)

方案 0 这其实不是并发服务器，而是 iterative 服务器，因为它一次只能服务一个客户。代码见 UNP figure 1.9，UNP 以此为对比其他方案的基准点。这个方案不适合长连接，到是很适合 daytime 这种 write-only 短连接服务。

方案 1 这是传统的 Unix 并发网络编程方案，UNP 称之为 child-per-client 或 fork()-per-client，另外也俗称 process-per-connection。这种方案适合并发连接数不大的情况。至今仍有一些网络服务程序用这种方式实现，比如 PostgreSQL 和 Perforce 的服务端。这种方案适合“计算响应的工作量远大于 fork() 的开销”这种情况，比如数据库服务器。这种方案适合长连接，但不太适合短连接，因为 fork() 开销大于求解 sudoku 的用时。

方案 2 这是传统的 Java 网络编程方案 thread-per-connection，在 Java 1.4 引入 NIO 之前，Java 网络服务程序多采用这种方案。它的初始化开销比方案 1 要小很多。这种方案的伸缩性受到线程数的限制，一两百个还行，几千个的话对操作系统的 scheduler 恐怕是个不小的负担。

方案 3 这是针对方案 1 的优化，UNP 详细分析了几种变化，包括对 accept 惊群问题的考虑。

方案 4 这是对方案 2 的优化，UNP 详细分析了它的几种变化。3 和 4 这两个方案都是 Apache httpd 长期使用的方案。

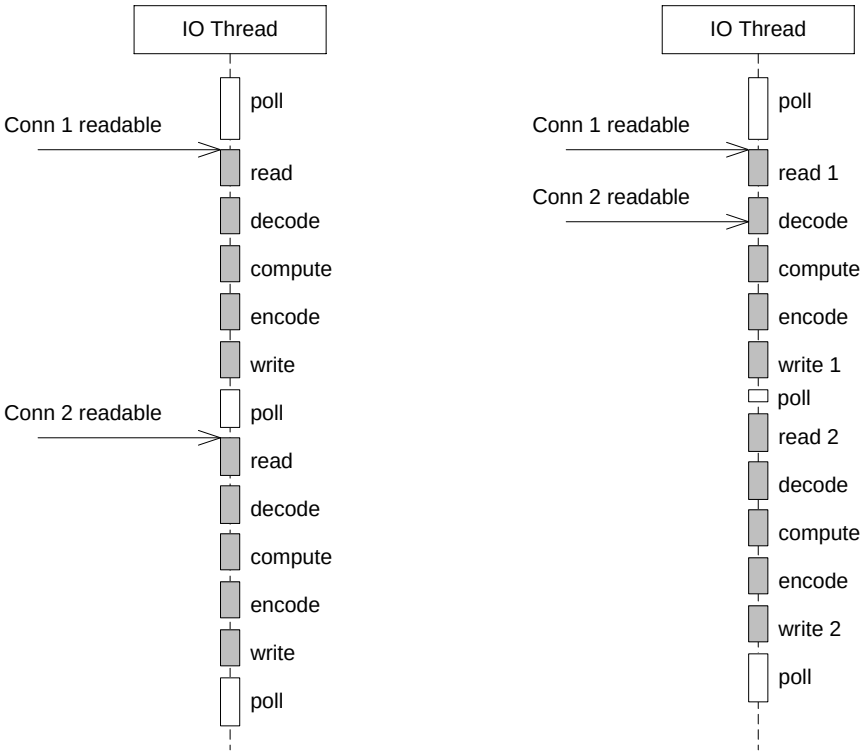
以上几种方案都是阻塞式网络编程，程序（thread-of-control）通常阻塞在 read() 上，等待数据到达。但是 TCP 是个全双工协议，同时支持 read() 和 write() 操作，当一个线程/进程阻塞在 read() 上，但程序又想给这个 TCP 连接发数据，那该怎么办？比如说 echo client，既要从 stdin 读，又要从网络读，当程序正在阻塞地读网络的时候，如何处理键盘输入？

又比如 proxy，既要把连接 a 收到的数据发给连接 b，又要把从连接 b 收到的数据发给连接 a，那么到底读哪个？（proxy 是《谈一谈网络编程学习经验》中举的三大 TCP 网络编程案例之一。）

一种方法是用两个线程/进程，一个负责读，一个负责写。UNP 也在实现 echo client 时介绍了这种方案。第 2.13 节举了一个 Python 双线程 tcp relay 的例子，另外见 Python Pinhole 的代码：http://code.activestate.com/recipes/114642/

另一种方法是使用 IO multiplexing，也就是 select/poll/epoll/kqueue 这一系列的“多路选择器”，让一个 thread-of-control 能处理多个连接。“IO 复用”其实复用的不是 IO 连接，而是复用线程。使用 select/poll 几乎肯定要配合 non-blocking IO，而使用 non-blocking IO 肯定要使用应用层 buffer，原因见《Muduo 设计与实现之一：Buffer 类的设计》。这就不是一件轻松的事儿了，如果每个程序都去搞一套自己的 IO multiplexing 机制（本质是 event-driven 事件驱动），这是一种很大的浪费。感谢 Doug Schmidt 为我们总结出了 Reactor 模式，让 event-driven 网络编程有章可循。继而出现了一些通用的 reactor 框架/库，比如 libevent、muduo、Netty、twisted、POE 等等。有了这些库，我想基本不用去编写阻塞式的网络程序了（特殊情况除外，比如 proxy 流量限制）。

单线程 reactor 的程序执行顺序如下图（左）。在没有事件的时候，线程等待在 select/poll/epoll_wait 函数上。注意由于只有一个线程，因此事件是顺序处理的。在这种协作式多任务中，事件的优先级得不到保证，因为从“poll 返回之后”到“下一次调用 poll 进入等待之前”这段时间内，线程不会被其他连接上的数据或事件抢占（下图右）。如果我们想要延迟计算（把 compute() 推迟 100ms），那么也不能用 sleep 之类的阻塞调用，而应该注册超时回调，以避免阻塞当前 IO 线程。



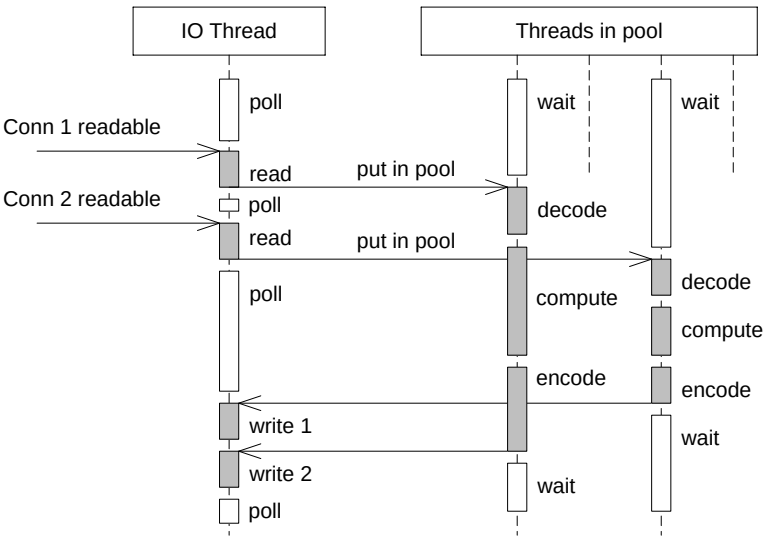
方案 5 基本的单线程 reactor 方案，即前面的 server_basic.cc 程序。本文以它作为对比其他方案的基准点。这种方案的优点是由网络库搞定数据收发，程序只关心业务逻辑；缺点在前面已经谈了：适合 IO 密集的应用，不太适合 CPU 密集的应用，因为较难发挥多核的威力。另外，与方案 2 相比，方案 5 处理网络消息的延迟可能要略大一些，因为方案 2 直接一次 `read(2)` 系统调用就能拿到请求数据，而方案 5 要先 `poll(2)` 再 `read(2)`，多了一次系统调用。

方案 6 这是一个过渡方案，收到 Sudoku 请求之后，不在 reactor 线程计算，而是创建一个新线程去计算，以充分利用多核 CPU。这是非常初级的多线程应用，因为它为每个请求（而不是每个连接）创建了一个新线程。这个开销可以用线程池来避免，即方案 8。这个方案还有一个特点是 out-of-order，即同时创建多个线程去计算同一个连接上收到的多个请求，那么算出结果的次序是不确定的，可能第 2 个 Sudoku 比较简单，比第 1 个先算出结果。这也是为什么我们在一开始设计协议的时候使用了 id，以便客户端区分 response 对应的是哪个 request。

方案 7 为了让返回结果的顺序确定，我们可以为每个连接创建一个计算线程，每个连接上的请求固定发给同一个线程去算，先到先得。这也是一个过渡方案，因为并发连接数受限于线程数目，这个方案或许还不如直接使用阻塞 IO 的 thread-per-connection 方案 2。

方案 7 与方案 6 的另外一个区别是单个 client 的最大 CPU 占用率，在方案 6 中，一个 TCP 连接上发来的一长串突发请求 (burst requests) 可以占满全部 8 个 core；而在方案 7 中，由于每个连接上的请求固定由同一个线程处理，那么它最多占用 12.5% 的 CPU 资源。这两种方案各有优劣，取决于应用场景的需要，到底是公平性重要还是突发性能重要。这个区别在方案 8 和方案 9 中同样存在，需要根据应用来取舍。

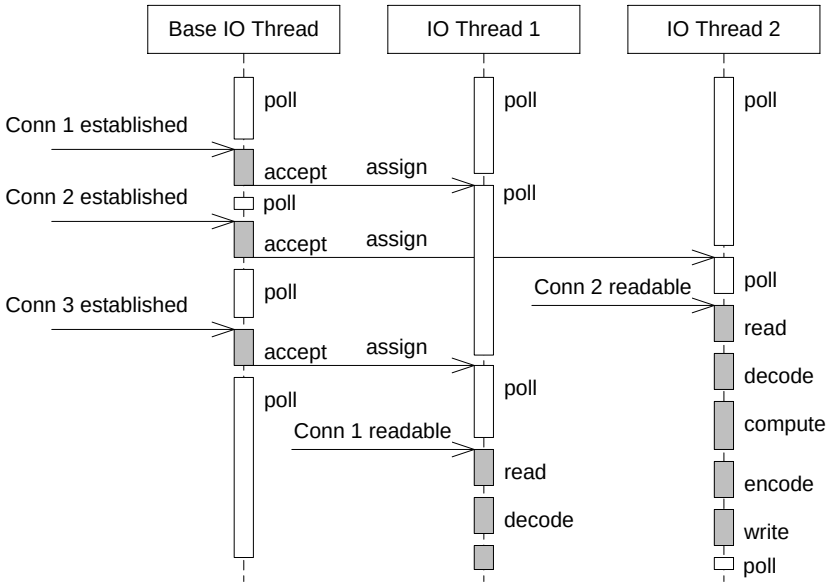
方案 8 为了弥补方案 6 中为每个请求创建线程的缺陷，我们使用固定大小线程池，程序结构如下图。全部的 IO 工作都在一个 reactor 线程完成，而计算任务交给 thread pool。如果计算任务彼此独立，而且 IO 的压力不大，那么这种方案是非常适用的。Sudoku Solver 正好符合。代码见：examples/sudoku/server_threadpool.cc 。后文给出了它与方案 9 的区别。



如果 IO 的压力比较大，一个 reactor 忙不过来，可以试试 multiple reactors 的方案 9。

方案 9 这是 muduo 内置的多线程方案，也是 Netty 内置的多线程方案。这种方案的特点是 one loop per thread，有一个 main reactor 负责 accept 连接，然后把连接挂在某个 sub reactor 中（muduo 采用 round-robin 的方式来选择 sub reactor），这

样该连接的所有操作都在那个 sub reactor 所处的线程中完成。多个连接可能被分派到多个线程中，以充分利用 CPU。Muduo 采用的是固定大小的 reactor pool，池子的大小通常根据 CPU 核数确定，也就是说线程数是固定的，这样程序的总体处理能力不会随连接数增加而下降。另外，由于一个连接完全由一个线程管理，那么请求的顺序性有保证，突发请求也不会占满全部 8 个核（如果需要优化突发请求，可以考虑方案 11）。这种方案把 IO 分派给多个线程，防止出现一个 reactor 的处理能力饱和。与方案 8 的线程池相比，方案 9 减少了进出 thread pool 的两次上下文切换，在把多个连接分散到多个 reactor 线程之后，小规模计算可以在当前 IO 线程完成并发回结果，从而降低响应的延迟。我认为这是一个适应性很强的多线程 IO 模型，因此把它作为 muduo 的默认线程模型。



方案 9 代码见：examples/sudoku/server_multiloop.cc 。它与 server_basic.cc 的区别很小，关键只有一行代码：server_.setThreadNum(numThreads);

```
$ diff server_basic.cc server_multiloop.cc -up
--- server_basic.cc     2011-06-15 13:40:59.000000000 +0800
+++ server_multiloop.cc 2011-06-15 13:39:53.000000000 +0800
@@ -21,19 +21,22 @@ using namespace muduo::net;
 class SudokuServer
 {
  public:
-  SudokuServer(EventLoop* loop, const InetAddress& listenAddr)
+  SudokuServer(EventLoop* loop, const InetAddress& listenAddr, int numThreads)
     : loop_(loop),
```

```
       server_(loop, listenAddr, "SudokuServer"),
+      numThreads_(numThreads),
       startTime_(Timestamp::now())
   {
     server_.setConnectionCallback(
         boost::bind(&SudokuServer::onConnection, this, _1));
     server_.setMessageCallback(
         boost::bind(&SudokuServer::onMessage, this, _1, _2, _3));
+    server_.setThreadNum(numThreads);
   }
```

方案 8 使用 thread pool 的代码与使用多 reactors 的方案 9 相比变化不大，只是把原来 onMessage() 中涉及计算和发回响应的部分抽出来做成一个函数，然后交给 ThreadPool 去计算。记住方案 8 有 out-of-order 的可能，客户端要根据 id 来匹配响应。

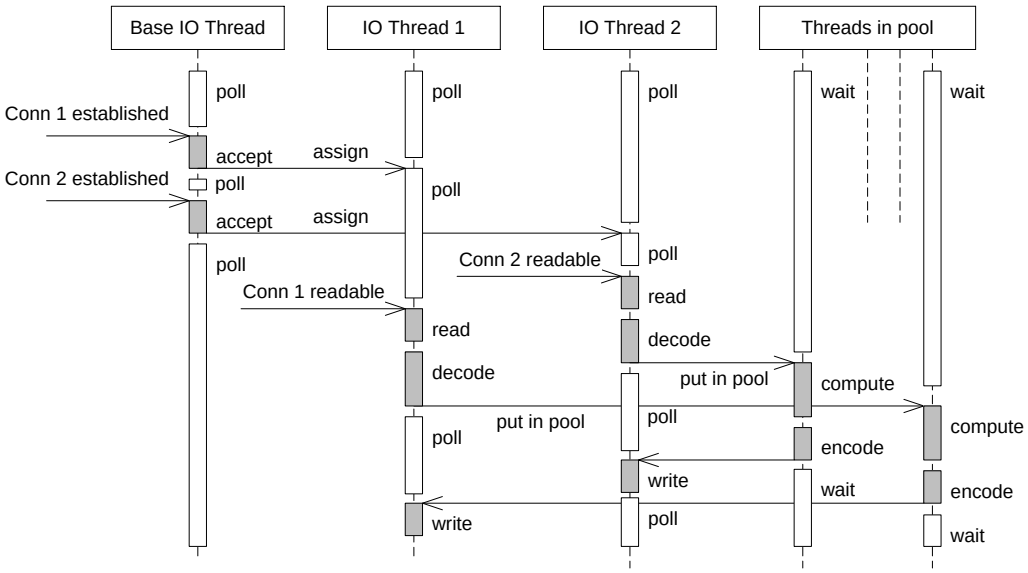
```
$ diff server_multiloop.cc server_threadpool.cc -up
--- server_multiloop.cc         2011-06-15 13:39:53.000000000 +0800
+++ server_threadpool.cc        2011-06-15 14:07:52.000000000 +0800
@@ -31,12 +32,12 @@ class SudokuServer
         boost::bind(&SudokuServer::onConnection, this, _1));
     server_.setMessageCallback(
         boost::bind(&SudokuServer::onMessage, this, _1, _2, _3));
-    server_.setThreadNum(numThreads);
   }

   void start()
   {
     LOG_INFO << "starting " << numThreads_ << " threads.";
+    threadPool_.start(numThreads_);
     server_.start();
   }

@@ -99,16 +100,7 @@ class SudokuServer

     if (puzzle.size() == implicit_cast<size_t>(kCells))
     {
-      LOG_DEBUG << conn->name();
-      string result = solveSudoku(puzzle);
-      if (id.empty())
-      {
-        conn->send(result+"\r\n");
-      }
-      else
-      {
-        conn->send(id+":"+result+"\r\n");
-      }
+      threadPool_.run(boost::bind(&solve, conn, puzzle, id));
     }
     else
```

```
     {
@@ -117,8 +109,25 @@ class SudokuServer
     return goodRequest;
   }

+  static void solve(const TcpConnectionPtr& conn,
+                    const string& puzzle,
+                    const string& id)
+  {
+    LOG_DEBUG << conn->name();
+    string result = solveSudoku(puzzle);
+    if (id.empty())
+    {
+      conn->send(result+"\r\n");
+    }
+    else
+    {
+      conn->send(id+":"+result+"\r\n");
+    }
+  }
+
   EventLoop* loop_;
   TcpServer server_;
+  ThreadPool threadPool_;
   int numThreads_;
   Timestamp startTime_;
 };
```

完整代码见：examples/sudoku/server_threadpool.cc

方案 10 这是 Nginx 的内置方案。如果连接之间无交互，这种方案也是很好的选择。工作进程之间相互独立，可以热升级。

方案 11 把方案 8 和方案 9 混合，既使用多个 reactors 来处理 IO，又使用线程池来处理计算。这种方案适合既有突发 IO （利用多线程处理多个连接上的 IO），又有突发计算的应用（利用线程池把一个连接上的计算任务分配给多个线程去做）。



这种其实方案看起来复杂，其实写起来很简单，只要把方案 8 的代码加一行 server_.setThreadNum(numThreads); 就行，这里就不举例了。

一个程序到底是使用一个 event loop 还是使用多个 event loops 呢？ZeroMQ 的手册给出的建议是[24]，按照每千兆比特每秒的吞吐量配一个 event loop 的比例来设置 event loop 的数目，即 muduo::TcpServer::setThreadNum() 的参数。依据这条经验规则，在编写运行于千兆以太网上的网络程序时，用一个 event loop 就足以应付网络 IO。如果程序本身没有多少计算，而主要瓶颈在网络带宽，那么可以按这条规则来办，只用一个 event loop。另一方面，如果程序的 IO 带宽较小，计算量较大，而且对延迟不敏感，那么可以把计算放到 thread pool 中，也可以只用一个 event loop。

值得指出的是，以上假定了 TCP 连接是同质的，没有优先级之分，我们看重的是服务程序的总吞吐量。但是如果 TCP 连接有优先级之分，那么单个 event loop 可能不适合，正确的做法是把高优先级的连接用单独的 event loop 来处理。

在 muduo 中，属于同一个 EventLoop 的连接之间没有事件优先级的差别。我这么设计的原因是为了防止优先级反转。比方说一个服务程序有 10 个心跳连接，有 10 个数据请求连接，都归同属一个 event loop，我们认为心跳连接有较高的优先级，心跳连接上的事件应该优先处理。但是由于事件循环的特性，如果数据请求连接上的数据先于心跳连接到达（早到 1ms），那么这个 event loop 就会调用相应的 event

[24]http://www.zeromq.org/area:faq#toc3

handler 去处理数据请求，而在下一次 epoll_wait() 的时候再来处理心跳事件。因此在同一个 event loop 中区分连接的优先级并不能达到预想的效果。我们应该用单独的 event loop 来管理心跳连接，这样就能避免数据连接上的事件阻塞了心跳事件，因为它们分属不同的线程。

1.6.3 结语

我在《多线程服务器的常用编程模型》一文中说

> 总结起来，我推荐的多线程服务端编程模式为：event loop per thread + thread pool。
>
> - event loop 用作 non-blocking IO 和定时器。
> - 线程池用来做计算，具体可以是任务队列或消费者 -生产者队列。

当时（2010 年 2 月）我还说“以这种方式写服务器程序，需要一个优质的基于 Reactor 模式的网络库来支撑，我只用过 in-house 的产品，无从比较并推荐市面上常见的 C++ 网络库，抱歉。”

现在有了 muduo 网络库，我终于能够用具体的代码示例把思想完整地表达出来。归纳一下[25]，实用的方案有 5 种，muduo 直接支持后 4 种：

方案	名称	接受新连接	网络 IO	计算任务
2	thread-per-connection	1 个线程	N 线程	在网络线程进行
5	单线程 reactor	1 个线程	在连接线程进行	在连接线程进行
8	reactor + 线程池	1 个线程	在连接线程进行	C_2 线程
9	one loop per thread	1 个线程	C_1 线程	在网络线程进行
11	one loop per thread + 线程池	1 个线程	C_1 线程	C_2 线程

表中 N 表示并发连接数目，C_1 和 C_2 是与连接数无关、与 CPU 数目有关的常数。

[25]此表参考了《Characteristics of multithreading models for high-performance IO driven network applications》一文

2

Muduo 编程示例

这一系列文章介绍用 muduo 网络库完成常见的 TCP 网络编程任务。内容如下：

1. UNP 中的五个简单协议，包括 echo、daytime、time、discard、chargen 等。
2. 文件传输，示范非阻塞 TCP 网络程序中如何完整地发送数据
3. Boost.Asio 中的示例，包括 timer2~6、chat 等。chat 实现了 TCP 封包与拆包 (codec)。
4. Muduo Buffer 类的设计与使用
5. Protobuf 编码解码器 (codec) 与消息分发器 (dispatcher)
6. Java Netty 中的示例，包括 discard、echo、uptime 等，其中的 discard 和 echo 带流量统计功能。
7. 用于测试两台机器的往返延迟的 roundtrip
8. 用 timing wheel 踢掉空闲连接
9. 限制服务器的最大并发连接数
10. 一个基于 TCP 的应用层广播 hub
11. 云风的串并转换连接服务器 multiplexer，及其自动化测试。
12. socks4a 代理服务器，包括简单的 TCP 中继 (relay)。
13. 一个提供短址服务的 httpd 服务器
14. 与其他库的集成，包括 UDNS、c-ares DNS、curl 等等。

这些例子都比较简单，逻辑不复杂，代码也很短，适合摘取关键部分放到博客上。其中一些有一定的代表性与针对性，比如“如何传输完整的文件”估计是网络编程的初学者经常遇到的问题。请注意，muduo 是设计来开发内网的网络程序，它没有做任何安全方面的加强措施，如果用在公网上可能会受到攻击，在后面的例子中我会谈到这一点。

2.1 五个简单 TCP 协议

本节将介绍第一个示例：五个简单 TCP 网络服务协议，包括 echo (RFC 862)、discard (RFC 863)、chargen (RFC 864)、daytime (RFC 867)、time (RFC 868)，以及 time 协议的客户端。各协议的功能简介如下：

- discard - 丢弃所有收到的数据；
- daytime - 服务端 accept 连接之后，以字符串形式发送当前时间，然后主动断开连接；
- time - 服务端 accept 连接之后，以二进制形式发送当前时间（从 Epoch 到现在的秒数），然后主动断开连接；我们需要一个客户程序来把收到的时间转换为字符串。
- echo - 回显服务，把收到的数据发回客户端；
- chargen - 服务端 accept 连接之后，不停地发送测试数据。

以上五个协议使用不同的端口，可以放到同一个进程中实现，且不必使用多线程。完整的代码见 muduo/examples/simple 。

2.1.1 discard

Discard 恐怕算是最简单的长连接 TCP 应用层协议，它只需要关注“三个半事件”中的“消息/数据到达”事件，事件处理函数如下：

examples/simple/discard/discard.cc

```
33 void DiscardServer::onMessage(const TcpConnectionPtr& conn,
34                               Buffer* buf,
35                               Timestamp time)
36 {
37   string msg(buf->retrieveAsString());
38   LOG_INFO << conn->name() << " discards " << msg.size()
39            << " bytes received at " << time.toString();
40 }
```

examples/simple/discard/discard.cc

与前面第 16 页的 echo 服务相比，除了省略 namespace 外，关键区别在于少了第 40 行：将收到的数据发回客户端。

剩下的都是例行公事的代码，此处从略，读者可对比参考 echo 服务。

2.1.2 daytime

Daytime 是短连接协议，在发送完当前时间后，由服务端主动断开连接。它只需要关注“三个半事件”中的“连接已建立”事件，事件处理函数如下：

examples/simple/daytime/daytime.cc

```
27 void DaytimeServer::onConnection(const TcpConnectionPtr& conn)
28 {
29   LOG_INFO << "DaytimeServer - " << conn->peerAddress().toIpPort() << " -> "
30            << conn->localAddress().toIpPort() << " is "
31            << (conn->connected() ? "UP" : "DOWN");
32   if (conn->connected())
33   {
34     conn->send(Timestamp::now().toFormattedString() + "\n");
35     conn->shutdown();
36   }
37 }
```

examples/simple/daytime/daytime.cc

第 34 行发送时间字符串，第 35 行主动断开连接。剩下的都是例行公事的代码，为节省篇幅，此处从略。

用 netcat 扮演客户端，运行结果如下：

```
$ nc 127.0.0.1 2013
2011-02-02 03:31:26.622647    # 服务器返回的时间字符串，UTC 时区
```

2.1.3 time

Time 协议与 daytime 极为类似，只不过它返回的不是日期时间字符串，而是一个 32-bit 整数，表示从 1970-01-01 00:00:00Z 到现在的秒数。当然，这个协议有“2038 年问题”。服务端只需要关注“三个半事件”中的“连接已建立”事件，事件处理函数如下：

examples/simple/time/time.cc

```
27 void TimeServer::onConnection(const muduo::net::TcpConnectionPtr& conn)
28 {
29   LOG_INFO << "TimeServer - " << conn->peerAddress().toIpPort() << " -> "
30            << conn->localAddress().toIpPort() << " is "
31            << (conn->connected() ? "UP" : "DOWN");
32   if (conn->connected())
33   {
34     time_t now = ::time(NULL);
35     int32_t be32 = sockets::hostToNetwork32(static_cast<int32_t>(now));
36     conn->send(&be32, sizeof be32);
37     conn->shutdown();
```

```
38   }
39 }
```
examples/simple/time/time.cc

第 34、35 取当前时间并转换为网络字节序 (Big Endian)，第 36 行发送 32-bit 整数，第 37 行主动断开连接。剩下的都是例行公事的代码，为节省篇幅，此处从略。

用 netcat 扮演客户端，并用 hexdump 来打印二进制数据，运行结果如下：

```
$ nc 127.0.0.1 2037 | hexdump -C
00000000  4d 48 d0 d5                                       |MHÐÕ|
```

time_client

因为 time 服务端发送的是二进制数据，不便直接阅读，我们编写一个客户端来解析并打印收到的 4 个字节数据。这个程序只需要关注“三个半事件”中的“消息/数据到达”事件，事件处理函数如下：

examples/simple/timeclient/timeclient.cc
```
53 void onMessage(const TcpConnectionPtr& conn, Buffer* buf, Timestamp receiveTime)
54 {
55   if (buf->readableBytes() >= sizeof(int32_t))
56   {
57     const void* data = buf->peek();
58     int32_t be32 = *static_cast<const int32_t*>(data);
59     buf->retrieve(sizeof(int32_t));
60     time_t time = sockets::networkToHost32(be32);
61     Timestamp ts(time * Timestamp::kMicroSecondsPerSecond);
62     LOG_INFO << "Server time = " << time << ", " << ts.toFormattedString();
63   }
64   else
65   {
66     LOG_INFO << conn->name() << " no enough data " << buf->readableBytes()
67              << " at " << receiveTime.toFormattedString();
68   }
69 }
```
examples/simple/timeclient/timeclient.cc

注意其中考虑到了如果数据没有一次性收全，已经收到的数据会累积在 Buffer 里 (在 else 分支里没有调用 Buffer::retrieve*)，以等待后续数据到达，程序也不会阻塞。这样即便服务器一个字节一个字节地发送数据，代码还是能正常工作，这也是非阻塞网络编程必须在用户态使用接受缓冲的主要原因。

这是我们第一次用到 TcpClient class，完整的代码如下：

examples/simple/timeclient/timeclient.cc

```
17 class TimeClient : boost::noncopyable
18 {
19  public:
20   TimeClient(EventLoop* loop, const InetAddress& serverAddr)
21     : loop_(loop),
22       client_(loop, serverAddr, "TimeClient")
23   {
24     client_.setConnectionCallback(
25         boost::bind(&TimeClient::onConnection, this, _1));
26     client_.setMessageCallback(
27         boost::bind(&TimeClient::onMessage, this, _1, _2, _3));
28     // client_.enableRetry();
29   }
30
31   void connect()
32   {
33     client_.connect();
34   }
35
36  private:
37
38   EventLoop* loop_;
39   TcpClient client_;
40
41   void onConnection(const TcpConnectionPtr& conn)
42   {
43     LOG_INFO << conn->localAddress().toIpPort() << " -> "
44              << conn->peerAddress().toIpPort() << " is "
45              << (conn->connected() ? "UP" : "DOWN");
46
47     if (!conn->connected())
48     {
49       loop_->quit();
50     }
51   }
```

以上第 49 行表示如果连接断开，就退出事件循环（第 82 行），程序也就终止了。

```
72 int main(int argc, char* argv[])
73 {
74   LOG_INFO << "pid = " << getpid();
75   if (argc > 1)
76   {
77     EventLoop loop;
78     InetAddress serverAddr(argv[1], 2037);
79
80     TimeClient timeClient(&loop, serverAddr);
81     timeClient.connect();
82     loop.loop();
83   }
84   else
```

```
85   {
86     printf("Usage: %s host_ip\n", argv[0]);
87   }
88 }
89
```

examples/simple/timeclient/timeclient.cc

程序的运行结果如下（有折行），假设 time server 运行在本机：

```
$ ./simple_timeclient 127.0.0.1
2011-02-02 04:10:35.181717  4296 INFO pid = 4296 - timeclient.cc:71
2011-02-02 04:10:35.183668  4296 INFO TcpClient::connect[TimeClient] -
                                    connecting to 127.0.0.1:2037 - TcpClient.cc:60
2011-02-02 04:10:35.185178  4296 INFO 127.0.0.1:40960 -> 127.0.0.1:2037
                                    is UP - timeclient.cc:39
2011-02-02 04:10:35.185279  4296 INFO Server time = 1296619835,
                                    2011-02-02 04:10:35.000000 - timeclient.cc:56
2011-02-02 04:10:35.185354  4296 INFO 127.0.0.1:40960 -> 127.0.0.1:2037
                                    is DOWN - timeclient.cc:39
```

2.1.4 echo

前面几个协议都是单向接收或发送数据，echo 是我们遇到的第一个双向的协议：服务端把客户端发过来的数据原封不动地传回去。它只需要关注“三个半事件”中的“消息/数据到达”事件，事件处理函数已在16页列出，这里复制一遍。

examples/simple/echo/echo.cc

```
33 void EchoServer::onMessage(const muduo::net::TcpConnectionPtr& conn,
34                           muduo::net::Buffer* buf,
35                           muduo::Timestamp time)
36 {
37   muduo::string msg(buf->retrieveAsString());
38   LOG_INFO << conn->name() << " echo " << msg.size() << " bytes, "
39            << "data received at " << time.toString();
40   conn->send(msg);
41 }
```

examples/simple/echo/echo.cc

这段代码实现的不是行回显 (line echo) 服务，而是有一点数据就发送一点数据。这样可以避免客户端恶意地不发送换行字符，而服务端又必须缓存已经收到的数据，导致服务器内存暴涨。但这个程序还是有一个安全漏洞，即如果客户端故意不断发送数据，但从不接收，那么服务端的发送缓冲区会一直堆积，导致内存暴涨。解决办法可以参考下面的 chargen 协议，或者在发送缓冲区累积到一定大小时主动断开连接。一般来说，非阻塞网络编程中正确处理数据发送比接收数据要困难，因为要应对对方接收缓慢的情况。

练习 1 ：修改 EchoServer::onMessage()，实现大小写互换。

练习 2 ：修改 EchoServer::onMessage()，实现 rot13 加密。

2.1.5 chargen

Chargen 协议很特殊，它只发送数据，不接收数据。而且，它发送数据的速度不能快过客户端接收的速度，因此需要关注“三个半事件”中的半个“消息/数据发送完毕”事件 (onWriteComplete)，事件处理函数如下：

examples/simple/chargen/chargen.cc

```
49 void ChargenServer::onConnection(const TcpConnectionPtr& conn)
50 {
51   LOG_INFO << "ChargenServer - " << conn->peerAddress().toIpPort() << " -> "
52            << conn->localAddress().toIpPort() << " is "
53            << (conn->connected() ? "UP" : "DOWN");
54   if (conn->connected())
55   {
56     conn->setTcpNoDelay(true);
57     conn->send(message_);
58   }
59 }
60
61 void ChargenServer::onMessage(const TcpConnectionPtr& conn,
62                               Buffer* buf,
63                               Timestamp time)
64 {
65   string msg(buf->retrieveAsString());
66   LOG_INFO << conn->name() << " discards " << msg.size()
67            << " bytes received at " << time.toString();
68 }
69
70 void ChargenServer::onWriteComplete(const TcpConnectionPtr& conn)
71 {
72   transferred_ += message_.size();
73   conn->send(message_);
74 }
```

examples/simple/chargen/chargen.cc

第 57 行，在连接建立时发生第一次数据；第 73 行，继续发送数据。剩下的都是例行公事的代码，为节省篇幅，此处从略。

完整的 chargen 服务端还带流量统计功能，用到了定时器，我们会在下一篇文章里介绍定时器的使用，到时候再回头来看相关代码。

用 netcat 扮演客户端，运行结果如下：

```
$ nc localhost 2019 | head
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghi
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijk
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklm
'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
)*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopq
```

2.1.6 五合一

前面五个程序都用到了 EventLoop，这其实是个 Reactor，用于注册和分发 IO 事件。Muduo 遵循 one loop per thread 模型，多个服务端 (TcpServer) 和客户端 (TcpClient) 可以共享同一个 EventLoop，也可以分配到多个 EventLoop 上以发挥多核多线程的好处。这里我们把五个服务端用同一个 EventLoop 跑起来，程序还是单线程的，功能却强大了很多：

examples/simple/allinone/allinone.cc

```
13 int main()
14 {
15   LOG_INFO << "pid = " << getpid();
16   EventLoop loop;  // one loop shared by multiple servers
17
18   ChargenServer ChargenServer(&loop, InetAddress(2019));
19   ChargenServer.start();
20
21   DaytimeServer daytimeServer(&loop, InetAddress(2013));
22   daytimeServer.start();
23
24   DiscardServer discardServer(&loop, InetAddress(2009));
25   discardServer.start();
26
27   EchoServer echoServer(&loop, InetAddress(2007));
28   echoServer.start();
29
30   TimeServer timeServer(&loop, InetAddress(2037));
31   timeServer.start();
32
33   loop.loop();
34 }
```

examples/simple/allinone/allinone.cc

以上几个协议的消息格式都非常简单，没有涉及 TCP 网络编程中常见的分包处理，在后文第 2.3 节讲 Boost.Asio 的聊天服务器时我们再来讨论这个问题。

2.2 文件传输

本节用发送文件的例子来说明 TcpConnection::send() 的使用。到目前为止，我们用到了 TcpConnection::send() 的两个重载，分别是 send(const string&) [1]和 send(const void* message, size_t len) [2]。

TcpConnection 目前提供了三个 send() 重载函数，原型如下。

muduo/net/TcpConnection.h

```
///
/// TCP connection, for both client and server usage.
///
class TcpConnection : boost::noncopyable,
                      public boost::enable_shared_from_this<TcpConnection>
{
 public:

  void send(const void* message, size_t len);
  void send(const StringPiece& message);
  void send(Buffer* message);  // this one might swap data without copying
  // void send(Buffer&& message); // C++11
  // void send(string&& message); // C++11

};
```

muduo/net/TcpConnection.h

在非阻塞网络编程中，发送消息通常是由网络库完成的，用户代码不会直接调用 write(2) 或 send(2) 等系统调用。原因见第 77 页“TcpConnection 必须要有 output buffer”。在使用 TcpConnection::send() 时值得注意的有几点：

- send() 的返回类型是 void，意味着用户不必关心调用 send() 时成功发送了多少字节，muduo 库会保证把数据发送给对方。
- send() 是非阻塞的。意味着客户代码只管把一条消息准备好，调用 send() 来发送，即便 TCP 的发送窗口满了，也绝对不会阻塞当前调用线程。
- send() 是线程安全、原子的。多个线程可以同时调用 send()，消息之间不会混叠或交织。但是多个线程同时发送的消息 (s) 的先后顺序是不确定的，muduo 只能保证每个消息本身的完整性 [3]。另外 send() 在多线程下仍然是非阻塞的。
- send(const void* message, size_t len) 这个重载最平淡无奇，可以发送任意字节序列。

[1]第 16 页第 40 行代码
[2]第 52 页第 36 行代码
[3]假设两个线程同时各自发送了一条任意长度的消息，那么这两条消息 a、b 的发送顺序要么是先 a 后 b，要么是先 b 后 a，不会出现 [a 的前一半, b, a 的后一半] 这种情况

- send(const StringPiece& message) 这个重载可以发送 string 和 const char*，其中 StringPiece[4] 是 Google 发明的专门用于传递字符串参数的 class，这样程序里就不必为 const char* 和 const string& 提供两份重载了。
- send(Buffer*) 有点特殊，它以指针为参数，而不是常见的 const 引用，因为函数里边可能用 Buffer::swap() 来高效地交换数据，避免内存拷贝 [5]，起到类似 C++ 右值引用的效果。
- 如果将来支持 C++11，那么可以增加对右值引用的重载，这样可以用 move 语义来避免内存拷贝。

下面我们来实现一个发送文件的命令行小工具，这个工具的协议很简单，在启动时通过命令行参数指定要发送的文件，然后在 2021 端口侦听，每当有新连接进来，就把文件内容完整地发送给对方。

如果不考虑并发，那么这个功能用 netcat 加重定向就能实现。这里展示的版本更加健壮，比方说发送 100MB 的文件，支持上万个并发客户连接；内存消耗只能并发连接数有关，跟文件大小无关；任何连接可以在任何时候断开，程序不会有内存泄漏或崩溃[6]。

我们一共写了三个版本，代码位于 examples/filetransfer。

1. 一次性把文件读入内存，一次性调用 send(const string&) 发送完毕，这个版本满足除了“内存消耗只能并发连接数有关，跟文件大小无关”之外的健壮性要求。
2. 一块一块地发送文件，减少内存使用，用到了 WriteCompleteCallback，这个版本满足了上述全部健壮性要求。
3. 同 2，但是采用 shared_ptr 来管理 FILE*，避免手动调用::fclose(3)。

版本一

在建立好连接之后，把文件的全部内容读入一个 string，一次性调用 TcpConnection::send() 发送。不用担心文件发送不完整。也不用担心 send() 之后立刻 shutdown() 会有什么问题，见下一节的说明。

[4] 代码位于 muduo/base/StringPiece.h
[5] 目前的实现尚未照此办理
[6] 我用 Java 实现了压力测试，代码位于 examples/filetransfer/loadtest

examples/filetransfer/download.cc

```
const char* g_file = NULL;

string readFile(const char* filename);  // read file content to string

void onConnection(const TcpConnectionPtr& conn)
{
  LOG_INFO << "FileServer - " << conn->peerAddress().toIpPort() << " -> "
           << conn->localAddress().toIpPort() << " is "
           << (conn->connected() ? "UP" : "DOWN");
  if (conn->connected())
  {
    LOG_INFO << "FileServer - Sending file " << g_file
             << " to " << conn->peerAddress().toIpPort();
    string fileContent = readFile(g_file);
    conn->send(fileContent);
    conn->shutdown();
    LOG_INFO << "FileServer - done";
  }
}

int main(int argc, char* argv[])
{
  LOG_INFO << "pid = " << getpid();
  if (argc > 1)
  {
    g_file = argv[1];

    EventLoop loop;
    InetAddress listenAddr(2021);
    TcpServer server(&loop, listenAddr, "FileServer");
    server.setConnectionCallback(onConnection);
    server.start();
    loop.loop();
  }
  else
  {
    fprintf(stderr, "Usage: %s file_for_downloading\n", argv[0]);
  }
}
```

examples/filetransfer/download.cc

注意每次建立连接的时候我们都去重新读一遍文件，这是考虑到文件有可能被其他程序修改，如果文件是 immutable 的，整个程序可以共享同一个 fileContent 对象。

这个版本有一个明显的缺陷，即内存消耗与 (并发连接数 × 文件大小) 成正比，文件越大内存消耗越多，如果文件大小上 GB，那几乎就是灾难了。只需要建立少量并发连接就能把服务器内存耗尽，因此我们有版本二。

版本二

为了解决版本一占用内存过多的问题，我们采用流水线的思路，当新建连接时，先发送文件的前 64k 数据，等这块数据发送完毕时再继续发送下 64k 数据，如此往复直到文件内容全部发送完毕。代码中使用了 TcpConnection::setContext() 和 getContext() 来保存 TcpConnection 的用户上下文（这里是 FILE*），因此不必使用额外的 std::map<TcpConnectionPtr, FILE*> 来记住每个连接的当前文件位置。

examples/filetransfer/download2.cc

```
15 const int kBufSize = 64*1024;
16 const char* g_file = NULL;
17
18 void onConnection(const TcpConnectionPtr& conn)
19 {
20   LOG_INFO << "FileServer - " << conn->peerAddress().toIpPort() << " -> "
21            << conn->localAddress().toIpPort() << " is "
22            << (conn->connected() ? "UP" : "DOWN");
23   if (conn->connected())
24   {
25     LOG_INFO << "FileServer - Sending file " << g_file
26              << " to " << conn->peerAddress().toIpPort();
27     conn->setHighWaterMarkCallback(onHighWaterMark, kBufSize+1);
28
29     FILE* fp = ::fopen(g_file, "rb");
30     if (fp)
31     {
32       conn->setContext(fp);
33       char buf[kBufSize];
34       size_t nread = ::fread(buf, 1, sizeof buf, fp);
35       conn->send(buf, nread);
36     }
37     else
38     {
39       conn->shutdown();
40       LOG_INFO << "FileServer - no such file";
41     }
42   }
43   else
44   {
45     if (!conn->getContext().empty())
46     {
47       FILE* fp = boost::any_cast<FILE*>(conn->getContext());
48       if (fp)
49       {
50         ::fclose(fp);
51       }
52     }
53   }
54 }
```

在 onWriteComplete() 回调函数里边读取下一块文件数据，继续发送。

```
56 void onWriteComplete(const TcpConnectionPtr& conn)
57 {
58   FILE* fp = boost::any_cast<FILE*>(conn->getContext());
59   char buf[kBufSize];
60   size_t nread = ::fread(buf, 1, sizeof buf, fp);
61   if (nread > 0)
62   {
63     conn->send(buf, nread);
64   }
65   else
66   {
67     ::fclose(fp);
68     fp = NULL;
69     conn->setContext(fp);
70     conn->shutdown();
71     LOG_INFO << "FileServer - done";
72   }
73 }
```

examples/filetransfer/download2.cc

注意每次建立连接的时候我们都去重新打开那个文件，使得程序中文件描述符的数量翻倍（每个连接占一个 socket fd 和一个 file fd），这是考虑到文件有可能被其他程序修改。如果文件是 immutable 的，一种改进措施是：整个程序可以共享同一个文件描述符，然后每个连接记住自己当前偏移量，在 onWriteComplete() 回调函数里用 pread(2) 来读取数据。

这个版本也存在一个问题，如果客户端故意只发起连接，不接收数据，那么要么把服务器进程的文件描述符耗尽，要么占用很多服务端内存（因为每个连接有 64kB 的发送缓冲区）。解决办法可参考后文 2.7 “限制服务器的最大并发连接数”和 2.10 “用 Timing wheel 踢掉空闲连接”。必须说明的是，muduo 并不是设计来编写面向公网的网络服务程序，这种服务程序需要在安全性方面下很多功夫，我个人对此不在行，我更关心实现内网（不一定是局域网）的高效服务程序。

版本三

用 shared_ptr 的 custom deleter 来减轻资源管理负担，使得 FILE* 的生命期和 TcpConnection 一样长，代码也更简单了。

examples/filetransfer/download3.cc

```
$ diff download2.cc download3.cc -U3
 const int kBufSize = 64*1024;
 const char* g_file = NULL;
+typedef boost::shared_ptr<FILE> FilePtr;
```

```
 void onConnection(const TcpConnectionPtr& conn)
 {
@@ -29,7 +32,8 @@
     FILE* fp = ::fopen(g_file, "rb");
     if (fp)
     {
-      conn->setContext(fp);
+      FilePtr ctx(fp, ::fclose);
+      conn->setContext(ctx);
       char buf[kBufSize];
       size_t nread = ::fread(buf, 1, sizeof buf, fp);
       conn->send(buf, nread);
@@ -40,33 +44,19 @@
       LOG_INFO << "FileServer - no such file";
     }
   }
-  else
-  {
-    if (!conn->getContext().empty())
-    {
-      FILE* fp = boost::any_cast<FILE*>(conn->getContext());
-      if (fp)
-      {
-        ::fclose(fp);
-      }
-    }
-  }
 }

 void onWriteComplete(const TcpConnectionPtr& conn)
 {
-  FILE* fp = boost::any_cast<FILE*>(conn->getContext());
+  FilePtr fp = boost::any_cast<FilePtr>(conn->getContext());
   char buf[kBufSize];
-  size_t nread = ::fread(buf, 1, sizeof buf, fp);
+  size_t nread = ::fread(buf, 1, sizeof buf, get_pointer(fp));
   if (nread > 0)
   {
     conn->send(buf, nread);
   }
   else
   {
-    ::fclose(fp);
-    fp = NULL;
-    conn->setContext(fp);
     conn->shutdown();
     LOG_INFO << "FileServer - done";
   }
 }
```

examples/filetransfer/download3.cc

2.2.1 为什么 TcpConnection::shutdown() 没有直接关闭 TCP 连接?

曾经收到一位网友来信: “在 simple 中的 daytime 示例中，服务端主动关闭时调用的是如下函数序列，这不是只是关闭了连接上的写操作吗，怎么是关闭了整个连接？”

```
void DaytimeServer::onConnection(const muduo::net::TcpConnectionPtr& conn)
{
  if (conn->connected())
  {
    conn->send(Timestamp::now().toFormattedString() + "\n");
    conn->shutdown();  // 调用 TcpConnection::shutdown()
  }
}

void TcpConnection::shutdown()
{
  if (state_ == kConnected)
  {
    setState(kDisconnecting);
    // 调用 TcpConnection::shutdownInLoop()
    loop_->runInLoop(boost::bind(&TcpConnection::shutdownInLoop, this));
  }
}

void TcpConnection::shutdownInLoop()
{
  loop_->assertInLoopThread();
  if (!channel_->isWriting())
  {
    // we are not writing
    socket_->shutdownWrite();  // 调用 Socket::shutdownWrite()
  }
}

void Socket::shutdownWrite()
{
  sockets::shutdownWrite(sockfd_);
}

void sockets::shutdownWrite(int sockfd)
{
  int ret = ::shutdown(sockfd, SHUT_WR);
  // 检查错误
}
```

陈硕答复如下:

Muduo TcpConnection 没有提供 close，而只提供 shutdown ，这么做是为了收发数据的完整性。

TCP 是一个全双工协议，同一个文件描述符既可读又可写，shutdownWrite() 关闭了“写”方向的连接，保留了“读”方向，这称为 TCP half-close。如果直接 close(socket_fd)，那么 socket_fd 就不能读或写了。

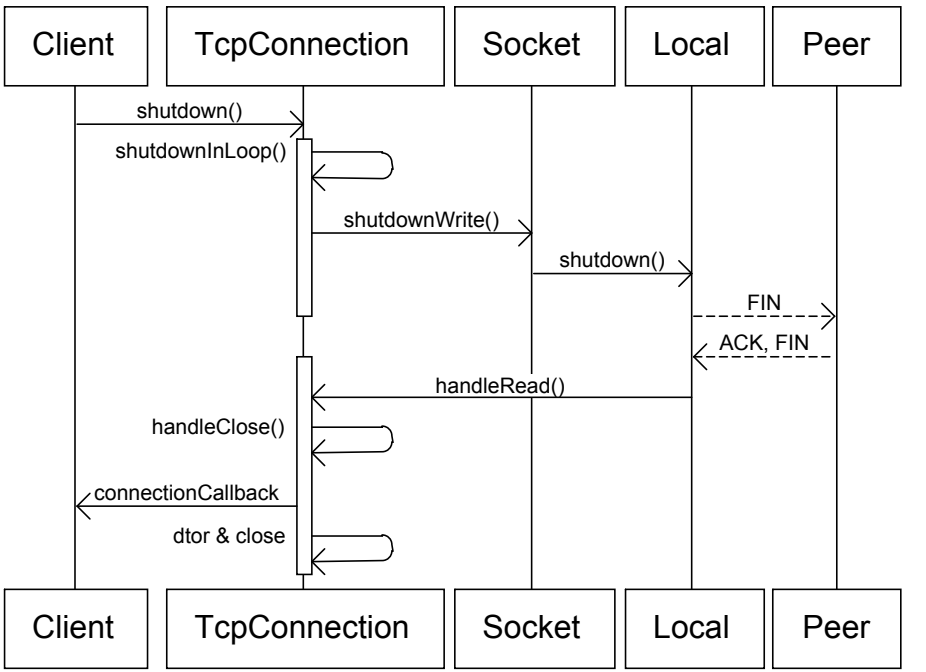
用 shutdown 而不用 close 的效果是，如果对方已经发送了数据，这些数据还“在路上”，那么 muduo 不会漏收这些数据。换句话说，muduo 在 TCP 这一层面解决了“当你打算关闭网络连接的时候，如何得知对方有没有发了一些数据而你还没有收到？”这一问题。当然，这个问题也可以在上面的协议层解决，双方商量好不再互发数据，就可以直接断开连接。

等于说 muduo 把“主动关闭连接”这件事情分成两步来做，如果要主动关闭连接，它会先关本地“写”端，等对方关闭之后，再关本地“读”端。

练习 阅读代码，回答“如果被动关闭连接，muduo 的行为如何？”
提示：muduo 在 read() 返回 0 的时候会回调 connection callback，这样客户代码就知道对方断开连接了。

Muduo 这种关闭连接的方式对对方也有要求，那就是对方 read() 到 0 字节之后会主动关闭连接（无论 shutdownWrite() 还是 close()），一般的网络程序都会这样，不是什么问题。当然，这么做有一个潜在的安全漏洞，万一对方故意不不关，那么 muduo 的连接就一直半开着，消耗系统资源。

完整的流程见下图。我们发完了数据，于是 shutdownWrite，发送 TCP FIN 分节，对方会读到 0 字节，然后对方通常会关闭连接，这样 muduo 会读到 0 字节，然后 muduo 关闭连接。（思考题：在 shutdown() 之后，muduo 回调 connection callback 的时间间隔大约是一个 round-trip time，为什么？）



如果有必要，对方可以在 read() 返回 0 之后继续发送数据，这是直接利用了 half-close TCP 连接。muduo 会收到这些数据，通过 message callback 通知客户代码。

那么 muduo 什么时候真正 close socket 呢？在 TcpConnection 对象析构的时候。TcpConnection 持有一个 Socket 对象，Socket 是一个 RAII handler，它的析构函数会 close(sockfd_)。这样，如果发生 TcpConnection 对象泄漏，那么我们从/proc/pid/fd/ 就能找到没有关闭的文件描述符，便于查错。

muduo 在 read() 返回 0 的时候会回调 connection callback，然后把 TcpConnection 的引用计数减一，如果 TcpConnection 的引用计数降到零，它就会析构了。

参考：

- 《TCP/IP 详解》第一卷第 18.5 节，TCP Half-Close。
- 《UNIX 网络编程》第一卷第三版第 6.6 节，shutdown() 函数。

网络编程中，发送往往比接收简单，下一节我们再谈接收消息的要领。

2.3 Boost.Asio 的聊天服务器

本文讲介绍一个与 Boost.Asio 的示例代码中的聊天服务器功能类似的网络服务程序，包括客户端与服务端的 muduo 实现。这个例子的主要目的是介绍如何处理分包，并初步涉及 Muduo 的多线程功能。本文的代码位于 examples/asio/chat/。

2.3.1 TCP 分包

前面一节“五个简单 TCP 协议”中处理的协议没有涉及分包，在 TCP 这种字节流协议上做应用层分包是网络编程的基本需求。分包指的是在发生一个消息 (message) 或一帧 (frame) 数据时，通过一定的处理，让接收方能从字节流中识别并截取（还原）出一个个消息。“粘包问题”是个伪问题。

对于短连接的 TCP 服务，分包不是一个问题，只要发送方主动关闭连接，就表示一条消息发送完毕，接收方 read() 返回 0，从而知道消息的结尾。例如前一节里的 daytime 和 time 协议。

对于长连接的 TCP 服务，分包有四种方法：

1. 消息长度固定，比如 muduo 的 roundtrip 示例就采用了固定的 16 字节消息；
2. 使用特殊的字符或字符串作为消息的边界，例如 HTTP 协议的 headers 以 "\r\n" 为字段的分隔符；
3. 在每条消息的头部加一个长度字段，这恐怕是最常见的做法，本文的聊天协议也采用这一办法；
4. 利用消息本身的格式来分包，例如 XML 格式的消息中 <root>...</root> 的配对，或者 JSON 格式中的 { ... } 的配对。解析这种消息格式通常会用到状态机。

在后文的代码讲解中还会仔细讨论用长度字段分包的常见陷阱。

2.3.2 聊天服务

本文实现的聊天服务非常简单，由服务端程序和客户端程序组成，协议如下：

- 服务端程序中某个端口侦听 (listen) 新的连接；
- 客户端向服务端发起连接；
- 连接建立之后，客户端随时准备接收服务端的消息并在屏幕上显示出来；
- 客户端接受键盘输入，以回车为界，把消息发送给服务端；

- 服务端接收到消息之后，依次发送给每个连接到它的客户端；原来发送消息的客户端进程也会收到这条消息；
- 一个服务端进程可以同时服务多个客户端进程。当有消息到达服务端后，每个客户端进程都会收到同一条消息，服务端广播发送消息的顺序是任意的，不一定哪个客户端会先收到这条消息。
- （可选）如果消息 A 先于消息 B 到达服务端，那么每个客户端都会先收到 A 再收到 B。

这实际上是一个简单的基于 TCP 的应用层广播协议，由服务端负责把消息发送给每个连接到它的客户端。参与“聊天”的既可以是人，也可以是程序。在以后的文章2.11中，我将介绍一个稍微复杂的一点的例子 hub，它有“聊天室”的功能，客户端可以注册特定的 topic(s)，并往某个 topic 发送消息，这样代码更有意思。

我在《谈一谈网络编程学习经验》[7] 中把聊天服务列为“最主要的三个例子”，与前面的“五个简单 TCP 协议”不同，聊天服务的特点是“连接之间的数据有交流，从 a 连接收到的数据要发给 b 连接。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？”muduo 的这个例子充分展示了解决以上问题的手法。

2.3.3 消息格式

本聊天服务的消息格式非常简单，“消息”本身是一个字符串，每条消息的有一个 4 字节的头部，以网络序存放字符串的长度。消息之间没有间隙，字符串也不一定以 '\0' 结尾。比方说有两条消息”hello” 和”chenshuo”，那么打包后的字节流是：

0x00, 0x00, 0x00, 0x05, 'h', 'e', 'l', 'l', 'o', 0x00, 0x00, 0x00, 0x08, 'c', 'h', 'e', 'n', 's', 'h', 'u', 'o'

共 21 字节。

打包的代码 这段代码把 string message 打包为 muduo::net::Buffer，并通过 conn 发送。由于这个 codec 的代码位于头文件中，因此反复出现了 muduo::net namespace。

examples/asio/chat/codec.h

```
55   void send(muduo::net::TcpConnection* conn,
56             const muduo::StringPiece& message)
57   {
58     muduo::net::Buffer buf;
```

[7] https://github.com/downloads/chenshuo/documents/LearningNetworkProgramming.pdf

```
59      buf.append(message.data(), message.size());
60      int32_t len = static_cast<int32_t>(message.size());
61      int32_t be32 = muduo::net::sockets::hostToNetwork32(len);
62      buf.prepend(&be32, sizeof be32);
63      conn->send(&buf);
64    }
```

examples/asio/chat/codec.h

muduo Buffer 有一个很好的功能，它在头部预留了 8 个字节的空间，这样第 61 行的 prepend() 操作就不需要移动已有的数据，效率较高。

分包的代码 解析数据往往比生成数据复杂，分包打包也不例外。

examples/asio/chat/codec.h

```
24    void onMessage(const muduo::net::TcpConnectionPtr& conn,
25                   muduo::net::Buffer* buf,
26                   muduo::Timestamp receiveTime)
27    {
28      while (buf->readableBytes() >= kHeaderLen) // kHeaderLen == 4
29      {
30        // FIXME: use Buffer::peekInt32()
31        const void* data = buf->peek();
32        int32_t be32 = *static_cast<const int32_t*>(data); // SIGBUS
33        const int32_t len = muduo::net::sockets::networkToHost32(be32);
34        if (len > 65536 || len < 0)
35        {
36          LOG_ERROR << "Invalid length " << len;
37          conn->shutdown();  // FIXME: disable reading
38          break;
39        }
40        else if (buf->readableBytes() >= len + kHeaderLen)
41        {
42          buf->retrieve(kHeaderLen);
43          muduo::string message(buf->peek(), len);
44          messageCallback_(conn, message, receiveTime);
45          buf->retrieve(len);
46        }
47        else
48        {
49          break;
50        }
51      }
52    }
```

examples/asio/chat/codec.h

第 43 行收到完整的消息，通过 `messageCallback_` 回调用户代码。第 32 行有潜在的问题，在某些不支持非对齐内存访问的体系结构上会造成 SIGBUS core dump，读取消息长度应该改用 `Buffer::peekInt32()`。

上面这段代码第 28 行用了 while 循环来反复读取数据，直到 Buffer 中的数据不够一条完整的消息。请读者思考，如果换成 `if (buf->readableBytes() >= kHeaderLen)` 会有什么后果。

以前面提到的两条消息的字节流为例：

0x00, 0x00, 0x00, 0x05, 'h', 'e', 'l', 'l', 'o', 0x00, 0x00, 0x00, 0x08, 'c', 'h', 'e', 'n', 's', 'h', 'u', 'o'

假设数据最终都全部到达，onMessage() 至少要能正确处理以下各种数据到达的次序，每种情况下 `messageCallback_` 都应该被调用两次：

1. 每次收到一个字节的数据，onMessage() 被调用 21 次；
2. 数据分两次到达，第一次收到 2 个字节，不足消息的长度字段；
3. 数据分两次到达，第一次收到 4 个字节，刚好够长度字段，但是没有 body；
4. 数据分两次到达，第一次收到 8 个字节，长度完整，但 body 不完整；
5. 数据分两次到达，第一次收到 9 个字节，长度完整，body 也完整；
6. 数据分两次到达，第一次收到 10 个字节，第一条消息的长度完整、body 也完整，第二条消息长度不完整；
7. 请自行移动分割点，验证各种情况；
8. 数据一次就全部到达，这时必须用 while 循环来读出两条消息，否则消息会堆积在 Buffer 中。

请读者验证 onMessage() 是否做到了以上几点。这个例子充分说明了 non-blocking read 必须和 input buffer 一起使用，而且在写 decoder 的时候一定要收到完整的消息再 retrieve（以上第 44 行）。

2.3.4 编解码器 LengthHeaderCodec

有人评论 Muduo 的接收缓冲区不能设置回调函数的触发条件 [8]，确实如此。每当 socket 可读，Muduo 的 TcpConnection 会读取数据并存入 Input Buffer，然后回调用户的函数。不过，一个简单的间接层就能解决问题，让用户代码只关心“消息到达”而不是“数据到达”，如本例中的 LengthHeaderCodec 所展示的那一样。

examples/asio/chat/codec.h

```
12 class LengthHeaderCodec : boost::noncopyable
13 {
14  public:
```

[8] http://www.cnblogs.com/Solstice/archive/2011/02/02/1948839.html#2022206

```
15    typedef boost::function<void (const muduo::net::TcpConnectionPtr&,
16                                  const muduo::string& message,
17                                  muduo::Timestamp)> StringMessageCallback;
18
19    explicit LengthHeaderCodec(const StringMessageCallback& cb)
20      : messageCallback_(cb)
21    {
22    }
```

onMessage() 和 send() 同前。

```
66   private:
67    StringMessageCallback messageCallback_;
68    const static size_t kHeaderLen = sizeof(int32_t);
69  };
```

examples/asio/chat/codec.h

这段代码把以 Buffer* 为参数的 MessageCallback 转换成了以 const string& 为参数的 StringMessageCallback，让用户代码不必关心分包操作，具体的调用时序图见第 103 页。如果编程语言相同，客户端和服务端可以（应该）共享同一个 codec，这样既节省工作量，又避免因对协议理解不一致而导致的错误。

2.3.5 服务端的实现

聊天服务器的服务端代码小于 100 行，不到 asio 的一半。

请先阅读第 65 行起的数据成员的定义。除了经常见到的 EventLoop 和 TcpServer，ChatServer 还定义了 codec_ 和 std::set<TcpConnectionPtr> connections_ 作为成员，connections_ 存放目前已建立的客户连接。在收到消息之后，服务器会遍历整个容器，把消息广播给其中每一个 TCP 连接。

首先，在构造函数里注册回调：

examples/asio/chat/server.cc

```
16  class ChatServer : boost::noncopyable
17  {
18   public:
19    ChatServer(EventLoop* loop,
20               const InetAddress& listenAddr)
21    : loop_(loop),
22      server_(loop, listenAddr, "ChatServer"),
23      codec_(boost::bind(&ChatServer::onStringMessage, this, _1, _2, _3))
24    {
25      server_.setConnectionCallback(
26          boost::bind(&ChatServer::onConnection, this, _1));
```

```
27      server_.setMessageCallback(
28          boost::bind(&LengthHeaderCodec::onMessage, &codec_, _1, _2, _3));
29    }
30
31    void start()
32    {
33      server_.start();
34    }
```

examples/asio/chat/server.cc

这里有几点值得注意，在以往的代码里是直接把本 class 的 onMessage() 注册给 server_；这里我们把 LengthHeaderCodec::onMessage() 注册给 server_，然后向 codec_ 注册了 ChatServer::onStringMessage()，等于说让 codec_ 负责解析消息，然后把完整的消息回调给 ChatServer。这正是我前面提到的“一个简单的间接层”，在不增加 Muduo 库的复杂度的前提下，提供了足够的灵活性让我们在用户代码里完成需要的工作。

另外，server_.start() 绝对不能在构造函数里调用，这么做将来会有线程安全的问题，见我在《当析构函数遇到多线程——C++ 中线程安全的对象回调》一文中的论述。

以下是处理连接的建立和断开的代码，注意它把新建的连接加入到 connections_ 容器中，把已断开的连接从容器中删除。这么做是为了避免内存和资源泄漏，TcpConnectionPtr 是 boost::shared_ptr<TcpConnection>，是 muduo 里唯一一个默认采用 shared_ptr 来管理生命期的对象。以后我们会谈到这么做的原因。

examples/asio/chat/server.cc

```
36  private:
37   void onConnection(const TcpConnectionPtr& conn)
38   {
39     LOG_INFO << conn->localAddress().toIpPort() << " -> "
40              << conn->peerAddress().toIpPort() << " is "
41              << (conn->connected() ? "UP" : "DOWN");
42
43     if (conn->connected())
44     {
45       connections_.insert(conn);
46     }
47     else
48     {
49       connections_.erase(conn);
50     }
51   }
```

以下是服务端处理消息的代码，它遍历整个 connections_ 容器，把消息打包发送给各个客户连接。

```
53    void onStringMessage(const TcpConnectionPtr&,
54                         const string& message,
55                         Timestamp)
56    {
57      for (ConnectionList::iterator it = connections_.begin();
58          it != connections_.end();
59          ++it)
60      {
61        codec_.send(get_pointer(*it), message);
62      }
63    }
```

数据成员：

```
65    typedef std::set<TcpConnectionPtr> ConnectionList;
66    EventLoop* loop_;
67    TcpServer server_;
68    LengthHeaderCodec codec_;
69    ConnectionList connections_;
70  };
```

examples/asio/chat/server.cc

main() 函数里边是例行公事的代码：

examples/asio/chat/server.cc

```
72  int main(int argc, char* argv[])
73  {
74    LOG_INFO << "pid = " << getpid();
75    if (argc > 1)
76    {
77      EventLoop loop;
78      uint16_t port = static_cast<uint16_t>(atoi(argv[1]));
79      InetAddress serverAddr(port);
80      ChatServer server(&loop, serverAddr);
81      server.start();
82      loop.loop();
83    }
84    else
85    {
86      printf("Usage: %s port\n", argv[0]);
87    }
88  }
```

examples/asio/chat/server.cc

如果你读过 asio 的对应代码，会不会觉得 Reactor 往往比 Proactor 容易使用？

2.3.6 客户端的实现

我有时觉得服务端的程序常常比客户端的更容易写，聊天服务器再次验证了我的看法。客户端的复杂性来自于它要读取键盘输入，而 EventLoop 是独占

线程的，所以我用了两个线程：main() 函数所在的线程负责读键盘，另外用一个 EventLoopThread 来处理网络 IO。[9]

来看代码，首先，在构造函数里注册回调，并使用了跟前面一样的 LengthHeaderCodec 作为中间层，负责打包分包。

examples/asio/chat/client.cc

```
17 class ChatClient : boost::noncopyable
18 {
19  public:
20   ChatClient(EventLoop* loop, const InetAddress& serverAddr)
21     : loop_(loop),
22       client_(loop, serverAddr, "ChatClient"),
23       codec_(boost::bind(&ChatClient::onStringMessage, this, _1, _2, _3))
24   {
25     client_.setConnectionCallback(
26         boost::bind(&ChatClient::onConnection, this, _1));
27     client_.setMessageCallback(
28         boost::bind(&LengthHeaderCodec::onMessage, &codec_, _1, _2, _3));
29     client_.enableRetry();
30   }
31
32   void connect()
33   {
34     client_.connect();
35   }
```

disconnect() 目前为空，客户端的连接由操作系统在进程终止时关闭。

```
37   void disconnect()
38   {
39     // client_.disconnect();
40   }
41
```

write() 会由 main 线程调用，所以要加锁，这个锁不是为了保护 TcpConnection，而是保护 shared_ptr。

```
42   void write(const StringPiece& message)
43   {
44     MutexLockGuard lock(mutex_);
45     if (connection_)
46     {
47       codec_.send(get_pointer(connection_), message);
48     }
49   }
```

[9]我暂时没有把标准输入输出融入 Reactor 的想法，因为服务器程序的 stdin 和 stdout 很少。

onConnection() 会由 EventLoop 线程调用，所以要加锁以保护 shared_ptr。

```
51  private:
52   void onConnection(const TcpConnectionPtr& conn)
53   {
54     LOG_INFO << conn->localAddress().toIpPort() << " -> "
55              << conn->peerAddress().toIpPort() << " is "
56              << (conn->connected() ? "UP" : "DOWN");
57
58     MutexLockGuard lock(mutex_);
59     if (conn->connected())
60     {
61       connection_ = conn;
62     }
63     else
64     {
65       connection_.reset();
66     }
67   }
```

把收到的消息打印到屏幕，这个函数由 EventLoop 线程调用，但是不用加锁，因为 printf() 是线程安全的。注意这里不能用 cout，它不是线程安全的。

```
69   void onStringMessage(const TcpConnectionPtr&,
70                        const string& message,
71                        Timestamp)
72   {
73     printf("<<< %s\n", message.c_str());
74   }
```

数据成员：

```
76   EventLoop* loop_;
77   TcpClient client_;
78   LengthHeaderCodec codec_;
79   MutexLock mutex_;
80   TcpConnectionPtr connection_;
81 };
```

examples/asio/chat/client.cc

main() 函数里除了例行公事，还要启动 EventLoop 线程和读取键盘输入。

examples/asio/chat/client.cc

```
83 int main(int argc, char* argv[])
84 {
85   LOG_INFO << "pid = " << getpid();
86   if (argc > 2)
87   {
88     EventLoopThread loopThread;
```

```
89      uint16_t port = static_cast<uint16_t>(atoi(argv[2]));
90      InetAddress serverAddr(argv[1], port);
91
92      ChatClient client(loopThread.startLoop(), serverAddr);
93      client.connect();
94      std::string line;
95      while (std::getline(std::cin, line))
96      {
97        client.write(line);
98      }
99      client.disconnect();
100   }
101   else
102   {
103     printf("Usage: %s host_ip port\n", argv[0]);
104   }
105 }
```

examples/asio/chat/client.cc

第 92 行，ChatClient 使用 EventLoopThread 的 EventLoop，而不是通常的主线程的 EventLoop。第 97 行，发送数据行。

简单测试开三个命令行窗口，在第一个运行

```
$ ./asio_chat_server 3000
```

第二个运行

```
$ ./asio_chat_client 127.0.0.1 3000
```

第三个运行同样的命令

```
$ ./asio_chat_client 127.0.0.1 3000 \\$
```

这样就有两个客户端进程参与聊天。在第二个窗口里输入一些字符并回车，字符会出现在本窗口和第三个窗口中。

代码示例中还有另外三个 server 程序，都是多线程的，详细介绍在第 2.11.1 节。

- server_threaded.cc 使用多线程 TcpServer，并用 Mutex 来保护共享数据。
- server_threaded_efficient.cc 对共享数据以“借 shared_ptr 实现 copy-on-write”手法来降低锁竞争。
- server_threaded_highperformance.cc 采用 thread local 变量，实现多线程高效转发。

后续文章我会介绍 Muduo 中的定时器，并实现 Boost.Asio 教程中的 timer2~5 示例，以及带流量统计功能的 discard 和 echo 服务器（来自 Java Netty）。流量等于单位时间内发送或接受的字节数，这要用到定时器功能。

2.4 Buffer 类的设计与使用

本文介绍 Muduo 中输入输出缓冲区的设计与实现。本文中 buffer 指一般的应用层缓冲区、缓冲技术，Buffer 特指 muduo::net::Buffer class。

2.4.1 Muduo 的 IO 模型

UNPv1 第 6.2 节总结了 Unix/Linux 上的五种 IO 模型：阻塞 (blocking)、非阻塞 (non-blocking)、IO 复用 (IO multiplexing)、信号驱动 (signal-driven)、异步 (asynchronous)。这些都是单线程下的 IO 模型。

C10k 问题[10]的页面介绍了五种 IO 策略，把线程也纳入考量。（现在 C10k 已经不是什么问题，C100k 也不是大问题，C1000k 才算得上挑战）。

在这个多核时代，线程是不可避免的。那么服务端网络编程该如何选择线程模型呢？我赞同 libev 作者的观点[11]：one loop per thread is usually a good model。之前我也不止一次表述过这个观点，见《多线程服务器的常用编程模型》[12] 《多线程服务器的适用场合》[13]。

如果采用 one loop per thread 的模型，多线程服务端编程的问题就简化为如何设计一个高效且易于使用的 event loop，然后每个线程 run 一个 event loop 就行了（当然、同步和互斥是不可或缺的）。在“高效”这方面已经有了很多成熟的范例（libev、libevent、memcached、varnish、lighttpd、nginx），在“易于使用”方面我希望 muduo 能有所作为。（muduo 可算是用现代 C++ 实现了 Reactor 模式，比起原始的 Reactor 来说要好用得多。）

event loop 是 non-blocking 网络编程的核心，在现实生活中，non-blocking 几乎总是和 IO-multiplexing 一起使用，原因有两点：

- 没有人真的会用轮询 (busy-pooling) 来检查某个 non-blocking IO 操作是否完成，这样太浪费 CPU cycles。
- IO-multiplex 一般不能和 blocking IO 用在一起，因为 blocking IO 中 read()/write()/accept()/connect() 都有可能阻塞当前线程，这样线程就没办法处理其他 socket 上的 IO 事件了。见 UNPv1 第 16.6 节“nonblocking accept”的例子。

[10] http://www.kegel.com/c10k.html
[11] http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#THREADS_AND_COROUTINES
[12] http://blog.csdn.net/Solstice/archive/2010/02/12/5307710.aspx
[13] http://blog.csdn.net/Solstice/archive/2010/02/28/5334243.aspx

所以，当我提到 non-blocking 的时候，实际上指的是 non-blocking + IO-multiplexing，单用其中任何一个是不现实的。另外，本文所有的“连接”均指 TCP 连接，socket 和 connection 在文中可互换使用。

当然，non-blocking 编程比 blocking 难得多，见第1.4.1节“TCP 网络编程本质论”一节列举的难点。基于 event loop 的网络编程跟直接用 C/C++ 编写单线程 Windows 程序颇为相像：程序不能阻塞，否则窗口就失去响应了；在 event handler 中，程序要尽快交出控制权，返回窗口的事件循环。

2.4.2 为什么 non-blocking 网络编程中应用层 buffer 是必须的？

Non-blocking IO 的核心思想是避免阻塞在 read() 或 write() 或其他 IO 系统调用上，这样可以最大限度地复用 thread-of-control，让一个线程能服务于多个 socket 连接。IO 线程只能阻塞在 IO-multiplexing 函数上，如 select()/poll()/epoll_wait()。这样一来，应用层的缓冲是必须的，每个 TCP socket 都要有 stateful 的 input buffer 和 output buffer。

TcpConnection 必须要有 output buffer 考虑一个常见场景：程序想通过 TCP 连接发送 100k 字节的数据，但是在 write() 调用中，操作系统只接受了 80k 字节（受 TCP advertised window 的控制，细节见 TCPv1），你肯定不想在原地等待，因为不知道会等多久（取决于对方什么时候接受数据，然后滑动 TCP 窗口）。程序应该尽快交出控制权，返回 event loop。在这种情况下，剩余的 20k 字节数据怎么办？

对于应用程序而言，它只管生成数据，它不应该关心到底数据是一次性发送还是分成几次发送，这些应该由网络库来操心，程序只要调用 TcpConnection::send() 就行了，网络库会负责到底。网络库应该接管这剩余的 20k 字节数据，把它保存在该 TCP connection 的 output buffer 里，然后注册 POLLOUT 事件，一旦 socket 变得可写就立刻发送数据。当然，这第二次 write() 也不一定能完全写入 20k 字节，如果还有剩余，网络库应该继续关注 POLLOUT 事件；如果写完了 20k 字节，网络库应该停止关注 POLLOUT，以免造成 busy loop。（Muduo EventLoop 采用的是 epoll level trigger，这么做的具体原因我以后再说。）

如果程序又写入了 50k 字节，而这时候 output buffer 里还有待发送的 20k 数据，那么网络库不应该直接调用 write()，而应该把这 50k 数据 append 在那 20k 数据之后，等 socket 变得可写的时候再一并写入。

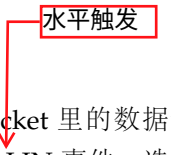
如果 output buffer 里还有待发送的数据，而程序又想关闭连接（对程序而言，调用 TcpConnection::send() 之后他就认为数据迟早会发出去），那么这时候网络库不

能立刻关闭连接，而要等数据发送完毕，见第 63 页“为什么 muduo 的 shutdown() 没有直接关闭 TCP 连接？”一节中的讲解。

综上，要让程序在 write 操作上不阻塞，网络库必须要给每个 tcp connection 配置 output buffer。

TcpConnection 必须要有 input buffer TCP 是一个无边界的字节流协议，接收方必须要处理“收到的数据尚不构成一条完整的消息”和“一次收到两条消息的数据”等等情况。一个常见的场景是，发送方 send 了两条 10k 字节的消息（共 20k），接收方收到数据的情况可能是：

- 一次性收到 20k 数据
- 分两次收到，第一次 5k，第二次 15k
- 分两次收到，第一次 15k，第二次 5k
- 分两次收到，第一次 10k，第二次 10k
- 分三次收到，第一次 6k，第二次 8k，第三次 6k
- 其他任何可能



网络库在处理“socket 可读”事件的时候，必须一次性把 socket 里的数据读完（从操作系统 buffer 搬到应用层 buffer），否则会反复触发 POLLIN 事件，造成 busy-loop。（Again, Muduo EventLoop 采用的是 epoll level trigger，这么做的具体原因我以后再说。）

那么网络库必然要应对“数据不完整”的情况，收到的数据先放到 input buffer 里，等构成一条完整的消息再通知程序的业务逻辑。这通常是 codec 的职责，见第2.3节“Boost.Asio 的聊天服务器”一文中的“TCP 分包”的论述与代码。

所以，在 tcp 网络编程中，网络库必须要给每个 tcp connection 配置 input buffer。

所有 muduo 中的 IO 都是带缓冲的 IO (buffered IO)，你不会自己去 read() 或 write() 某个 socket，只会操作 TcpConnection 的 input buffer 和 output buffer。更确切的说，是在 onMessage() 回调里读取 input buffer；调用 TcpConnection::send() 来间接操作 output buffer，一般不会直接操作 output buffer。

btw, muduo 的 onMessage() 的原型如下，它既可以是 free function，也可以是 member function，反正 muduo TcpConnection 只认 boost::function<>。

```
void onMessage(const TcpConnectionPtr& conn, Buffer* buf, Timestamp receiveTime);
```

对于网络程序来说，一个简单的验收测试是：输入数据每次收到一个字节（200 字节的输入数据会分 200 次收到，每次间隔 10 ms），程序的功能不受影响。对于 Muduo 程序，通常可以用 codec 来分离“消息接收”与“消息处理”，见第 2.6 节“在 muduo 中实现 protobuf 编解码器与消息分发器”对“编解码器 codec”的介绍。

如果某个网络库只提供相当于 char buf[8192] 的缓冲，或者根本不提供缓冲区，而仅仅通知程序“某 socket 可读/某 socket 可写”，要程序自己操心 IO buffering，这样的网络库用起来就很不方便了。（我有所指，你懂得。）那样就用proactor

2.4.3 Buffer 的要求

Muduo Buffer 的设计考虑了常见的网络编程需求，我试图在易用性和性能之间找一个平衡点，目前这个平衡点更偏向于易用性。

Muduo Buffer 的设计要点：

- 对外表现为一块连续的内存 (char*, len)，以方便客户代码的编写。
- 其 size() 可以自动增长，以适应不同大小的消息。它不是一个 fixed size array (即 char buf[8192])。
- 内部以 vector of char 来保存数据，并提供相应的访问函数。

writerIndex_ readerIndex_

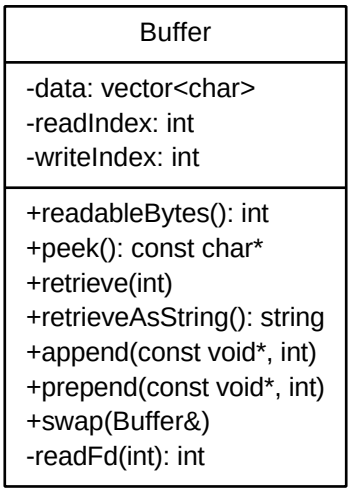
Buffer 其实像是一个 queue，从末尾写入数据，从头部读出数据。

谁会用 Buffer？谁写谁读？根据前文分析，TcpConnection 会有两个 Buffer 成员，input buffer 与 output buffer。

- input buffer，TcpConnection 会从 socket 读取数据，然后写入 input buffer（其实这一步是用 Buffer::readFd() 完成的）；客户代码从 input buffer 读取数据。
- output buffer，客户代码会把数据写入 output buffer （其实这一步是用 TcpConnection::send() 完成的）；TcpConnection 从 output buffer 读取数据并写入 socket。

其实，input 和 output 是针对客户代码而言，客户代码从 input 读，往 output 写。TcpConnection 的读写正好相反。

以下是 muduo::net::Buffer 的类图。请注意，为了后面画图方便，这个类图跟实际代码略有出入，但不影响我要表达的观点。代码位于 muduo/net/Buffer.{h,cc}



这里不介绍每个成员函数的作用，留给《Muduo 网络编程示例》系列。下文会仔细介绍 readIndex 和 writeIndex 的作用。

Buffer::readFd() 我在第 15 页写道

> 在非阻塞网络编程中，如何设计并使用缓冲区？一方面我们希望减少系统调用，一次读的数据越多越划算，那么似乎应该准备一个大的缓冲区。另一方面，我们系统减少内存占用。如果有 10k 个连接，每个连接一建立就分配 64k 的读缓冲的话，将占用 640M 内存，而大多数时候这些缓冲区的使用率很低。muduo 用 readv 结合栈上空间巧妙地解决了这个问题。

具体做法是，在栈上准备一个 65536 字节的 stackbuf，然后利用 readv() 来读取数据，iovec 有两块，第一块指向 muduo Buffer 中的 writable 字节，另一块指向栈上的 stackbuf。这样如果读入的数据不多，那么全部都读到 Buffer 中去了；如果长度超过 Buffer 的 writable 字节数，就会读到栈上的 stackbuf 里，然后程序再把 stackbuf 里的数据 append 到 Buffer 中。代码见第 **??** 节 [14]。

这么做利用了临时栈上空间，避免开巨大 Buffer 造成的内存浪费，也避免反复调用 read() 的系统开销（通常一次 readv() 系统调用就能读完全部数据）。

这算是一个小小的创新吧。

[14] http://code.google.com/p/muduo/source/browse/trunk/muduo/net/Buffer.cc#38

线程安全？ muduo::net::Buffer 不是线程安全的，这么做是有意的，原因如下：

- 对于 input buffer，onMessage() 回调始终发生在该 TcpConnection 所属的那个 IO 线程，应用程序应该在 onMessage() 完成对 input buffer 的操作，并且不要把 input buffer 暴露给其他线程。这样所有对 input buffer 的操作都在同一个线程，Buffer class 不必是线程安全的。
- 对于 output buffer，应用程序不会直接操作它，而是调用 TcpConnection::send() 来发送数据，后者是线程安全的。

如果 TcpConnection::send() 调用发生在该 TcpConnection 所属的那个 IO 线程，那么它会转而调用 TcpConnection::sendInLoop()，sendInLoop() 会在当前线程（也就是 IO 线程）操作 output buffer；如果 TcpConnection::send() 调用发生在别的线程，它不会在当前线程调用 sendInLoop()，而是通过 EventLoop::runInLoop() 把 sendInLoop() 函数调用转移到 IO 线程（听上去颇为神奇?），这样 sendInLoop() 还是会在 IO 线程操作 output buffer，不会有线程安全问题。当然，跨线程的函数转移调用涉及函数参数的跨线程传递，一种简单的做法是把数据拷一份，绝对安全（不明白的同学请阅读代码 [15]）。

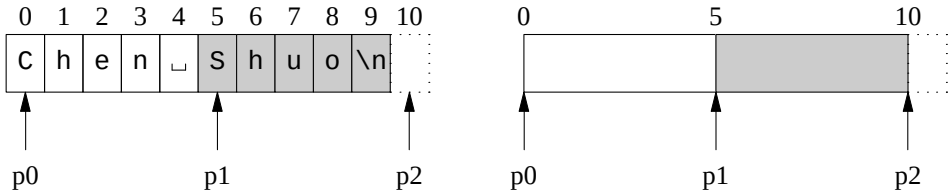
另一种更为高效做法是用 swap()。这就是为什么 TcpConnection::send() 的某个重载以 Buffer* 为参数，而不是 const Buffer&，这样可以避免拷贝，而用 Buffer::swap() 实现高效的线程间数据转移。（最后这点，仅为设想，暂未实现。目前仍然以数据拷贝方式在线程间传递，略微有些性能损失。）

2.4.4 Muduo Buffer 的数据结构

Buffer 的内部是一个 vector of char，它是一块连续的内存。此外，Buffer 有两个 data members，指向该 vector 中的元素。这两个 indices 的类型是 int，不是 char*，目的是应对迭代器失效。muduo Buffer 的设计参考了 Netty 的 ChannelBuffer 和 libevent 1.4.x 的 evbuffer。不过，其 prependable 可算是一点“微创新”。

在介绍 Buffer 的数据结构之前，先简单说一下图中表示指针或下标的箭头所指位置的具体含义。对于长度为 10 的字符串 "Chen Shuo\n"，如果 p0 指向第 0 个字符（白色区域的开始），p1 指向第 5 个字符（灰色区域的开始），p2 指向 '\n' 之后的那个位置（通常是 end() 迭代器所指的位置），那么精确的画法如下左图，简略的画法如下右图，后文都采用这种简略画法。

[15] http://code.google.com/p/muduo/source/browse/trunk/muduo/net/TcpConnection.cc#70



Muduo Buffer 的数据结构如下：

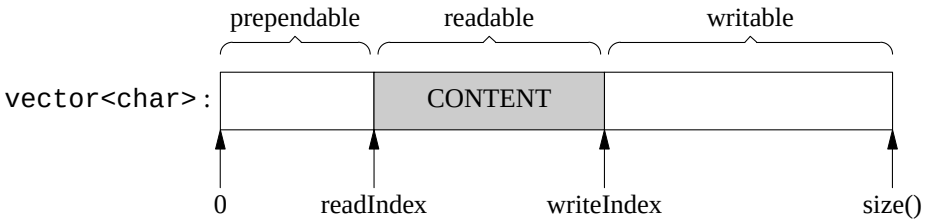


图 1

两个 indices 把 vector 的内容分为三块：prependable、readable、writable，各块的大小是（公式一）：

```
prependable = readIndex
readable = writeIndex - readIndex
writable = size() - writeIndex
```

灰色部分是 Buffer 的有效载荷 (payload)，prependable 的作用留到后面讨论。

readIndex 和 writeIndex 满足以下不变式 (invariant):

$$0 \le \text{readIndex} \le \text{writeIndex} \le \text{data.size()}$$

Muduo Buffer 里有两个常数 kCheapPrepend 和 kInitialSize，定义了 prependable 的初始大小和 writable 的初始大小，readable 的初始大小为 0。在初始化之后，Buffer 的数据结构如下：括号里的数字是该变量或常量的值。

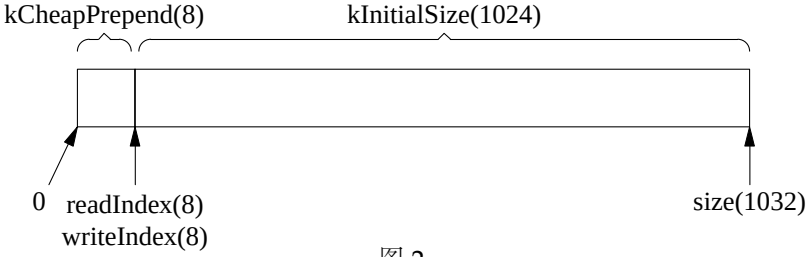


图 2

根据以上（公式一）可算出各块的大小，刚刚初始化的 Buffer 里没有 payload 数据，所以 readable == 0。

2.4.5 Muduo Buffer 的操作

1. 基本的 read-write cycle

Buffer 初始化后的情况见图 1，如果向 Buffer 写入了 200 字节，那么其布局是：

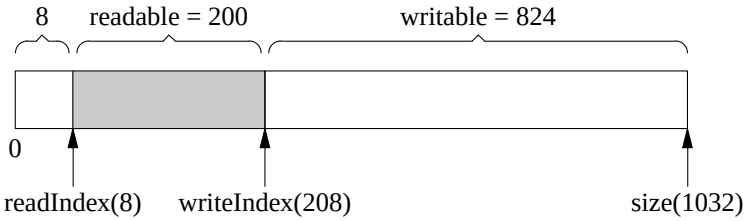


图 3

图 3 中 writeIndex 向后移动了 200 字节，readIndex 保持不变，readable 和 writable 的值也有变化。

如果从 Buffer read() & retrieve() （下称“读入”）了 50 字节，结果见图 4。与上图相比，readIndex 向后移动 50 字节，writeIndex 保持不变，readable 和 writable 的值也有变化（这句话往后从略）。

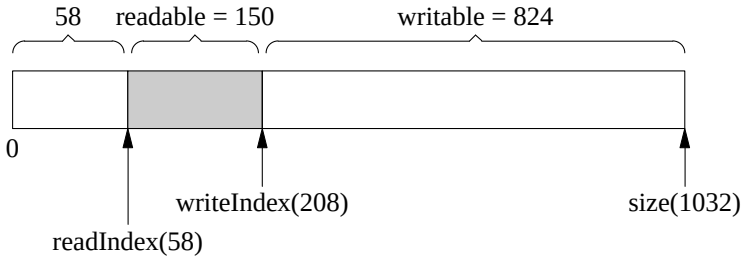


图 4

然后又写入了 200 字节，writeIndex 向后移动了 200 字节，readIndex 保持不变，见图 5。

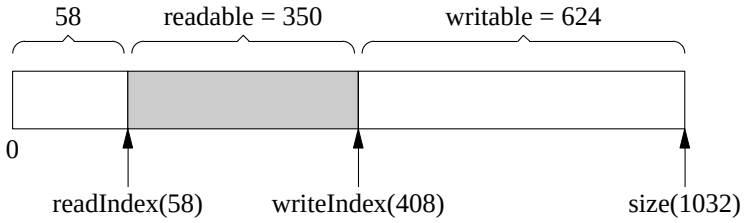


图 5

接下来，一次性读入 350 字节，请注意，由于全部数据读完了，readIndex 和 writeIndex 返回原位以备新一轮使用，见图 6，这和图 2 是一样的。

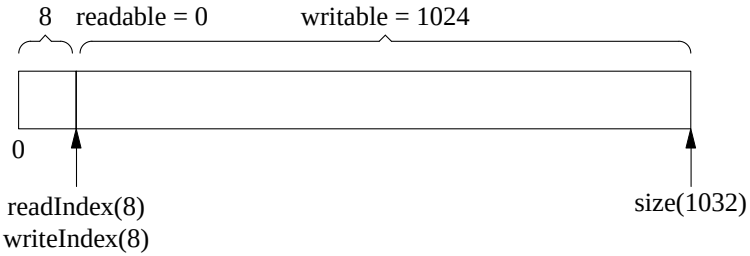


图 6

以上过程可以看作是发送方发送了两条消息，长度分别为 50 字节和 350 字节，接收方分两次收到数据，每次 200 字节，然后进行分包，再分两次回调客户代码。

2. 自动增长

Muduo Buffer 不是固定长度的，它可以自动增长，这是使用 vector 的直接好处。假设当前的状态如图 7 所示。(这和上面图 5 是一样的。)

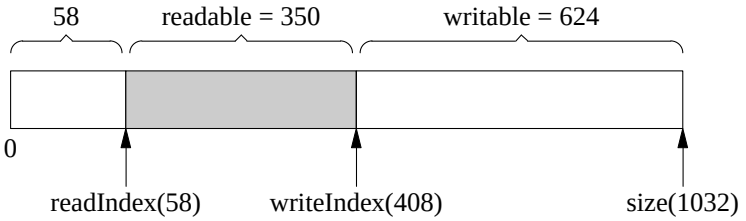


图 7

客户代码一次性写入 1000 字节，而当前可写的字节数只有 624，那么 buffer 会自动增长以容纳全部数据，得到的结果是图 8。注意 readIndex 返回到了前面，以保持 prependable 等于 kCheapPrependable。由于 vector 重新分配了内存，原来指向它元素的指针会失效，这就是为什么 readIndex 和 writeIndex 是整数下标而不是指针。

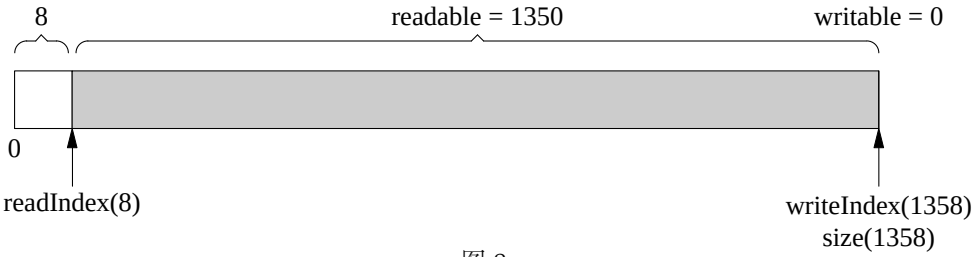


图 8

然后读入 350 字节，readIndex 前移，见图 9。

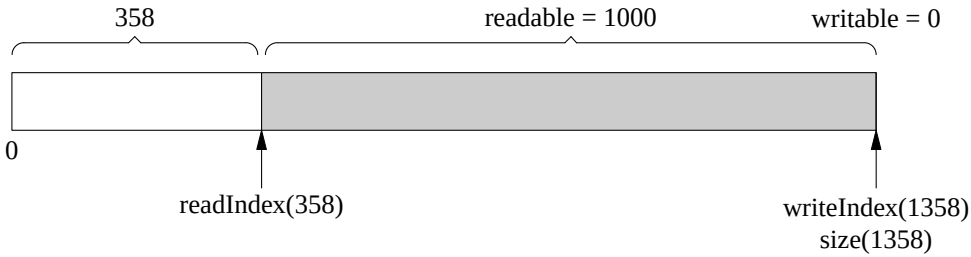


图 9

最后，读完剩下的 1000 字节，readIndex 和 writeIndex 返回 kCheapPrependable，见图 10。

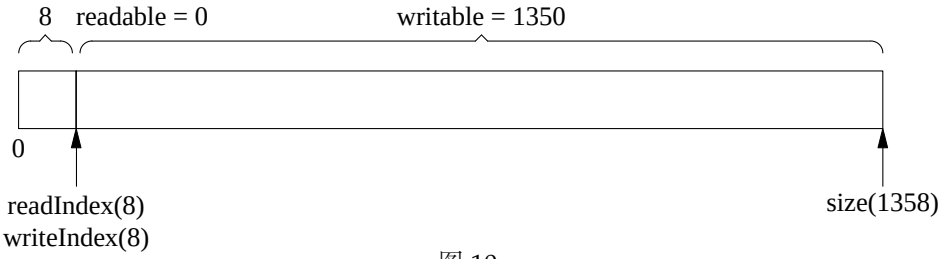


图 10

注意 buffer 并没有缩小大小，下次写入 1350 字节就不会重新分配内存了。换句话说，Muduo Buffer 的 size() 是自适应的，它一开始的初始值是 1k，如果程序里边经常收发 10k 的数据，那么用几次之后它的 size() 会自动增长到 10k，然后就保持不变。这样一方面避免浪费内存（Buffer 的初始大小直接决定了高并发连接时的内存消耗），另一方面避免反复分配内存。当然，客户代码可以手动 shrink() buffer size()。

3. size() 与 capacity()

使用 vector 的另一个好处是它的 capcity() 机制减少了内存分配的次数。比方说程序反复写入 1 字节，muduo Buffer 不会每次都分配内存，vector 的 capacity() 以指数方式增长，让 push_back() 的平均复杂度是常数。比方说经过第一次增长，size() 刚好满足写入的需求，如图 11。但这个时候 vector 的 capacity() 已经大于 size()，在接下来写入 capacity()-size() 字节的数据时，都不会重新分配内存，见图 12。

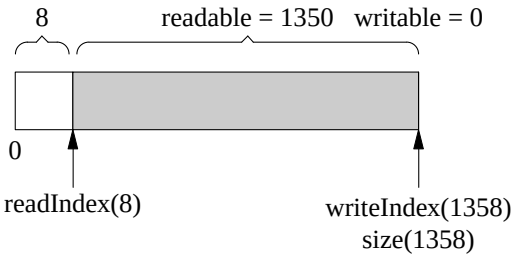


图 11

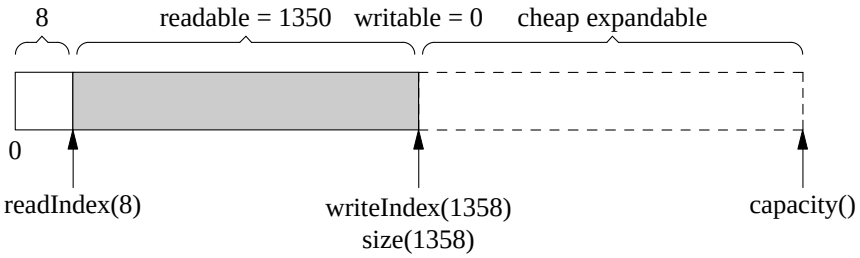


图 12

思考题：为什么我们不需要调用 reserve() 来预先分配空间？因为 Buffer 在构造函数里把初始 size() 设为 1k，这样当 size() 超过 1k 的时候 vector 会把 capacity() 加倍，等于说 resize() 替我们做了 reserve() 的事。用一段简单的代码验证一下：

```
vector<char> vec;
printf("%zd %zd\n", vec.size(), vec.capacity());
vec.resize(1024);
printf("%zd %zd\n", vec.size(), vec.capacity());
vec.resize(1300);
printf("%zd %zd\n", vec.size(), vec.capacity());
```

运行结果：

```
0 0         # 一开始 size() 和 capacity() 都是 0
1024 1024   # resize(1024) 之后 size() 和 capacity() 都是 1024
1300 2048   # resize(稍大) 之后 capacity() 翻倍，相当于 reserve(2048)
```

细心的读者可能会发现用 capacity() 也不是完美的，它有优化的余地。具体来说，vector::resize() 会初始化 (memset/bzero) 内存，而我们不需要它初始化，因为反正立刻就要填入数据。比如，在图 12 的基础上写入 200 字节，由于 capacity() 足够大，不会重新分配内存，这是好事；但是 vector::resize() 会先把那 200 字节 memset 为 0 （图 13a），然后 muduo Buffer 再填入数据（图 13b）。这么做稍微有点浪费，不过我不打算优化它，除非它确实造成了性能瓶颈。（精通 STL 的读者可能会说用 vec.insert(vec.end(), ...) 以避免浪费，但是 writeIndex 和 size() 不一定是对齐的，会有别的麻烦。）

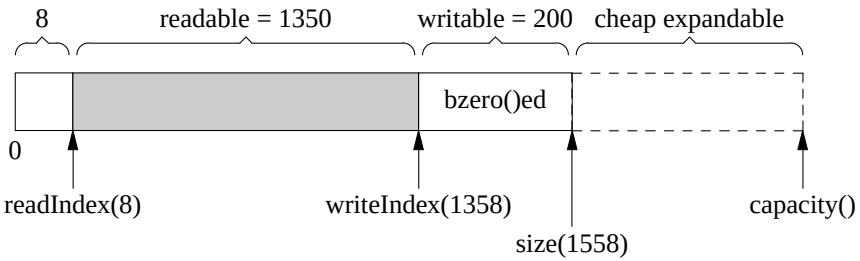


图 13a

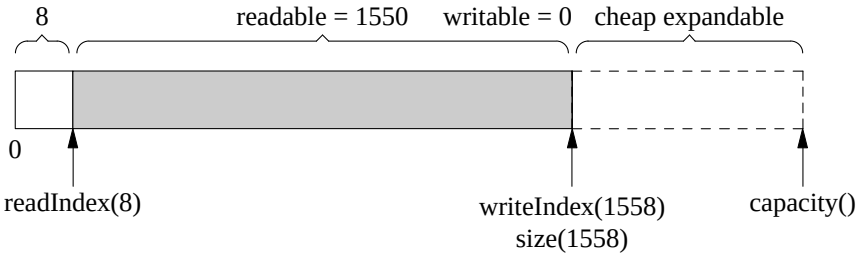


图 13b

google protobuf 中有一个 STLStringResizeUninitialized 函数 [16]，干的就是这个事情。

4. 内部腾挪

有时候，经过若干次读写，readIndex 移到了比较靠后的位置，留下了巨大的 prependable 空间，见图 14。

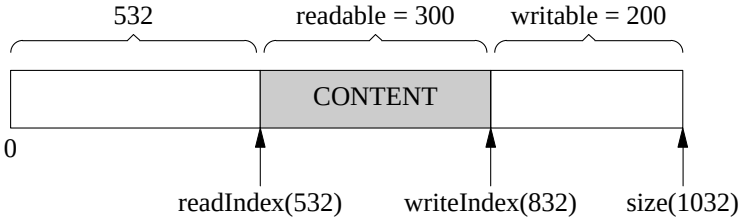


图 14

这时候，如果我们想写入 300 字节，而 writable 只有 200 字节，怎么办？muduo Buffer 在这种情况下不会重新分配内存，而是先把已有的数据移到前面去，腾出 writable 空间，见图 15。

[16] http://code.google.com/p/protobuf/source/browse/tags/2.4.0a/src/google/protobuf/stubs/stl_util-inl.h#60

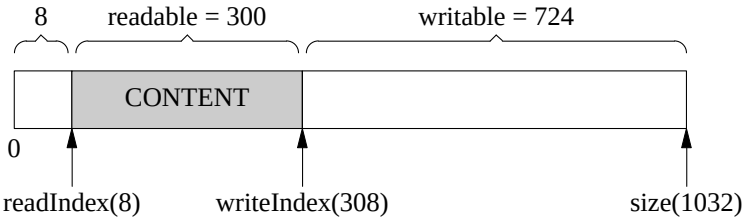


图 15

然后，就可以写入 300 字节了，见图 16。

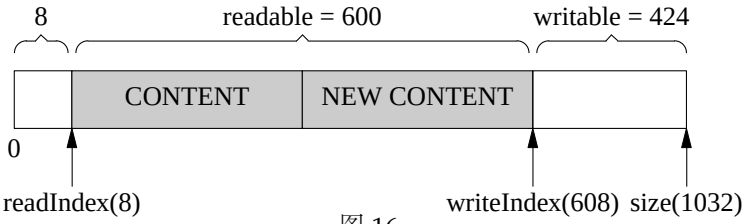


图 16

这么做的原因是，如果重新分配内存，反正也是要把数据拷到新分配的内存区域，代价只会更大。

5. prepend

前面说 muduo Buffer 有个小小的创新（或许不是创新，我记得在哪儿看到过类似的做法，忘了出处），即提供 prependable 空间，让程序能以很低的代价在数据前面添加几个字节。

比方说，程序以固定的 4 个字节表示消息的长度（第2.3节“Boost.Asio 的聊天服务器”中的 LengthHeaderCodec），我要序列化一个消息，但是不知道它有多长，那么我可以一直 append() 直到序列化完成（图 17，写入了 200 字节），然后再在序列化数据的前面添加消息的长度（图 18，把 200 这个数 prepend 到首部）。

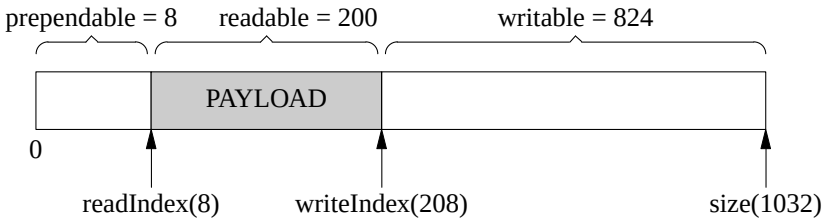


图 17

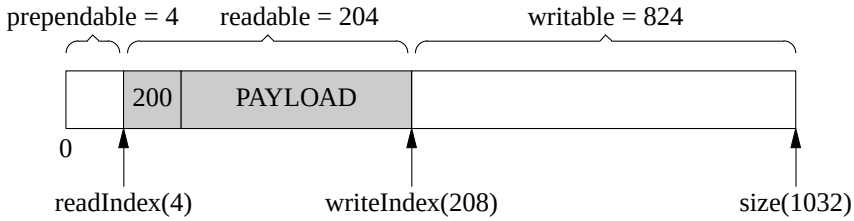


图 18

通过预留 kCheapPrependable 空间，可以简化客户代码，一个简单的空间换时间思路。

2.4.6 其他设计方案

这里简单谈谈其他可能的应用层 buffer 设计方案。

不用 vector<char>?

如果有 STL 洁癖，那么可以自己管理内存，以 4 个指针为 buffer 的成员，数据结构见图 19。

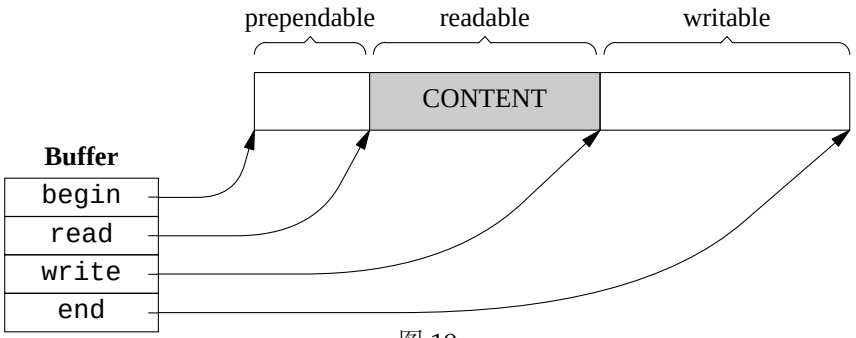


图 19

说实话我不觉得这种方案比 vector 好。代码变复杂，性能也未见得有 noticeable 的改观。如果放弃“连续性”要求，可以用 circular buffer，这样可以减少一点内存拷贝（没有“内部腾挪”）。

Zero copy ?

如果对性能有极高的要求，受不了 copy() 与 resize()，那么可以考虑实现分段连续的 zero copy buffer 再配合 gather scatter IO，数据结构如图 20，这是 libevent 2.0.x

的设计方案。TCPv2 介绍的 BSD TCP/IP 实现中的 mbuf 也是类似的方案，Linux 的 sk_buff 估计也差不多。细节有出入，但基本思路都是不要求数据在内存中连续，而是用链表把数据块链接到一起。

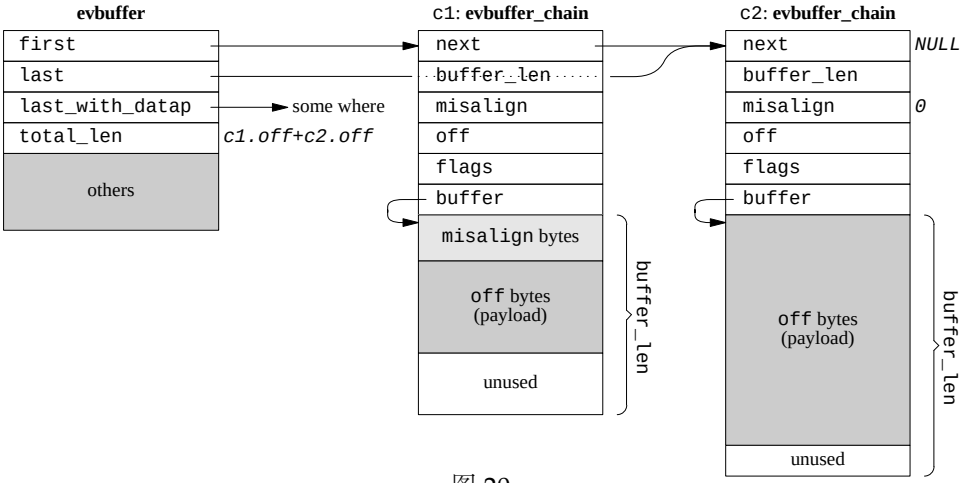


图 20

图 20 绘制的是由两个 evbuffer_chain 构成的 evbuffer，右边两个 evbuffer_chain 结构体中深灰色部分是 payload，可见 evbuffer 的缓冲区不是连续的，而是分块的。

当然，高性能的代价是代码变得晦涩难读，buffer 不再是连续的，parse 消息会稍微麻烦。如果你的程序只处理 protobuf Message，这不是问题，因为 protobuf 有 ZeroCopyInputStream 接口，只要实现这个接口，parsing 的事情就交给 protobuf Message 去操心了。

2.4.7 性能是不是问题？看跟谁比

看到这里，有的读者可能会嘀咕，muduo Buffer 有那么多可以优化的地方，其性能会不会太低？对此，我的回应是“可以优化，不一定值得优化。”

Muduo 的设计目标是用于开发公司内部的分布式程序。换句话说，它是用来写专用的 Sudoku server 或者游戏服务器，不是用来写通用的 httpd 或 ftpd 或 www proxy。前者通常有业务逻辑，后者更强调高并发与高吞吐。

以 Sudoku 为例，假设求解一个 Sudoku 问题需要 0.2ms，服务器有 8 个核，那么理想情况下每秒最多能求解 40,000 个问题。每次 Sudoku 请求的数据大小低于 100 字节（一个 9x9 的数独只要 81 字节，加上 header 也可以控制在 100 bytes 以下），就

是说 100 x 40000 = 4 MB per second 的吞吐量就足以让服务器的 CPU 饱和。在这种情况下，去优化 Buffer 的内存拷贝次数似乎没有意义。

再举一个例子，目前最常用的千兆以太网的裸吞吐量是 125MB/s，扣除以太网 header、IP header、TCP header 之后，应用层的吞吐率大约在 115 MB/s 上下。而现在服务器上最常用的 DDR2/DDR3 内存的带宽至少是 4GB/s，比千兆以太网高 40 倍以上。就是说，对于几 k 或几十 k 大小的数据，在内存里边拷几次根本不是问题，因为受以太网延迟和带宽的限制，跟这个程序通信的其他机器上的程序不会觉察到性能差异。

最后举一个例子，如果你实现的服务程序要跟数据库打交道，那么瓶颈常常在 DB 上，优化服务程序本身不见得能提高性能（从 DB 读一次数据往往就抵消了你做的全部 low-level 优化），这时不如把精力投入在 DB 调优上。

专用服务程序与通用服务程序的另外一点区别是 benchmark 的对象不同。如果你打算写一个 httpd，自然有人会拿来和目前最好的 nginx 对比，立马就能比出性能高低。然而，如果你写一个实现公司内部业务的服务程序（比如分布式存储或者搜索或者微博或者短网址），由于市面上没有同等功能的开源实现，你不需要在优化上投入全部精力，只要一版做得比一版好就行。先正确实现所需的功能，投入生产应用，然后再根据真实的负载情况来做优化，这恐怕比在编码阶段就盲目调优要更 effective 一些。

Muduo 的设计目标之一是吞吐量能让千兆以太网饱和，也就是每秒收发 120 兆字节的数据。这个很容易就达到，不用任何特别的努力。

如果确实在内存带宽方面遇到问题，说明你做的应用实在太 critical，或许应该考虑放到 Linux kernel 里边去，而不是在用户态尝试各种优化。毕竟只有把程序做到 kernel 里才能真正实现 zero copy，否则，核心态和用户态之间始终是有一次内存拷贝的。如果放到 kernel 里还不能满足需求，那么要么自己写新的 kernel，或者直接用 FPGA 或 ASIC 操作 network adapter 来实现你的高性能服务器。

2.5 一种自动反射消息类型的 Protobuf 网络传输方案

本文假定读者了解 Google Protocol Buffers 是什么，这不是一篇 Protobuf 入门教程。本文的示例代码位于 examples/protobuf/codec。

这篇文章要解决的问题是：通信双方在已知 proto 文件的情况下，接收方在收到 Protobuf 数据之后如何自动创建具体的 Protobuf Message 对象，再做的反序列化。

“自动”的意思是：当程序中新增一个 Protobuf Message 类型时，这部分代码不需要修改，不需要自己去注册消息类型。其实，Google Protobuf 本身具有很强的反射 (reflection) 功能，可以根据 type name 创建具体类型的 Message 对象，我们直接利用即可。

2.5.1 网络编程中使用 protobuf 的两个问题

Google Protocol Buffers (Protobuf) 是一款非常优秀的库，它定义了一种紧凑的可扩展二进制消息格式，特别适合网络数据传输。它为多种语言提供 binding，大大方便了分布式程序的开发，让系统不再局限于用某一种语言来编写。

在网络编程中使用 protobuf 需要解决两个问题[17]：

1. 长度，protobuf 打包的数据没有自带长度信息或终结符，需要由应用程序自己在发生和接收的时候做正确的切分；
2. 类型，protobuf 打包的数据没有自带类型信息，需要由发送方把类型信息传给给接收方，接收方创建具体的 Protobuf Message 对象，再做的反序列化。

第一个很好解决，通常的做法是在每个消息前面加个固定长度的 length header，例如第 2.3 节中实现的 LengthHeaderCodec。

第二个问题其实也很好解决，Protobuf 对此有内建的支持。但是奇怪的是，从网上简单搜索的情况看，我发现了很多山寨的做法。

2.5.2 山寨做法

以下均为在 protobuf data 之前加上 header，header 中包含 int length 和类型信息。类型信息的山寨做法主要有两种：

- 在 header 中放 int typeId，接收方用 switch-case 来选择对应的消息类型和处理函数；
- 在 header 中放 string typeName，接收方用 look-up table 来选择对应的消息类型和处理函数。

这两种做法都有问题。

第一种做法要求保持 typeId 的唯一性，它和 protobuf message type 一一对应。如果 protobuf message 的使用范围不广，比如接收方和发送方都是自己维护的程序，

[17] Protobuf 这么设计的原因见下一节。

那么 typeId 的唯一性不难保证，用版本管理工具即可。如果 protobuf message 的使用范围很大，比如全公司都在用，而且不同部门开发的分布式程序可能相互通信，那么就需要一个公司内部的全局机构来分配 typeId，每次增加新 message type 都要去注册一下，比较麻烦。

第二种做法稍好一点。typeName 的唯一性比较好办，因为可以加上 package name（也就是用 message 的 fully qualified type name），各个部门事先分好 namespace，不会冲突与重复。但是每次新增消息类型的时候都要去手工修改 look-up table 的初始化代码，也比较麻烦。

其实，不需要自己重新发明轮子，protobuf 本身已经自带了解决方案。

2.5.3 根据 type name 反射自动创建 Message 对象

Google Protobuf 本身具有很强的反射 (reflection) 功能，可以根据 type name 创建具体类型的 Message 对象。但是奇怪的是，其官方教程里没有明确提及这个用法，我估计还有很多人不知道这个用法，所以觉得值得写这篇文章谈一谈。

以下是陈硕绘制的 Protobuf class diagram（见下一页）。

我估计大家通常关心和使用的是图的左半部分：MessageLite、Message、Generated Message Types (Person, AddressBook) 等，而较少注意到图的右半部分：Descriptor, DescriptorPool, MessageFactory。

下图中，其关键作用的是 Descriptor class，每个具体 Message Type 对应一个 Descriptor 对象。尽管我们没有直接调用它的函数，但是 Descriptor 在“根据 type name 创建具体类型的 Message 对象”中扮演了重要的角色，起了桥梁作用。下图的红色箭头描述了根据 type name 创建具体 Message 对象的过程，后文会详细介绍。

原理简述

Protobuf Message class 采用了 prototype pattern [18]，Message class 定义了 New() 虚函数，用以返回本对象的一份新实例，类型与本对象的真实类型相同。也就是说，拿到 Message* 指针，不用知道它的具体类型，就能创建和它类型一样的具体 Message Type 的对象。

[18] http://en.wikipedia.org/wiki/Prototype_pattern

每个具体 Message Type 都有一个 default instance，可以通过 ConcreteMessage::default_instance() 获得，也可以通过 MessageFactory::GetPrototype(const Descriptor*) 来获得。所以，现在问题转变为 1. 如何拿到 MessageFactory；2. 如何拿到 Descriptor*。

当然，ConcreteMessage::descriptor() 返回了我们想要的 Descriptor*，但是，在不知道 ConcreteMessage 的时候，如何调用它的静态成员函数呢？这似乎是个鸡与蛋的问题。

我们的英雄是 DescriptorPool，它可以根据 type name 查到 Descriptor*，只要找到合适的 DescriptorPool，再调用 DescriptorPool::FindMessageTypeByName(const string& type_name) 即可。看到下页的图是不是眼前一亮？

在最终解决问题之前，先简单测试一下，看看我上面说的对不对。

验证思路

本文用于举例的 proto 文件：

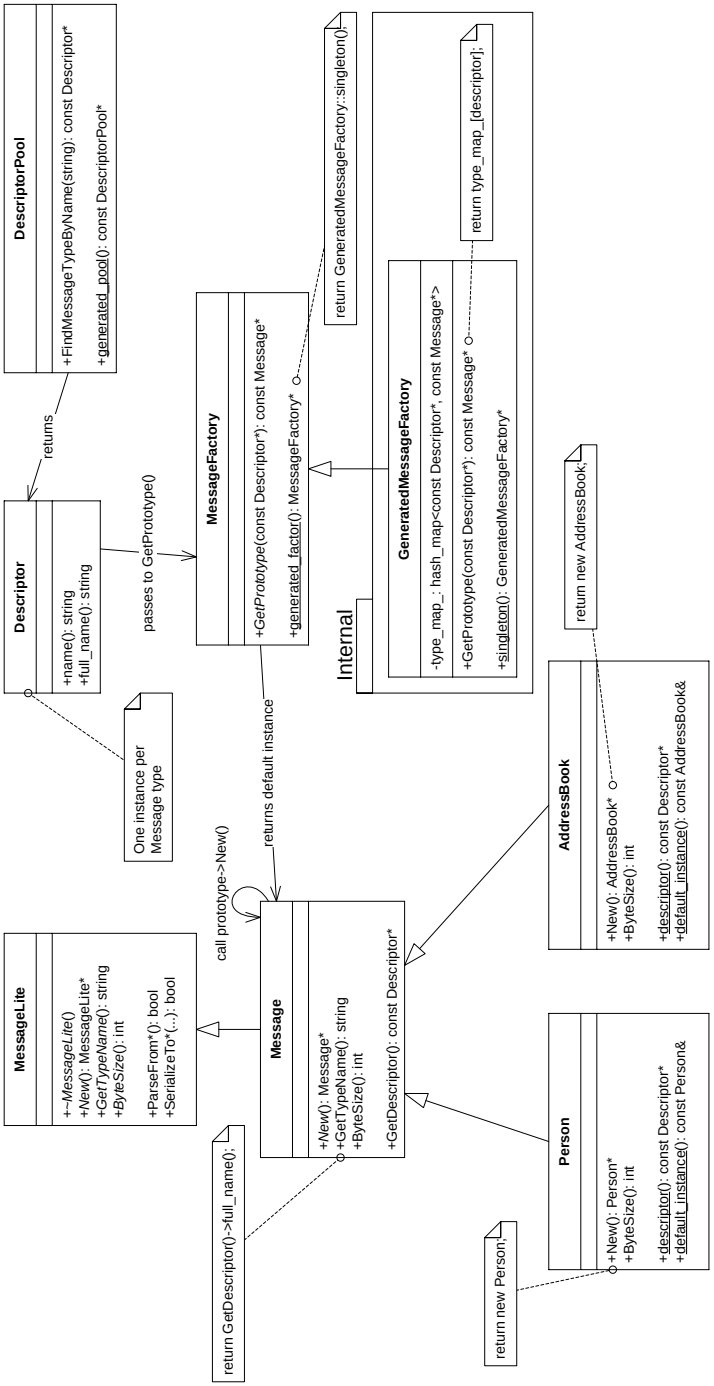
examples/protobuf/codec/query.proto

```
package muduo;

message Query {
  required int64 id = 1;
  required string questioner = 2;

  repeated string question = 3;
}

message Answer {
  required int64 id = 1;
  required string questioner = 2;
  required string answerer = 3;

  repeated string solution = 4;
}

message Empty {
  optional int32 id = 1;
}
```

examples/protobuf/codec/query.proto

其中的 Query.questioner 和 Answer.answerer 是我在前一篇文章这提到的《分布式系统中的进程标识》[19]。

[19]http://blog.csdn.net/Solstice/article/details/6285216


DescriptorPool
+FindMessageTypeByName(string): const Descriptor*
+generated_pool(): const DescriptorPool*
returns
Descriptor
+name(): string
+full_name(): string
passes to GetPrototype()
MessageFactory
+GetPrototype(const Descriptor*): const Message*
+generated_factor(): MessageFactory*
return GeneratedMessageFactory::singleton();
Internal
GeneratedMessageFactory
-type_map_: hash_map<const Descriptor*, const Message*>
+GetPrototype(const Descriptor*): const Message*
+singleton(): GeneratedMessageFactory*
return type_map_[descriptor];
One instance per Message type
returns default instance
call prototype->New()
MessageLite
+~MessageLite()
+New(): MessageLite*
+GetTypeName(): string
+ByteSize(): int
+ParseFrom*(): bool
+SerializeTo*(...): bool
Message
+New(): Message*
+GetTypeName(): string
+ByteSize(): int
+GetDescriptor(): const Descriptor*
AddressBook
+New(): AddressBook*
+ByteSize(): int
+descriptor(): const Descriptor*
+default_instance(): const AddressBook&
return new AddressBook;
return GetDescriptor()->full_name();
Person
+New(): Person*
+ByteSize(): int
+descriptor(): const Descriptor*
+default_instance(): const Person&
return new Person;

以下代码[20] 验证 ConcreteMessage::default_instance()、ConcreteMessage::descriptor()、MessageFactory::GetPrototype()、DescriptorPool::FindMessageTypeByName() 之间的不变式 (invariant)，注意其中的 asserts：

```
typedef muduo::Query T;

std::string type_name = T::descriptor()->full_name();
cout << type_name << endl;

const Descriptor* descriptor
    = DescriptorPool::generated_pool()->FindMessageTypeByName(type_name);
assert(descriptor == T::descriptor());
cout << "FindMessageTypeByName() = " << descriptor << endl;
cout << "T::descriptor()         = " << T::descriptor() << endl;
cout << endl;

const Message* prototype
    = MessageFactory::generated_factory()->GetPrototype(descriptor);
assert(prototype == &T::default_instance());
cout << "GetPrototype()        = " << prototype << endl;
cout << "T::default_instance() = " << &T::default_instance() << endl;
cout << endl;

T* new_obj = dynamic_cast<T*>(prototype->New());
assert(new_obj != NULL);
assert(new_obj != prototype);
assert(typeid(*new_obj) == typeid(T::default_instance()));
cout << "prototype->New() = " << new_obj << endl;
cout << endl;
delete new_obj;
```

程序运行结果如下

```
muduo.Query
FindMessageTypeByName() = 0xd4e720
T::descriptor()         = 0xd4e720

GetPrototype()        = 0xd47710
T::default_instance() = 0xd47710

prototype->New() = 0xd459e0
```

根据 type name 自动创建 Message 的关键代码

好了，万事具备，开始行动：

1. 用 DescriptorPool::generated_pool() 找到一个 DescriptorPool 对象，它包含了程序编译的时候所链接的全部 protobuf Message types。

[20] https://github.com/chenshuo/recipes/blob/master/protobuf/descriptor_test.cc

2. 用 DescriptorPool::FindMessageTypeByName() 根据 type name 查找 Descriptor。
3. 再用 MessageFactory::generated_factory() 找到 MessageFactory 对象，它能创建程序编译的时候所链接的全部 protobuf Message types。
4. 然后，用 MessageFactory::GetPrototype() 找到具体 Message Type 的 default instance。
5. 最后，用 prototype->New() 创建对象。

示例代码如下。

examples/protobuf/codec/codec.cc

```
147 Message* createMessage(const std::string& typeName)
148 {
149   Message* message = NULL;
150   const Descriptor* descriptor
151       = DescriptorPool::generated_pool()->FindMessageTypeByName(typeName);
152   if (descriptor)
153   {
154     const Message* prototype
155         = MessageFactory::generated_factory()->GetPrototype(descriptor);
156     if (prototype)
157     {
158       message = prototype->New();
159     }
160   }
161   return message;
162 }
```

examples/protobuf/codec/codec.cc

调用方式：

```
Message* newQuery = createMessage("muduo.Query");
assert(newQuery != NULL);
assert(typeid(*newQuery) == typeid(muduo::Query::default_instance()));
cout << "createMessage(\"muduo.Query\") = " << newQuery << endl;
```

确实能从消息名称创建消息对象，古之人不余欺也:-)

注意，createMessage() 返回的是动态创建的对象的指针，调用方有责任释放它，不然就会内存泄露。在 muduo 里，我用 shared_ptr<Message> 来自动管理 Message 对象的生命期。

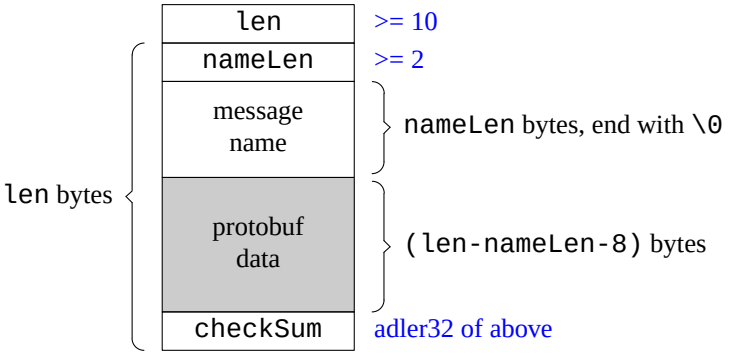
拿到 Message* 之后怎么办呢？怎么调用这个具体消息类型的处理函数？这就需要消息分发器 (dispatcher) 出马了，且听下回分解。

线程安全性

Google 的文档说，我们用到的那几个 MessageFactory 和 DescriptorPool 都是线程安全的，Message::New() 也是线程安全的。并且它们都是 const member function。关键问题解决了，那么剩下工作就是设计一种包含长度和消息类型的 protobuf 传输格式。

2.5.4 Protobuf 传输格式

陈硕设计了一个简单的格式，包含 protobuf data 和它对应的长度与类型信息，消息的末尾还有一个 check sum。格式如下图，图中方块的宽度是 32-bit。



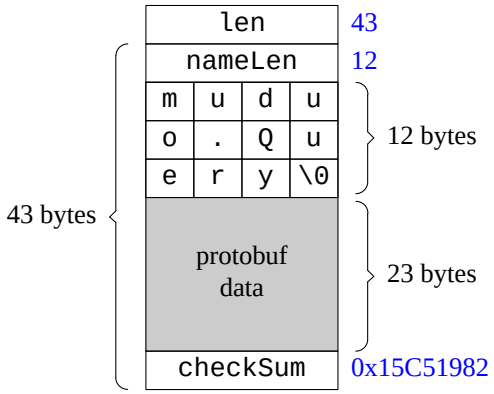
用 C struct 伪代码描述：

```
struct ProtobufTransportFormat __attribute__ ((__packed__))
{
  int32_t  len;
  int32_t  nameLen;
  char     typeName[nameLen];
  char     protobufData[len-nameLen-8];
  int32_t  checkSum; // adler32 of nameLen, typeName and protobufData
};
```

注意，这个格式不要求 32-bit 对齐，我们的 decoder 会自动处理非对齐的消息。

例子

用这个格式打包一个 muduo.Query 对象的结果是：



设计决策

以下是我在设计这个传输格式时的考虑：

- signed int。消息中的长度字段只使用了 signed 32-bit int，而没有使用 unsigned int，这是为了移植性，因为 Java 语言没有 unsigned 类型。另外 Protobuf 一般用于打包小于 1M 的数据，unsigned int 也没用。
- check sum。虽然 TCP 是可靠传输协议，虽然 Ethernet 有 CRC-32 校验，但是网络传输必须要考虑数据损坏的情况，对于关键的网络应用，check sum 是必不可少的。对于 protobuf 这种紧凑的二进制格式而言，肉眼看不出数据有没有问题，需要用 check sum。
- adler32 算法。我没有选用常见的 CRC-32，而是选用 adler32，因为它计算量小、速度比较快，强度和 CRC-32 差不多。另外，zlib 和 java.unit.zip 都直接支持这个算法，不用我们自己实现。
- type name 以 '\0' 结束。这是为了方便 troubleshooting，比如通过 tcpdump 抓下来的包可以用肉眼很容易看出 type name，而不用根据 nameLen 去一个个数字节。同时，为了方便接收方处理，加入了 nameLen，节省 strlen()，空间换时间。
- 没有版本号。Protobuf Message 的一个突出优点是用 optional fields 来避免协议的版本号（凡是在 protobuf Message 里放版本号的人都没有理解 protobuf 的设计，甚至可能没有仔细阅读 protobuf 的文档 [21] [22] [23]），让通信双方的程序

[21] http://code.google.com/apis/protocolbuffers/docs/overview.html "A bit of history"
[22] http://code.google.com/apis/protocolbuffers/docs/proto.html#updating
[23] http://code.google.com/apis/protocolbuffers/docs/cpptutorial.html "Extending a Protocol Buffer"

能各自升级，便于系统演化。如果我设计的这个传输格式又把版本号加进去，那就画蛇添足了。具体请见本人《分布式系统的工程化开发方法》[24]第 57 页：消息格式的选择。

Protobuf 可谓是网络协议格式的典范，值得我单独花一篇文章讲述其思想。

2.6 在 muduo 中实现 Protobuf 编解码器与消息分发器

本文是前一节的自然延续，介绍如何将前文介绍的打包方案与 muduo::net::Buffer 结合，实现了 protobuf codec 和 dispatcher。

在介绍 codec 和 dispatcher 之前，先讲讲前文的一个未决问题。

为什么 Protobuf 的默认序列化格式没有包含消息的长度与类型？

Protobuf 是经过深思熟虑的消息打包方案，它的默认序列化格式没有包含消息的长度与类型，自然有其道理。哪些情况下不需要在 protobuf 序列化得到的字节流中包含消息的长度和（或）类型？我能想到的答案有：

- 如果把消息写入文件，一个文件存一个消息，那么序列化结果中不需要包含长度和类型，因为从文件名和文件长度中可以得知消息的类型与长度。
- 如果把消息写入文件，一个文件存多个消息，那么序列化结果中不需要包含类型，因为文件名就代表了消息的类型。
- 如果把消息存入数据库（或者 NoSQL），以 VARBINARY 字段保存，那么序列化结果中不需要包含长度和类型，因为从字段名和字段长度中可以得知消息的类型与长度。
- 如果把消息以 UDP 方式发生给对方，而且对方一个 UDP port 只接收一种消息类型，那么序列化结果中不需要包含长度和类型，因为从 port 和 UDP packet 长度中可以得知消息的类型与长度。
- 如果把消息以 TCP 短连接方式发给对方，而且对方一个 TCP port 只接收一种消息类型，那么序列化结果中不需要包含长度和类型，因为从 port 和 TCP 字节流长度中可以得知消息的类型与长度。
- 如果把消息以 TCP 长连接方式发给对方，但是对方一个 TCP port 只接收一种消息类型，那么序列化结果中不需要包含类型，因为 port 代表了消息的类型。

[24]http://blog.csdn.net/Solstice/article/details/5950190

- 如果采用 RPC 方式通信，那么只需要告诉对方 method name，对方自然能推断出 Request 和 Response 的消息类型，这些可以由 protoc 生成的 RPC stubs 自动搞定。

对于以上最后一点，比方说 sudoku.proto 定义为：

```
service SudokuService {
  rpc Solve (SudokuRequest) returns (SudokuResponse);
}
```

那么 RPC method Sudoku.Solve 对应的请求和响应分别是 SudokuRequest 和 SudokuResponse。在发送 RPC 请求的时候，不需要包含 SudokuRequest 的类型，只需要发送 method name Sudoku.Solve，对方自知道应该按照 SudokuRequest 来解析 (parse) 请求。

对于上述这些情况，如果 protobuf 无条件地把长度和类型放到序列化的字节串中，只会浪费网络带宽和存储。可见 protobuf 默认不发送长度和类型是正确的决定。Protobuf 为消息格式的设计树立了典范，哪些该自己搞定，哪些留给外部系统去解决，这些都考虑得很清楚。

只有在使用 TCP 长连接，且在一个连接上传递不止一种消息的情况下（比方同时发 Heartbeat 和 Request/Response），才需要我前文提到的那种打包方案 [25]。这时候我们需要一个分发器 dispatcher，把不同类型的消息分给各个消息处理函数，这正是本文的主题之一。

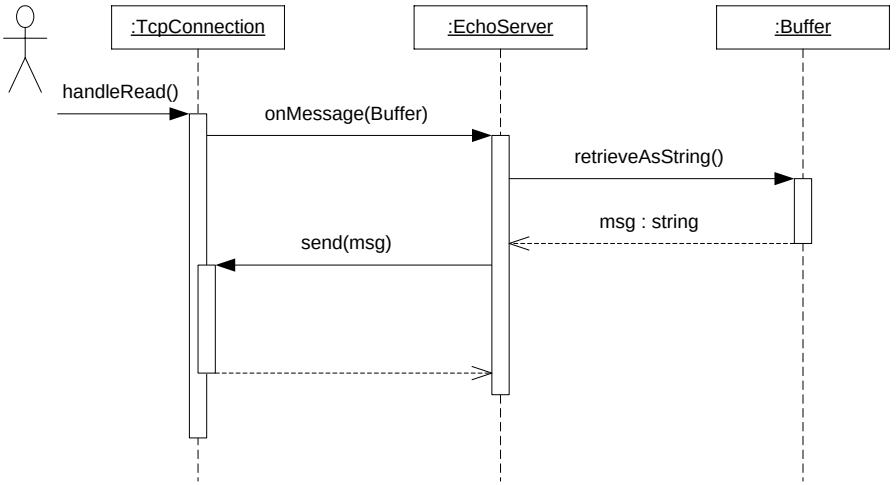
以下均只考虑 TCP 长连接这一应用场景。先谈谈编解码器。

2.6.1 什么是编解码器 codec?

Codec [26] 是 encoder 和 **dec**oder 的缩写，这是一个软硬件领域都在使用的术语，这里我借指“把网络数据和业务消息之间互相转换”的代码。

在最简单的网络编程中，没有消息 message 只有字节流数据，这时候是用不到 codec 的。比如我们前面讲过的 echo server，它只需要把收到的数据原封不动地发送回去，它不必关心消息的边界（也没有“消息”的概念），收多少就发多少，这种情况下它干脆直接使用 muduo::net::Buffer，取到数据再交给 TcpConnection 发送回去，见下图。

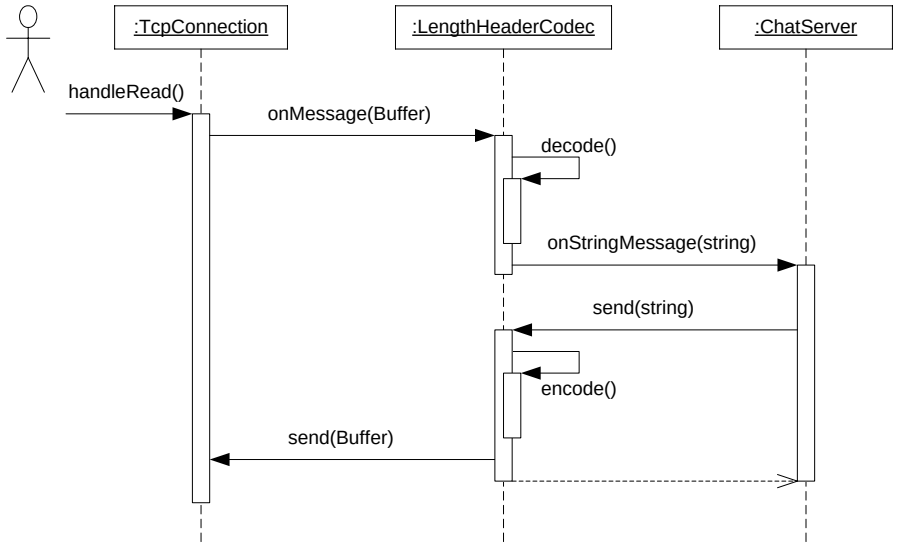
[25] 为什么要在一个连接上同时发 Heartbeat 和业务消息？见第 XX 节心跳协议的设计。
[26] http://en.wikipedia.org/wiki/Codec



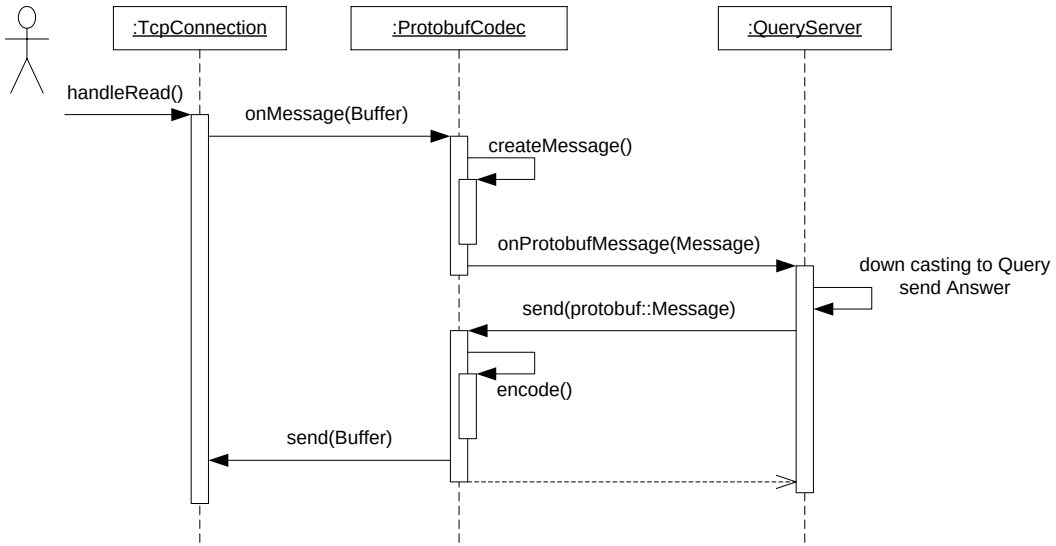
non-trivial 的网络服务程序通常会以消息为单位来通信，每条消息有明确的长度与界限。程序每次收到一个完整的消息的时候才开始处理，发送的时候也是把一个完整的消息交给网络库。比如我们前面讲过的 asio chat 服务，它的一条聊天记录就是一条消息。为此我们设计了一个简单的消息格式，即在聊天记录前面加上 4 字节的 length header，LengthHeaderCodec 代码及解说见第 2.3 节。

codec 的基本功能之一是做 TCP 分包：确定每条消息的长度，为消息划分界限。在 non-blocking 网络编程中，codec 几乎是必不可少的。如果只收到了半条消息，那么不会触发消息事件回调，数据会停留在 Buffer 里（数据已经读到 Buffer 中了），等待收到一个完整的消息再通知处理函数。既然这个任务太常见，我们干脆做一个 utility class，避免服务端和客户端程序都要自己处理分包，这就有了 LengthHeaderCodec。这个 codec 的使用有点奇怪，不需要继承，它也没有基类，只要把它当成普通 data member 来用，把 TcpConnection 的数据喂给它，然后向它注册 onXXXMessage() 回调，代码见 asio chat 示例。muduo 里的 codec 都是这样的风格：通过 boost::function 粘合到一起。

codec 是一层间接性，它位于 TcpConnection 和 ChatServer 之间，拦截处理收到的数据，在收到完整的消息之后再调用 CharServer 对应的处理函数，注意 CharServer::onStringMessage() 的参数是 std::string，不再是 muduo::net::Buffer，也就是说 LengthHeaderCodec 把 Buffer **解码**成了 string。另外，在发送消息的时候，ChatServer 通过 LengthHeaderCodec::send() 来发送 string，LengthHeaderCodec 负责把它**编码**成 Buffer。这正是“编解码器”名字的由来。



Protobuf codec 与此非常类似，只不过消息类型从 std::string 变成了 protobuf::Message。对于只接收处理 Query 消息的 QueryServer 来说，用 ProtobufCodec 非常方便，收到 protobuf::Message 之后 down cast 成 Query 来用就行。如果要接收处理不止一种消息，ProtobufCodec 恐怕还不能单独完成工作，请继续阅读下文。



2.6.2 实现 ProtobufCodec

Protobuf 的打包方案我已经在前一节中讲过。编码算法很直截了当，按照前文定义的消息格式一路打包下来，最后更新一下首部的长度即可。代码位于 examples/protobuf/codec/codec.cc 中的 ProtobufCodec::fillEmptyBuffer()。

解码算法有几个要点：

- protobuf::Message 是 new 出来的对象，它的生命期如何管理？muduo 采用 shared_ptr<Message> 来自动管理对象生命期，与整体风格保持一致的。
- 出错如何处理？比方说长度超出范围、check sum 不正确、message type name 不能识别、message parse 出错等等。ProtobufCodec 定义了 ErrorCallback，用户代码可以注册这个回调。如果不注册，默认的处理是断开连接，让客户重连重试。codec 的单元测试里模拟了各种出错情况。
- 如何处理一次收到半条消息、一条消息、一条半消息、两条消息等等情况？这是每个 non-blocking 网络程序中的 codec 都要面对的问题。在第68的代码中我们已经解决了这个问题。

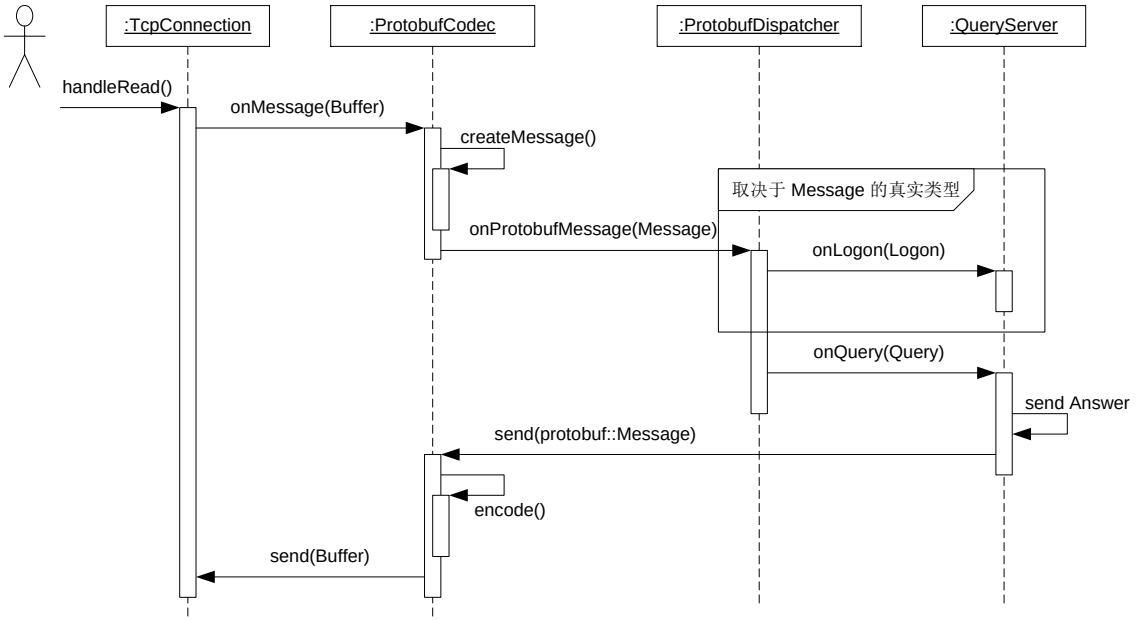
ProtobufCodec 在实际使用中有明显的不足：它只负责把 muduo::net::Buffer 转换为具体类型的 protobuf::Message，应用程序拿到 Message 之后还有再根据其具体类型做一次分发。我们可以考虑做一个简单通用的分发器 dispatcher，以简化客户代码。

此外，目前 ProtobufCodec 的实现非常初级，它没有充分利用 ZeroCopyInputStream 和 ZeroCopyOutputStream，而是把收到的数据作为 byte array 交给 protobuf Message 去解析，这给性能优化留下了空间。protobuf Message 不要求数据连续 (像 vector 那样)，只要求数据分段连续 (像 deque 那样)，这给 buffer 管理带来性能上的好处（避免重新分配内存，减少内存碎片），当然也使得代码变复杂。muduo::net::Buffer 非常简单，它内部是 vector<char>，我目前不想让 protobuf 影响 muduo 本身的设计，毕竟 muduo 是个通用的网络库，不是为实现 protobuf RPC 而特制的。

2.6.3 消息分发器 dispatcher 有什么用？

前面提到，在使用 TCP 长连接，且在一个连接上传递不止一种 protobuf 消息的情况下，客户代码需要对收到的消息按类型做分发。比方说，收到 Logon 消息就交给 QueryServer::onLogon() 去处理，收到 Query 消息就交给 QueryServer::onQuery() 去处理。这个消息分派机制可以做得稍微有点通用性，让所有 muduo+protobuf 程序收益，而且不增加复杂性。

换句话说，又是一层间接性，ProtobufCodec 拦截了 TcpConnection 的数据，把它转换为 Message，ProtobufDispatcher 拦截了 ProtobufCodec 的 callback，按消息具体类型把它分派给多个 callbacks。



2.6.4 ProtobufCodec 与 ProtobufDispatcher 的综合运用

我写了两个示例代码，client 和 server，把 ProtobufCodec 和 ProtobufDispatcher 串联起来使用。server 响应 Query 消息，发生回 Answer 消息，如果收到未知消息类型，则断开连接。client 可以选择发送 Query 或 Empty 消息，由命令行控制。这样可以测试 unknown message callback。

为节省篇幅，这里就不列出代码了，见 examples/protobuf/codec/{client, server}.cc。

在构造函数中，通过注册回调函数把四方 (TcpConnection、codec、dispatcher、QueryServer) 结合起来。

2.6.5 ProtobufDispatcher 的两种实现

要完成消息分发，那么就是对消息做 type-switch，这似乎是一个 bad smell，但是 protobuf Message 的 Descriptor 没有留下定制点（比如暴露一个 boost::any 成

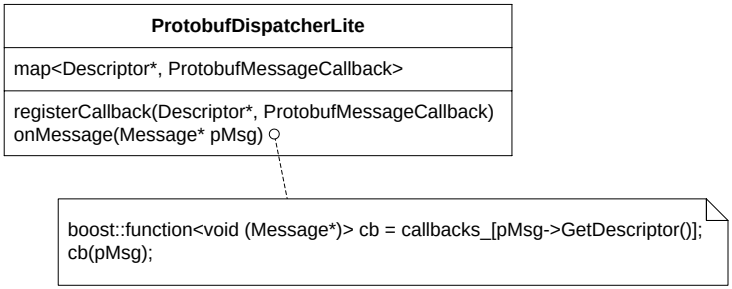
员），我们只好硬来了。

先定义

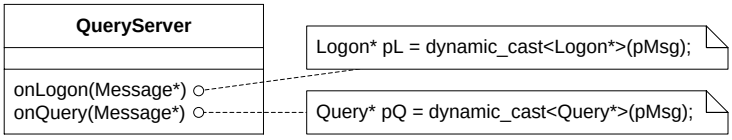
```
typedef boost::function<void (Message*)> ProtobufMessageCallback;
```

注意，本节出现的不是 muduo dispatcher 真实的代码，仅为示意，突出重点，便于画图。

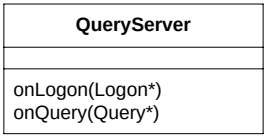
ProtobufDispatcherLite 的结构非常简单，它有一个 map<Descriptor*, ProtobufMessageCallback> 成员，客户代码可以以 Descriptor* 为 key 注册回调（recall: 每个具体消息类型都有一个全局的 Descriptor 对象，其地址是不变的，可以用来当 key）。在收到 protobuf Message 之后，在 map 中找到对应的 ProtobufMessageCallback，然后调用之。如果找不到，就调用 defaultCallback。



当然，它的设计也有小小的缺陷，那就是 ProtobufMessageCallback 限制了客户代码只能接受基类 Message，客户代码需要自己做向下转型，比如：

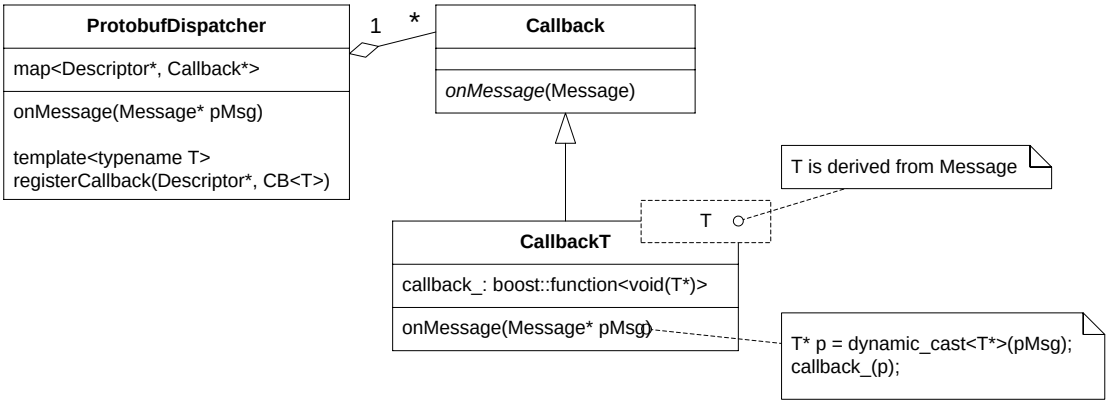


如果我希望 QueryServer 这么设计：不想每个消息处理函数自己做 down casting，而是交给 dispatcher 去处理，客户代码拿到的就已经是想要的具体类型。如下：



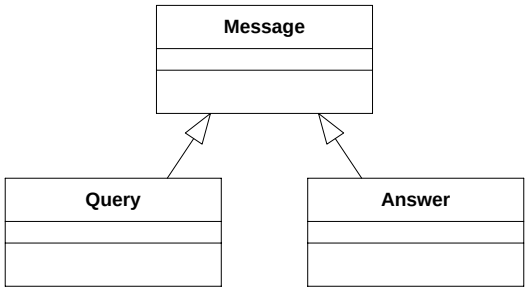
那么该该如何实现 ProtobufDispatcher 呢？它如何与多个未知的消息类型合作？做 down cast 需要知道目标类型，难道我们要用一长串模板类型参数吗？

有一个办法，把多态与模板结合，利用 templated derived class 来提供类型上的灵活性。设计如下。



ProtobufDispatcher 有一个模板成员函数，可以接受注册任意消息类型 T 的回调，然后它创建一个模板化的派生类 CallbackT<T>，这样消息的类新信息就保存在了 CallbackT<T> 中，做 down casting 就简单了。

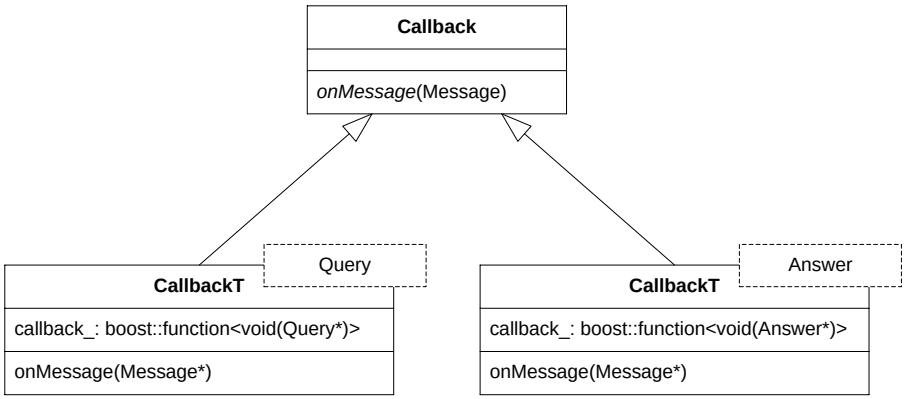
比方说，我们有两个具体消息类型 Query 和 Answer。



然后我们这样注册回调：

```
dispatcher_.registerMessageCallback<muduo::Query>(
    boost::bind(&QueryServer::onQuery, this, _1, _2, _3));
dispatcher_.registerMessageCallback<muduo::Answer>(
    boost::bind(&QueryServer::onAnswer, this, _1, _2, _3));
```

这样会具现化 (instantiation) 出两个 CallbackT 实体，如下：



以上设计参考了 shared_ptr 的 deleter，Scott Meyers 也谈到过 [27]。

2.6.6 ProtobufCodec 和 ProtobufDispatcher 有何意义？

ProtobufCodec 和 ProtobufDispatcher 把每个直接收发 protobuf Message 的网络程序都会用到的功能提炼出来做成了公用的 utility，这样以后新写 protobuf 网络程序就不必为打包分包和消息分发劳神了。它俩以库的形式存在，是两个可以拿来就当 data member 用的 class。它们没有基类，也没有用到虚函数或者别的什么面向对象特征，不侵入 muduo::net 或者你的代码。如果不这么做，那将来每个 protobuf 网络程序都要自己重新实现类似的功能，徒增负担。

第 **??** 节讲“分布式程序的自动回归测试”[28] 会介绍利用 protobuf 的跨语言特性，采用 Java 为 C++ 服务程序编写 test harness。

2.7 限制服务器的最大并发连接数

本节已以大家都熟悉的 EchoServer 介绍如何限制服务器的并发连接数。代码见 examples/maxconnection/。

这篇文章中的“并发连接数”是指一个 server program 能同时支持的客户端连接数，连接系由客户端主动发起，服务端被动接受 (accept) 连接。（如果要限制应用程序主动发起的连接，则问题要简单得多，毕竟主动权和决定权都在程序本身。）

[27] http://www.artima.com/cppsource/top_cpp_aha_moments.html
[28] http://blog.csdn.net/solstice/article/details/6359748

2.7.1 为什么要限制并发连接数?

一方面，我们不希望服务程序超载，另一方面，更因为 file descriptor 是稀缺资源，如果出现 file descriptor 耗尽，很棘手，跟“malloc() 失败/new 抛出 std::bad_alloc”差不多同样棘手。

我 2010 年 10 月在《分布式系统的工程化开发方法》演讲 [29] 中曾谈到 libev 的作者 Marc Lehmann 建议的一种应对“accept()ing 时 file descriptor 耗尽”的办法 [30]。

在服务端网络编程中，我们通常用 reactor 模式来处理并发连接。listening socket 是一种特殊的 IO 对象，当有新连接到达时，此 listening 文件描述符变得可读，epoll_wait 返回这一事件。然后我们用 accept(2) 系统调用获得新连接的 socket 文件描述符。代码主体逻辑如下 (Python):

```
1  serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2  serversocket.bind(('', 2007))
3  serversocket.listen(5)
4  serversocket.setblocking(0)
5
6  poll = select.poll()  # epoll() should work the same
7  poll.register(serversocket.fileno(), select.POLLIN)
8  connections = {}
9
10 while True:
11     events = poll.poll(10000)  # 10 seconds
12     for fileno, event in events:
13         if fileno == serversocket.fileno():
14             (clientsocket, address) = serversocket.accept()
15             clientsocket.setblocking(0)
16             poll.register(clientsocket.fileno(), select.POLLIN)
17             connections[clientsocket.fileno()] = clientsocket
18         elif event & select.POLLIN:
19             # ...
```

假如第 14 行 accept(2) 返回 EMFILE 该如何应对？这意味着本进程的文件描述符已经达到上限，无法为新连接创建 socket 文件描述符。但是，既然没有 socket 文件描述符来表示这个连接，我们就无法 close(2) 它。程序继续运行，回到第 11 行再一次调用 epoll_wait。这时候 epoll_wait 会立刻返回，因为新连接还等待处理，listening fd 还是可读的。这样程序立刻就陷入了 busy loop，CPU 占用率接近 100%。既影响同一 event loop 上的连接，也影响同一机器上的其他服务。

该怎么办呢？Marc Lehmann 提到了几种做法

[29] http://blog.csdn.net/solstice/article/details/5950190
http://www.youku.com/playlist_show/id_5238686.html

[30] http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#The_special_problem_of_accept_ing_wh

1. 调高进程的文件描述符数目。治标不治本，因为只要有足够多的客户端就一定能把一个服务进程的文件描述符用完。
2. 死等。鸵鸟算法。
3. 退出程序。小题大作，为了这种暂时的错误而中断现有的服务似乎不值得。
4. 关闭 listening fd。那么什么时候重新打开呢？
5. 改用 edge trigger。如果漏掉了一次 `accept(2)` 程序再也不会收到新连接。
6. 准备一个空闲的文件描述符。遇到这种情况，先关闭这个空闲文件，获得一个文件描述符的名额；再 `accept(2)` 拿到新 socket 连接的描述符；随后立刻 `close(2)` 它，这样就优雅地断开了客户端连接；最后重新打开一个空闲文件，把坑占住，以备再次出现这种情况时使用。

第 2、5 两种做法会导致客户端认为连接已建立，但无法获得服务，因为服务端程序没有拿到连接的文件描述符。

Muduo 的 acceptor 正是用第 6 种方案实现的，见 muduo/net/Acceptor.cc。但是，这个做法在多线程下不能保证正确，会有 race condition。（思考题：是什么 race condition？）

其实有另外一种比较简单的办法：file descriptor 是 hard limit，我们可以自己设一个稍低一点的 soft limit，如果超过 soft limit 就主动关闭新连接，这样就避免触及“file descriptor 耗尽”这种边界条件。比方说当前进程的 max file descriptor 是 1024，那么我们可以在连接数达到 1000 的时候进入“拒绝新连接”状态，这样留给我们足够的腾挪空间。

2.7.2 Muduo 中限制并发连接数

Muduo 中限制并发连接数的做法简单得出奇。以在第 1.4.2 节的 EchoServer 为例，只需要为它增加一个 int 成员，表示当前的活动连接数。（如果是多线程程序，应该用 muduo::AtomicInt32。）

```
$ diff examples/simple/echo/echo.h examples/maxconnection/echo.h -u
--- examples/simple/echo/echo.h 2012-03-14 21:51:13.000000000 +0800
+++ examples/maxconnection/echo.h       2012-03-11 12:55:44.000000000 +0800
@@ -8,9 +8,10 @@
 {
  public:
   EchoServer(muduo::net::EventLoop* loop,
!             const muduo::net::InetAddress& listenAddr,
+             int maxConnections);  // kMaxConnections_ = maxConnections
```

```
    void start();

  private:
    void onConnection(const muduo::net::TcpConnectionPtr& conn);
@@ -21,6 +22,8 @@

    muduo::net::EventLoop* loop_;
    muduo::net::TcpServer server_;
+   int numConnected_; // should be atomic_int
+   const int kMaxConnections_;
  };
```

然后，在 EchoServer::onConnection() 中判断当前活动连接数，如果超过最大允许数，则踢掉连接。

examples/maxconnection/echo.cc

```
  void EchoServer::onConnection(const TcpConnectionPtr& conn)
  {
    LOG_INFO << "EchoServer - " << conn->peerAddress().toIpPort() << " -> "
             << conn->localAddress().toIpPort() << " is "
             << (conn->connected() ? "UP" : "DOWN");
+
+   if (conn->connected())
+   {
+     ++numConnected_;
+     if (numConnected_ > kMaxConnections_)  // 如果超过最大允许数，则踢掉连接
+     {
+       conn->shutdown();
+     }
+   }
+   else
+   {
+     --numConnected_;
+   }
+   LOG_INFO << "numConnected = " << numConnected_;
  }
```

examples/maxconnection/echo.cc

这种做法可以积极地防止耗尽 file descriptor。

另外，如果是有业务逻辑的服务，可以在 shutdown() 之前发送一个简单的响应，表明本服务程序的负载能力已经饱和，提示客户端尝试下一个可用的 server（当然，下一个可用的 server 地址不一定要在这个响应里给出，客户端可以自己去 name service 查询），这样方便客户端快速 failover。

后文第 2.10 节将介绍如何处理空闲连接的超时：如果一个连接长时间（若干秒）没有输入数据，则踢掉此连接。办法有很多种，我用 Time Wheel 解决。

2.8 定时器

从本节开始的三节内容都与非阻塞网络编程中的定时任务有关。

2.8.1 程序中的时间

程序中对时间的处理是个大问题，在这一节中我先简要谈谈与编程直接相关的内容，把更深入的内容留给日期与时间专题文章。

在一般的服务端程序设计中，与时间有关的常见任务有：

1. 获取当前时间，计算时间间隔；
2. 时区转换与日期计算；把纽约当地时间转换为上海当地时间；2011-02-05 之后第 100 天是几月几号星期几？等等
3. 定时操作，比如在预定的时间执行一项任务，或者在一段延时之后执行一项任务。

其中第 2 项看起来复杂，其实最简单。日期计算用 Julian Day Number，时区转换用 tz database；惟一麻烦一点的是夏令时，但也可以用 tz database 解决。这些操作都是纯函数，很容易用一套单元测试来验证代码的正确性。需要特别注意的是，用 tzset/localtime_r 来做时区转换在多线程环境下可能会有问题；对此我的解决办法是写一个 TimeZone class，以避免影响全局，将来在日期与时间专题中会讲到。以下本文不考虑时区，均为 UTC 时间。

真正麻烦的是第 1 项和第 3 项。一方面，Linux 有一大把令人眼花缭乱的与时间相关的函数和结构体，在程序中该如何选用？另一方面，计算机中的时钟不是理想的计时器，它可能会漂移或跳变；最后，民用的 UTC 时间与闰秒的关系也让定时任务变得复杂和微妙。当然，与系统当前时间有关的操作也让单元测试变得困难。

2.8.2 Linux 时间函数

Linux 的计时函数，用于获得当前时间：

- time(2) / time_t （秒）
- ftime(3) / struct timeb （毫秒）
- gettimeofday(2) / struct timeval （微秒）
- clock_gettime(2) / struct timespec （纳秒）

- gmtime / localtime / timegm / mktime / strftime / struct tm （这些与当前时间无关）

定时函数，用于让程序等待一段时间或安排计划任务：

- sleep
- alarm
- usleep
- nanosleep
- clock_nanosleep
- getitimer / setitimer
- timer_create / timer_settime / timer_gettime / timer_delete
- timerfd_create / timerfd_gettime / timerfd_settime

我的取舍如下：

- （计时）只使用 gettimeofday 来获取当前时间。
- （定时）只使用 timerfd_* 系列函数来处理定时。

gettimeofday 入选原因：（这也是 muduo::Timestamp class 的主要设计考虑）

1. time 的精度太低，ftime 已被废弃，clock_gettime 精度最高，但是它系统调用的开销比 gettimeofday 大。
2. 在 x86-64 平台上，gettimeofday 不是系统调用，而是在用户态实现的（搜 vsyscall），没有上下文切换和陷入内核的开销。
3. gettimeofday 的分辨率 (resolution) 是 1 微秒，足以满足日常计时的需要。muduo::Timestamp 用一个 int64_t 来表示从 Epoch 到现在的微秒数，其范围可达上下 30 万年。

timerfd_* 入选的原因：

1. sleep / alarm / usleep 在实现时有可能用了信号 SIGALRM，在多线程程序中处理信号是个相当麻烦的事情，应当尽量避免。
2. nanosleep 和 clock_nanosleep 是线程安全的，但是在非阻塞网络编程中，绝对不能用让线程挂起的方式来等待一段时间，程序会失去响应。正确的做法是注册一个时间回调函数。

3. getitimer 和 timer_create 也是用信号来 deliver 超时，在多线程程序中也会有麻烦。timer_create 可以指定信号的接收方是进程还是线程，算是一个进步，不过在信号处理函数 (signal handler) 能做的事情实在很受限。
4. timerfd_create 把时间变成了一个文件描述符，该“文件”在定时器超时的那一刻变得可读，这样就能很方便地融入到 select/poll 框架中，用统一的方式来处理 IO 事件和超时事件，这也正是 Reactor 模式的长处。我在以前发表的《Linux 新增系统调用的启示》[31]中也谈到这个想法，现在我把这个想法在 muduo 网络库中实现了。
5. 传统的 Reactor 利用 select/poll/epoll 的 timeout 来实现定时功能，但 poll 和 epoll 的定时精度只有毫秒，远低于 timerfd_settime 的定时精度。

必须要说明，在 Linux 这种非实时多任务操作系统中，在用户态实现完全精确可控的计时和定时是做不到的，因为当前任务可能会被随时切换出去，这在 CPU 负载大的时候尤为明显。但是，我们的程序可以尽量提高时间精度，必要的时候通过控制 CPU 负载来提高时间操作的可靠性，在程序在 99.99% 的时候都是按预期执行的。这或许比换用实时操作系统并重新编写并测试代码要经济一些。

关于时间的精度 (accuracy) 问题我留到专题博客文章中讨论，它与分辨率 (resolution) 不完全是一回事儿。时间跳变和闰秒的影响与应对也不在此处展开讨论了。

2.8.3 Muduo 的定时器接口

Muduo EventLoop 有三个定时器函数：

muduo/net/EventLoop.h

```
typedef boost::function<void()> TimerCallback;

class EventLoop : boost::noncopyable
{
 public:
  // ...

  // timers

  /// Runs callback at 'time'.
  TimerId runAt(const Timestamp& time, const TimerCallback& cb);

  /// Runs callback after @c delay seconds.
```

[31] http://blog.csdn.net/Solstice/article/details/5327881

```
    TimerId runAfter(double delay, const TimerCallback& cb);

    /// Runs callback every @c interval seconds.
    TimerId runEvery(double interval, const TimerCallback& cb);

    /// Cancels the timer.
    void cancel(TimerId timerId);

    // ...
  };
```

muduo/net/EventLoop.h

函数名称很好地反映了其用途：

- runAt 在指定的时间调用 TimerCallback
- runAfter 等一段时间调用 TimerCallback
- runEvery 以固定的间隔反复调用 TimerCallback
- cancel 取消 timer

回调函数在 EventLoop 对象所属的线程发生，与 onMessage() onConnection() 等网络事件函数在同一个线程。Muduo 的 TimerQueue 采用了平衡二叉树来管理未到期的 timers，因此这些操作的事件复杂度是 $O(\log N)$。

2.8.4 Boost.Asio Timer 示例

Boost.Asio 教程 [32] 里以 Timer 和 Daytime 为例介绍 asio 的基本使用，daytime 已经在第 2.1 节中介绍过，这里着重谈谈 Timer。Asio 有 5 个 Timer 示例，muduo 把其中四个重新实现了一遍，并扩充了第 5 个示例。

1. 阻塞式的定时，muduo 不支持这种用法，无代码。
2. 非阻塞定时，见 examples/asio/tutorial/timer2
3. 在 TimerCallback 里传递参数，见 examples/asio/tutorial/timer3
4. 以成员函数为 TimerCallback，见 examples/asio/tutorial/timer4
5. 在多线程中回调，用 mutex 保护共享变量，见 examples/asio/tutorial/timer5
6. 在多线程中回调，缩小临界区，把不需要互斥执行的代码移出来，见 examples/asio/tutorial/timer6

[32] http://www.boost.org/doc/libs/release/doc/html/boost_asio/tutorial.html

为节省篇幅，这里只列出 timer4。这个程序的功能是以 1 秒为间隔打印 5 个整数，乍看起来代码有的小题大做，但是值得注意的是定时器事件与 IO 事件是在同一线程发生，程序就像处理 IO 事件一样处理超时事件。

examples/asio/tutorial/timer4/timer.cc

```
7  class Printer : boost::noncopyable
8  {
9   public:
10   Printer(muduo::net::EventLoop* loop)
11     : loop_(loop),
12       count_(0)
13   {
14     loop_->runAfter(1, boost::bind(&Printer::print, this));
15   }
16
17   ~Printer()
18   {
19     std::cout << "Final count is " << count_ << "\n";
20   }
21
22   void print()
23   {
24     if (count_ < 5)
25     {
26       std::cout << count_ << "\n";
27       ++count_;
28
29       loop_->runAfter(1, boost::bind(&Printer::print, this));
30     }
31     else
32     {
33       loop_->quit();
34     }
35   }
36
37 private:
38   muduo::net::EventLoop* loop_;
39   int count_;
40 };
41
42 int main()
43 {
44   muduo::net::EventLoop loop;
45   Printer printer(&loop);
46   loop.loop();
47 }
```

examples/asio/tutorial/timer4/timer.cc

最后我再强调一遍，在非阻塞服务端编程中，绝对不能用 sleep 或类似的办法来让程序原地停留等待，这会让程序失去响应，因为主事件循环被挂起了，无法处

理 IO 事件。这就像在 Windows 编程中绝对不能在消息循环里执行耗时的代码一样，会让程序界面失去响应。Reactor 模式的网络编程确实有些类似传统的消息驱动的 Windows 编程。对于“定时”任务，就把它变成一个特定的消息，到时候触发相应的消息处理函数就行了。

Boost.Asio 的 timer 示例只用到了 EventLoop::runAfter，我再举一个 EventLoop::runEvery 的例子。

2.8.5 Java Netty 示例

Netty 是一个非常好的 Java NIO 网络库，它附带的示例程序有 echo 和 discard 两个简单网络协议。与第 2.1 节不同，Netty 版的 echo 和 discard 服务端有流量统计功能，这需要用到固定间隔的定时器（EventLoop::runEvery）。

其 client 的代码类似前文的 chargen，为节省篇幅，请阅读源码 examples/netty/discard/client.cc 。

这里列出 discard server 的完整代码。代码整体结构上与第 1.4.2 节的 EchoServer 差别不大，这算是简单网络服务器的典型模式了。

DiscardServer 可以配置成多线程服务器，muduo TcpServer 有一个内置的 one loop per thread 多线程 IO 模型，可以通过 setThreadNum() 来开启。

muduo/examples/netty/discard/server.cc

```
19 int numThreads = 0;
20
21 class DiscardServer
22 {
23  public:
24   DiscardServer(EventLoop* loop, const InetAddress& listenAddr)
25     : loop_(loop),
26       server_(loop, listenAddr, "DiscardServer"),
27       oldCounter_(0),
28       startTime_(Timestamp::now())
29   {
30     server_.setConnectionCallback(
31         boost::bind(&DiscardServer::onConnection, this, _1));
32     server_.setMessageCallback(
33         boost::bind(&DiscardServer::onMessage, this, _1, _2, _3));
34     server_.setThreadNum(numThreads);
35     loop->runEvery(3.0, boost::bind(&DiscardServer::printThroughput, this));
36   }
```

构造函数注册了一个间隔为 3 秒的定时器，调用 DiscardServer::printThroughput 打印出吞吐量。

消息回调只比与第 2.1.1 节的代码多两行，用于统计收到的数据长度和消息次数。

```
52   void onMessage(const TcpConnectionPtr& conn, Buffer* buf, Timestamp)
53   {
54     size_t len = buf->readableBytes();
55     transferred_.add(len);
56     receivedMessages_.incrementAndGet();
57     buf->retrieveAll();
58   }
```

在每一个统计周期，打印数据吞吐量。

```
60   void printThroughput()
61   {
62     Timestamp endTime = Timestamp::now();
63     int64_t newCounter = transferred_.get();
64     int64_t bytes = newCounter - oldCounter_;
65     int64_t msgs = receivedMessages_.getAndSet(0);
66     double time = timeDifference(endTime, startTime_);
67     printf("%4.3f MiB/s %4.3f Ki Msgs/s %6.2f bytes per msg\n",
68         static_cast<double>(bytes)/time/1024/1024,
69         static_cast<double>(msgs)/time/1024,
70         static_cast<double>(bytes)/static_cast<double>(msgs));
71
72     oldCounter_ = newCounter;
73     startTime_ = endTime;
74   }
```

以下是数据成员，注意用了整数的原子操作 AtomicInt64 来记录收到的字节数和消息数，这是为了多线程安全性。

```
76   EventLoop* loop_;
77   TcpServer server_;
78
79   AtomicInt64 transferred_;
80   AtomicInt64 receivedMessages_;
81   int64_t oldCounter_;
82   Timestamp startTime_;
83 };
```

主函数，有一个可选的命令行参数，用于指定线程数目。

```
85 int main(int argc, char* argv[])
86 {
87   LOG_INFO << "pid = " << getpid() << ", tid = " << CurrentThread::tid();
88   if (argc > 1)
89   {
90     numThreads = atoi(argv[1]);
91   }
```

```
92    EventLoop loop;
93    InetAddress listenAddr(2009);
94    DiscardServer server(&loop, listenAddr);
95
96    server.start();
97
98    loop.loop();
99  }
```

muduo/examples/netty/discard/server.cc

运行方法，在同一台机器的两个命令行窗口分别运行：

```
$ bin/netty_discard_server
$ bin/netty_discard_client 127.0.0.1 256
```

第一个窗口显示吞吐量：

```
41.001 MiB/s 73.387 Ki Msgs/s 572.10 bytes per msg
72.441 MiB/s 129.593 Ki Msgs/s 572.40 bytes per msg
77.724 MiB/s 137.251 Ki Msgs/s 579.88 bytes per msg
```

改变第二个命令的最后一个参数（上面的 256），可以观察不同的消息大小对吞吐量的影响。

练习 1：把二者的关系绘制成函数曲线，看看有什么规律，想想为什么。

练习 2：在局域网的两台机器上运行客户端和服务端，找出让吞吐量达到最大的消息长度。这个数字与练习 1 中的相比是大还是小？为什么？

有兴趣的读者可以对比一下 Netty 的吞吐量，muduo 应该能轻松取胜。

discard client/server 测试的是单向吞吐量，echo client/server 测试的是双向吞吐量。这两个服务端都支持多个并发连接，两个客户端都是单连接的。前文第 1.5 节实现了一个 pingpong 协议，客户端和服务端都是多连接，用来测试 muduo 在多线程大量连接情况下的表现

2.9 测量两台机器的网络延迟

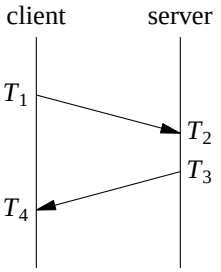
本节介绍一个简单的网络程序 roundtrip，用于测量两台机器之间的网络延迟，即“往返时间/ round trip time / RTT”。这篇文章主要考察定长 TCP 消息的分包与 TCP_NODELAY 的作用。

本节的代码见 examples/roundtrip/roundtrip.cc

测量 RTT 的办法很简单：

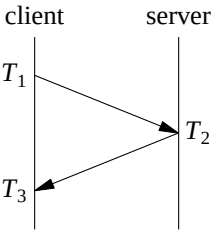
- host A 发一条消息给 host B，其中包含 host A 发送消息的本地时间
- host B 收到之后立刻把消息 echo 回 host A
- host A 收到消息之后，用当前时间减去消息中的时间就得到了 RTT。

NTP 协议的工作原理与之类似，不过，除了测量 RTT，NTP 还需要知道两台机器之间的时间差 (clock offset)，这样才能校准时间。



以上是 NTP 协议收发消息的协议，RTT = $(T4-T1)-(T3-T2)$，clock offset = $\frac{(T4+T1)-(T2+T3)}{2}$。NTP 的要求是往返路径上的单程延迟要尽量相等，这样才能减少系统误差。偶然误差由单程延迟的不确定性决定。

在我设计的 roundtrip 示例程序中，协议有所简化：

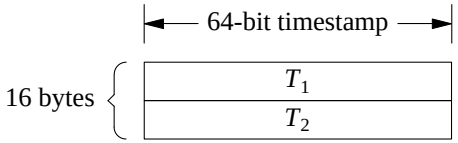


$$\text{round trip time} = T_3 - T_1$$

$$\text{clock offset} = T_2 - \frac{T_1 + T_3}{2}$$

简化之后的协议少取一次时间，因为 server 收到消息之后立刻发送回 client，耗时很少（若干微秒），基本不影响最终结果。

我设计的消息格式是 16 字节定长消息：

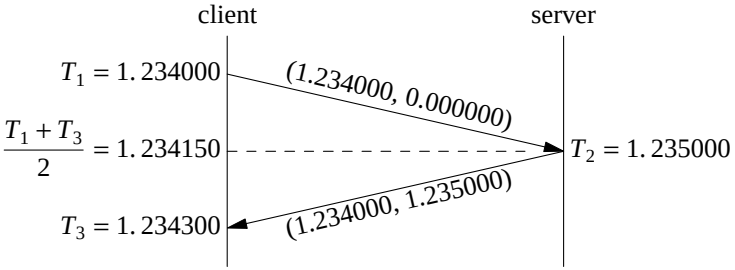


T1 和 T2 都是 muduo::Timestamp，成员是一个 int64_t，表示从 Epoch 到现在的微秒数。为了让消息的单程往返时间接近，server 和 client 发送的消息都是 16 bytes，这样做到对称。由于是定长消息，可以不必使用 codec，在 message callback 中直接用

```
while (buffer->readableBytes() >= frameLen)
{
  // ...
}
```

就能 decode。请读者思考，如果把 while 换成 if 会有什么后果？

client 程序以 200ms 为间隔发送消息，在收到消息之后打印 RTT 和 clock offset。一次运作实例如下：



$$\text{round trip time} = T_3 - T_1 = 300\ \mu\text{s}$$

$$\text{clock offset} = T_2 - \frac{T_1+T_3}{2} = 850\ \mu\text{s}$$

这个例子中，client 和 server 的时钟不是完全对准的，server 的时间快了 850 us，用 roundtrip 程序能测量出这个时间差。有了这个时间差就能校正分布式系统中测量得到的消息延迟。

比方说以上图为例，server 在它本地 1.235000 时刻发送了一条消息，client 在它本地 1.234300 收到这条消息，直接计算的话延迟是 -700us。这个结果肯定是错的，因为 server 和 client 不在一个时钟域（clock domain，这是数字电路中的概念），它们的时间直接相减无意义。如果我们已经测量得到 server 比 client 快 850us，那么做用这个数据一次校正：-700+850 = 150us，这个结果就比较符合实际了。当然，在实际应用中，clock offset 要经过一个低通滤波才能使用，不然偶然性太大。

请读者思考，为什么不能直接以 RTT/2 作为两台机器之间收发消息的单程延迟？这个数字是偏大还是偏小？

这个程序在局域网中使用没有问题，如果在广域网上使用，而且 RTT 大于 200ms，那么受 Nagle 算法影响，测量结果是错误的。因为应用程序记录的发包时间与操作系统真正发出数据包的时间之差不再是一个可以忽略的小间隔，具体分析留作练习，这能测试对 Nagle 的理解。这时候我们需要设置 TCP_NODELAY 参数，让程序在广域网上也能正常工作。

2.10 用 Timing wheel 踢掉空闲连接

本节介绍如何使用 timing wheel 来踢掉空闲的连接，一个连接如果若干秒没有收到数据，就认为是空闲连接。本文的代码见 examples/idleconnection。

在严肃的网络程序中，应用层的心跳协议是必不可少的。应该用心跳消息来判断对方进程是否能正常工作，“踢掉空闲连接”只是一时权宜之计。我这里想顺便讲讲 shared_ptr 和 weak_ptr 的用法。

如果一个连接连续几秒钟（后文以 8s 为例）内没有收到数据，就把它断开，为此有两种简单粗暴的做法：

- 每个连接保存“最后收到数据的时间 lastReceiveTime”，然后用一个定时器，每秒钟遍历一遍所有连接，断开那些 (now - connection.lastReceiveTime) > 8s 的 connection。这种做法全局只有一个 repeated timer，不过每次 timeout 都要检查全部连接，如果连接数目比较大（几千上万），这一步可能会比较费时。
- 每个连接设置一个 one-shot timer，超时定为 8s，在超时的时候就断开本连接。当然，每次收到数据要去更新 timer。这种做法需要很多个 one-shot timer，会频繁地更新 timers。如果连接数目比较大，可能对 reactor 的 timer queue 造成压力。

使用 timing wheel 能避免上述两种做法的缺点。timing wheel 可以翻译为“时间轮盘”或“刻度盘”，本文保留英文。

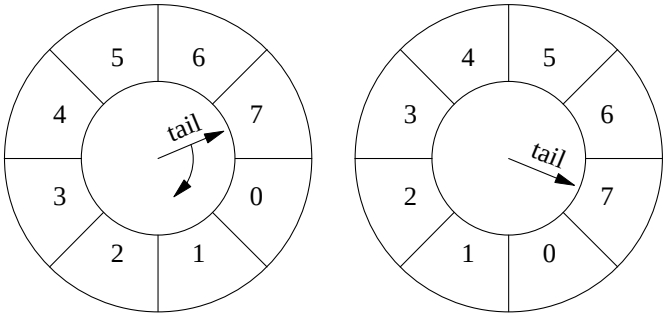
连接超时不需要精确定时，只要大致 8 秒钟超时断开就行，多一秒少一秒关系不大。处理连接超时可以用一个简单的数据结构：8 个桶组成的循环队列。第一个桶放下一秒将要超时的连接，第二个放下 2 秒将要超时的连接。每个连接一收到数据就把自己放到第 8 个桶，然后在每秒钟的 callback 里把第一个桶里的连接断开，把这个空桶挪到队尾。这样大致可以做到 8 秒钟没有数据就超时断开连接。更重要的是，每次不用检查全部的 connection，只要检查第一个桶里的 connections，相当于把任务分散了。

2.10.1 Timing wheel 原理

《Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility》这篇论文详细比较了实现定时器的各种数据结构，并提出了层次化的 timing wheel 与 hash timing wheel 等新结构。针对本文要解决的问题的特点，我们不需要实现一个通用的定时器，只用实现 simple timing wheel 即可。

Simple timing wheel 的基本结构是一个循环队列，还有一个指向队尾的指针 (tail)，这个指针每秒钟移动一格，就像钟表上的时针，timing wheel 由此得名。

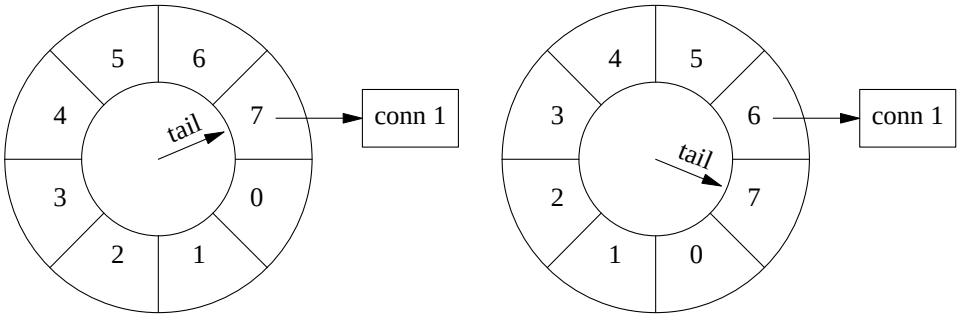
以下是某一时刻 timing wheel 的状态（左图），格子里的数字是倒计时（与通常的 timing wheel 相反），表示这个格子（桶子）中的连接的剩余寿命。



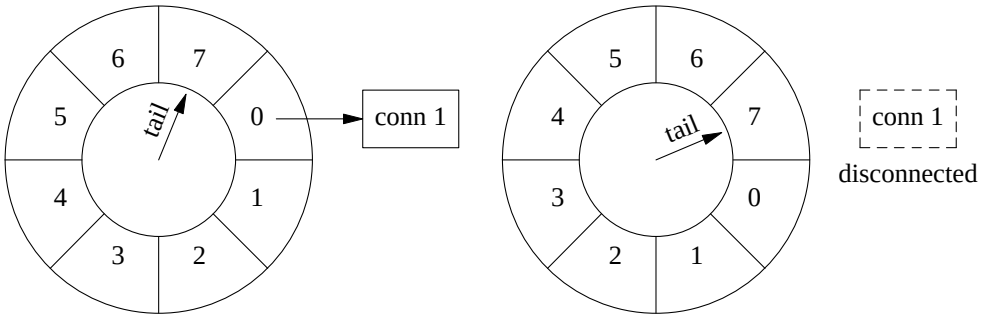
一秒钟以后（右图），tail 指针移动一格，原来四点钟方向的格子被清空，其中的连接已被断开。

连接超时被踢掉的过程

假设在某个时刻，conn 1 到达，把它放到当前格子中，它的剩余寿命是 7 秒（下左图）。此后 conn 1 上没有收到数据。1 秒钟之后（下右图），tail 指向下一个格子，conn 1 的剩余寿命是 6 秒。

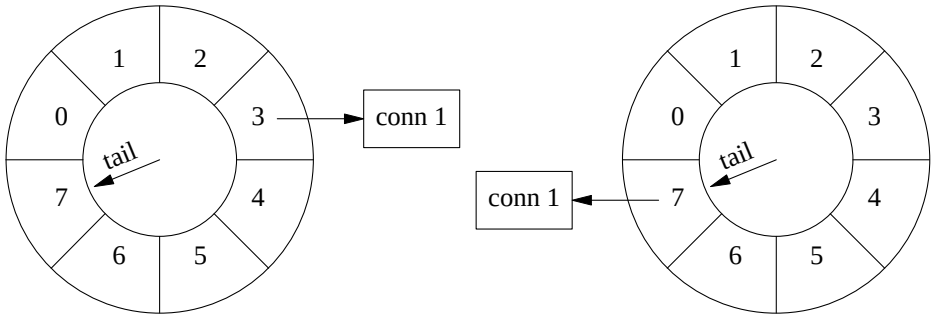


又过了几秒钟，tail 指向 conn 1 之前的那个格子，conn 1 即将被断开（下左图）。下一秒（下右图），tail 重新指向 conn 1 原来所在的格子，清空其中的数据，断开 conn 1 连接。

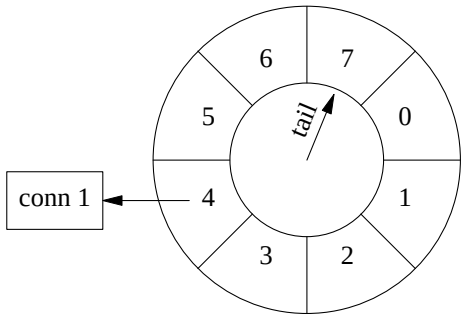


连接刷新

如果在断开 conn 1 之前收到数据，就把它移到当前的格子里。conn 1 的剩余寿命是 3 秒（下左图），此时 conn 1 收到数据，它的寿命恢复为 7 秒（下右图）。



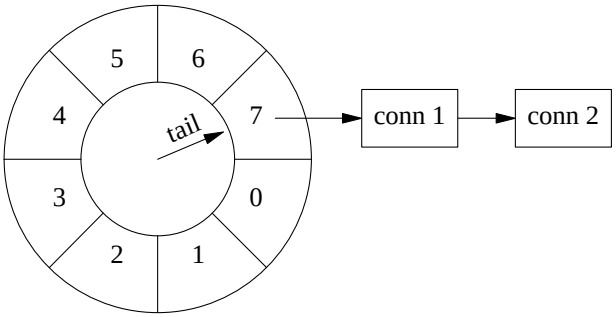
时间继续前进，conn 1 寿命递减，不过它已经比第一种情况长寿了。



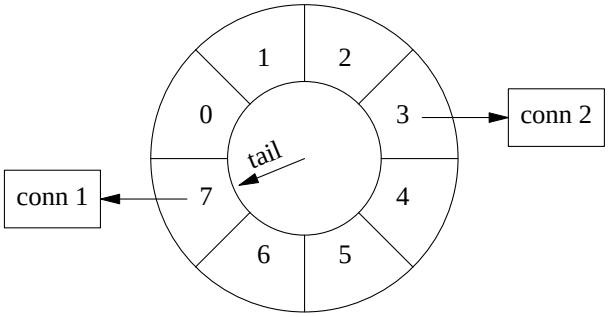
多个连接

timing wheel 中的每个格子是个 hash set，可以容纳不止一个连接。

比如一开始，conn 1 到达。随后，conn 2 到达（下图），这时候 tail 还没有移动，两个连接位于同一个格子中，具有相同的剩余寿命。(下图中画成链表，代码中是哈希表。)



几秒钟之后，conn 1 收到数据，而 conn 2 一直没有收到数据，那么 conn 1 被移到当前的格子中。这时 conn 1 的寿命比 conn 2 长。



2.10.2 代码实现与改进

我们用以前多次出现的 EchoServer 来说明具体如何实现 timing wheel。代码见 examples/idleconnection。

在具体实现中，格子里放的不是连接，而是一个特制的 Entry struct，每个 Entry 包含 TcpConnection 的 weak_ptr。Entry 的析构函数会判断连接是否还存在（用 weak_ptr），如果还存在则断开连接。

数据结构：

examples/idleconnection/echo.h

```
33  struct Entry : public muduo::copyable
34  {
35    Entry(const WeakTcpConnectionPtr& weakConn)
36      : weakConn_(weakConn)
37    {
38    }
39
40    ~Entry()
41    {
42      muduo::net::TcpConnectionPtr conn = weakConn_.lock();
43      if (conn)
44      {
45        conn->shutdown();
46      }
47    }
48
49    WeakTcpConnectionPtr weakConn_;
50  };
51  typedef boost::shared_ptr<Entry> EntryPtr;
52  typedef boost::weak_ptr<Entry> WeakEntryPtr;
53  typedef boost::unordered_set<EntryPtr> Bucket;
54  typedef boost::circular_buffer<Bucket> WeakConnectionList;
```

examples/idleconnection/echo.h

在实现中，为了简单起见，我们不会真的把一个连接从一个格子移到另一个格子，而是采用引用计数的办法，用 shared_ptr 来管理 Entry。如果从连接收到数据，就把对应的 EntryPtr 放到这个格子里，这样它的引用计数就递增了。当 Entry 的引用计数递减到零，说明它没有在任何一个格子里出现，那么连接超时，Entry 的析构函数会断开连接。

Timing wheel 用 boost::circular_buffer 实现，其中每个 Bucket 元素是个 hash set of EntryPtr。

在构造函数中，注册每秒钟的回调（EventLoop::runEvery() 注册 EchoServer::onTimer() ），然后把 timing wheel 设为适当的大小。

examples/idleconnection/echo.cc

```
15  EchoServer::EchoServer(EventLoop* loop,
16                         const InetAddress& listenAddr,
17                         int idleSeconds)
18    : loop_(loop),
19      server_(loop, listenAddr, "EchoServer"),
20      connectionBuckets_(idleSeconds)
21  {
22    server_.setConnectionCallback(
23        boost::bind(&EchoServer::onConnection, this, _1));
24    server_.setMessageCallback(
```

```
25        boost::bind(&EchoServer::onMessage, this, _1, _2, _3));
26    loop->runEvery(1.0, boost::bind(&EchoServer::onTimer, this));
27    connectionBuckets_.resize(idleSeconds);
28  }
```

examples/idleconnection/echo.cc

其中 EchoServer::onTimer() 的实现只有一行：往队尾添加一个空的 Bucket，这样 circular_buffer 会自动弹出队首的 Bucket，并析构之。在析构 Bucket 的时候，会依次析构其中的 EntryPtr 对象，这样 Entry 的引用计数就不用我们去操心，C++ 的值语意会帮我们搞定一切。

```
void EchoServer::onTimer()
{
  connectionBuckets_.push_back(Bucket());
}
```

在连接建立时，创建一个 Entry 对象，把它放到 timing wheel 的队尾。另外，我们还需要把 Entry 的弱引用保存到 TcpConnection 的 context 里，因为在收到数据的时候还要用到 Entry。（思考题：如果 TcpConnection::setContext 保存的是强引用 EntryPtr，会出现什么情况？）

examples/idleconnection/echo.cc

```
36  void EchoServer::onConnection(const TcpConnectionPtr& conn)
37  {
38    LOG_INFO << "EchoServer - " << conn->peerAddress().toIpPort() << " -> "
39             << conn->localAddress().toIpPort() << " is "
40             << (conn->connected() ? "UP" : "DOWN");
41
42    if (conn->connected())
43    {
44      EntryPtr entry(new Entry(conn));
45      connectionBuckets_.back().insert(entry);
46      WeakEntryPtr weakEntry(entry);
47      conn->setContext(weakEntry);
48    }
49    else
50    {
51      assert(!conn->getContext().empty());
52      WeakEntryPtr weakEntry(boost::any_cast<WeakEntryPtr>(conn->getContext()));
53      LOG_DEBUG << "Entry use_count = " << weakEntry.use_count();
54    }
55  }
```

examples/idleconnection/echo.cc

在收到消息时，从 TcpConnection 的 context 中取出 Entry 的弱引用，把它提升为强引用 EntryPtr，然后放到当前的 timing wheel 队尾。（思考题，为什么要把

Entry 作为 TcpConnection 的 context 保存，如果这里再创建一个新的 Entry 会有什么后果？）

examples/idleconnection/echo.cc

```
58  void EchoServer::onMessage(const TcpConnectionPtr& conn,
59                             Buffer* buf,
60                             Timestamp time)
61  {
62    string msg(buf->retrieveAsString());
63    LOG_INFO << conn->name() << " echo " << msg.size()
64             << " bytes at " << time.toString();
65    conn->send(msg);
66
67    assert(!conn->getContext().empty());
68    WeakEntryPtr weakEntry(boost::any_cast<WeakEntryPtr>(conn->getContext()));
69    EntryPtr entry(weakEntry.lock());
70    if (entry)
71    {
72      connectionBuckets_.back().insert(entry);
73    }
74  }
```

examples/idleconnection/echo.cc

然后呢？没有然后了，程序已经完成了我们想要的功能。（完整的代码会调用 dumpConnectionBuckets() 来打印 circular_buffer 变化的情况，运行一下即可理解。）

希望本文有助于您理解 shared_ptr 和 weak_ptr。

改进

在现在的实现中，每次收到消息都会往队尾添加 EntryPtr （当然，hash set 会帮我们去重。）一个简单的改进措施是，在 TcpConnection 里保存“最后一次往队尾添加引用时的 tail 位置”，然后先检查 tail 是否变化，若无变化则不重复添加 EntryPtr。这样或许能提高效率。

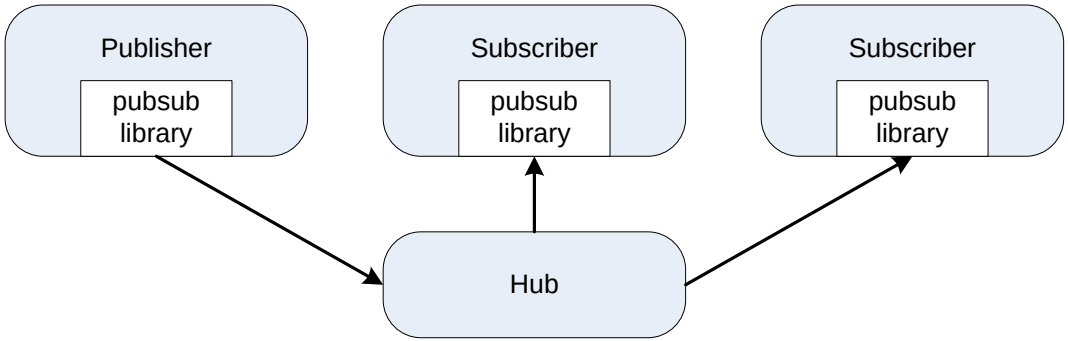
以上改进留作练习。

2.11 简单的消息广播服务

本文介绍用 muduo 实现一个简单的 topic-based 消息广播服务，这其实是“聊天室”的一个简单扩展，不过聊天的不是人，而是分布式系统中的程序。

本文的代码见 examples/hub

在分布式系统中，除了常用的 end-to-end 通信，还有一对多的广播通信。一提到“广播”，或许会让人联想到 IP 多播或 IP 组播，这不是本文的主题。本文将要谈的是基于 TCP 协议的应用层广播。示意图如下：



上图中圆角矩形代表程序，”Hub” 是一个服务程序，不是网络集线器，它起到类似集线器的作用，故而得名。Publisher 和 Subscriper 通过 TCP 协议与 Hub 程序通信。Publisher 把消息发到某个 topic 上，Subscribers 订阅该 topic，然后就能收到消息。即 publisher 借助 hub 把消息广播给了多个 subscribers。这种 pub/sub 结构的好处在于可以增加多个 Subscriber 而不用修改 Publisher，一定程度上实现了“解耦”（也可以看成分布式的 observer pattern）。由于走的是 TCP 协议，广播是基本可靠的，这里的“可靠”指的是“比 UDP 可靠”，不是“完全可靠”。（思考：如何避免 Hub 成为 single point of failure？）

为了避免串扰（cross-talk），每个 topic 在同一时间只应该有一个 publisher，hub 不提供 compare-and-swap 操作。

（“可靠广播、原子广播”在分布式系统中有重大意义，是以 replicated state machine 方式实现可靠的分布式服务的基础，“可靠广播”涉及 consensus 算法，超出了本文的范围。）

应用层广播在分布式系统中用处很大，这里略举几例：

体育比分转播 有 8 片比赛场地正在进行羽毛球比赛，每个场地的计分程序把当前比分发送到各自的 topic 上（第 1 号场地发送到 court1，第 2 号发送到 court2，以此类推）。需要用到比分的程序（赛场的大屏幕显示，网上比分转播等等）自己订阅感兴趣的 topic，就能及时收到最新比分数据。由于本文实现的不是 100% 可靠广播，那么消息应该是 snapshot，而不是 incremental。（换句话说，消息的内容是“现在是几比几”，而不是“刚才谁得分”。）

负载监控 每台机器上运行一个监控程序，周期性地把本机当前负载（CPU、网络、磁盘、温度）publish 到以 hostname 命名的 topic 上，这样需要用到这些数据的程序只要在 hub 订阅相应的 topic 就能获得数据，无需与多台机器直接打交道。（为了可靠起见，监控程序发送的消息里边应该包含时间戳，这样能防止 stale 数据，甚至一定程度上起到心跳的作用。）沿着这个思路，分布式系统中的服务程序也可以把自己的当前负载发布到 hub 上，供 load balancer 和 monitor 取用。

协议

为了简单起见，muduo 的 hub 示例采用以"\r\n" 分界的文本协议，这样用 telnet 就能测试 hub。协议只有三个命令：

- `sub <topic>\r\n`
  该命令表示订阅 <topic>，以后该 topic 有任何跟新都会发给这个 tcp 连接。在 sub 的时候，hub 会把该 <topic> 上最近的消息发给此 subscriber。
- `unsub <topic>\r\n`
  该命令表示退订 <topic>
- `pub <topic>\r\n<content>\r\n`
  往 <topic> 发送消息，内容为 <content>。所有订阅了此 <topic> 的 subscribers 会收到同样的消息“`pub <topic>\r\n<content>\r\n`”

代码

muduo 示例中的 hub 分为几个部分：

- hub 服务程序，负责一对多的消息分发。它会记住每个 client 订阅了哪些 topic，只把消息发给特定的订阅者。代码 examples/hub/hub.cc
- pubsub 库，为了方便编写使用 hub 服务的应用程序，我写了一个简单的 client library，用来和 hub 打交道。这个 library 可以订阅 topic、退订 topic、往指定 topic 发布消息。代码 examples/hub/pubsub.{h,cc}
- sub 示例程序，这个命令行程序订阅一个或多个 topic，然后等待 hub 的数据。代码 examples/hub/sub.cc
- pub 示例程序，这个命令行程序往某个 topic 发布一条消息，消息内容由命令行参数指定。代码 examples/hub/pub.cc

一个程序可以既是 publisher 又是 subscriber，而且 pubsub 库只用一个 tcp 连接（这样 failover 比较简便）。使用范例：

1. 开启 4 个命令行窗口
2. 在第一个窗口运行 $ hub 9999
3. 在第二个窗口运行 $ sub 127.0.0.1:9999 mytopic
4. 在第三个窗口运行 $ sub 127.0.0.1:9999 mytopic court
5. 在第四个窗口运行 $ pub 127.0.0.1:9999 mytopic "Hello world."，这时第二三号窗口都会打印“mytopic: Hello world.”，表明收到了 mytopic 这个主题上的消息。
6. 在第四个窗口运行 $ pub 127.0.0.1:9999 court "13:11"，这时第三号窗口会打印“court: 13:11”，表明收到了 court 这个主题上的消息。第二号窗口没有订阅此消息，故无输出。

借助这个简单的 pub/sub 机制，还可以做很多有意思的事情。比如把分布式系统中的程序的一部分 end-to-end 通信改为通过 pub/sub 来做（例如，原来是 A 向 B 发一个 SOAP request，B 通过同一个 tcp 连接发回 response (分析二者的通信只能通过查看 log 或用 tcpdump 截获)；现在是 A 往 topic_a_to_b 上发布 request，B 在 topic_b_to_a 上发 response)，这样多挂一个 monitoring subscriber 就能轻易地查看通信双方的沟通情况，很容易做状态监控与 trouble shooting。

2.11.1 多现程的高效广播

在本节这个例子中，hub 是个单线程程序。假如有一条消息要广播给 1000 个订阅者，那么只能一个一个地发，第 1 个订阅者收到消息和第 1000 个订阅者收到消息的时差可以长达若干毫秒。那么有没有办法提高速度降低延迟呢？当然会想到用多线程。但是简单的办法并不一定能奏效，因为一个全局锁就把多线程程序退化为单线程执行。为了真正提速，我想到了用 thread local 的办法，比如把 1000 个订阅者分给 4 个线程，每个线程的操作基本都是无锁的，这样可以做到并行地发送消息。示例代码见 examples/asio/chat/server_threaded_highperformance.cc

2.12 “串并转换”连接服务器及其自动化测试

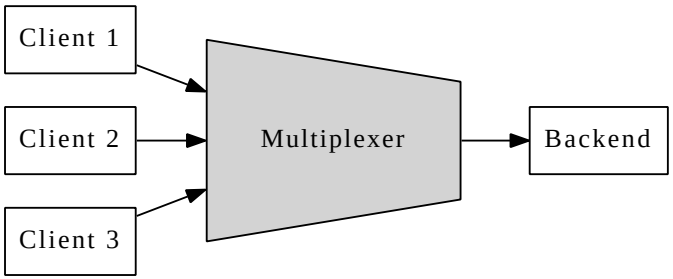
本文介绍如何使用 test harness 来测试一个具有内部逻辑的网络服务程序。代码见 examples/multiplexer

云风在他的博客中提到了网游连接服务器的功能需求[33]，我用 C++ 初步实现了这些需求，并为之编写了配套的自动化 test harness，作为 muduo 网络库的示例。

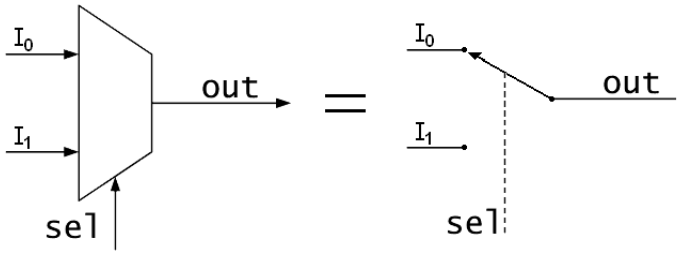
注意：本文呈现的代码仅仅实现了基本的功能需求，没有考虑安全性，也没有特别优化性能，不适合用作真正的放在公网上运行的网游连接服务器。

功能需求

这个连接服务器把多个客户连接汇聚为一个内部 TCP 连接，起到“数据串并转换”的作用，让 backend 的逻辑服务器专心处理业务，而无需顾及多连接的并发性。以下是系统的框图：



这个连接服务器的作用与数字电路中的数据选择器 (multiplexer) 类似，所以我把它命名为 multiplexer。（其实 IO-Multiplexing 也是取的这个意思，让一个 thread-of-control 能有选择地处理多个 IO 文件描述符。）



(上图取自 wikipedia，是 public domain 版权)

实现

Multiplexer 的功能需求不复杂，无非是在 backend connection 和 client connections 之间倒腾数据。对每个新 client connection 分配一个新的整数 id，如果 id 用完

[33] http://blog.codingnow.com/2010/11/go_prime.html 搜“练手项目”

了，则断开新连接（这样通过控制 id 的数目就能控制最大连接数）。另外，为了避免 id 过快地被复用（有可能造成 backend 串话），multiplexer 采用 queue 来管理 free id，每次从队列的头部取 id，用完之后放回 queue 的尾部。具体来说，主要是处理四种事件：

- 当 client connection 到达或断开时，向 backend 发出通知。onClientConnection()
- 当从 client connection 收到数据时，把数据连同 connection id 一同发给 backend。onClientMessage()
- 当从 backend connection 收到数据时，辨别数据是发给哪个 client connection，并执行相应的转发操作。onBackendMessage()
- 如果 backend connection 断开连接，则断开所有 client connections（假设 client 会自动重试）。onBackendConnection()

由上可见，multiplexer 的功能与 proxy 颇为类似。multiplexer_simple.cc 是一个单线程版的实现，借助 muduo 的 IO-multiplexing 特性，可以方便地处理多个并发连接。

在实现的时候有两点值得注意：

TcpConnection 的 id 如何存放？ 当从 backend 收到数据，如何根据 id 找到对应的 client connection？当从 client connection 收到数据，如何得知其 id ？

第一个问题比较好解决，用 std::map〈int, TcpConnectionPtr〉clientConns_; 保存从 id 到 client connection 的映射就行。

第二个问题固然可以用类似的办法解决，但是我想借此介绍一下 muduo::net::TcpConnection 的 context 功能。每个 TcpConnection 都有一个 boost::any 成员，可由客户代码自由支配 (get/set)，代码如下。这个 boost::any 是 TcpConnection 的 context，可以用于保存与 connection 绑定的任意数据（比方说 connection id、connection 的最后数据到达时间、connection 所代表的用户的名字等等）。这样客户代码不必继承 TcpConnection 就能 attach 自己的状态，而且也用不着 TcpConnectionFactory 了（如果允许继承，那么必然要向 TcpServer 注入此 factory）。

muduo/net/TcpConnection.h

```
class TcpConnection : public boost::enable_shared_from_this<TcpConnection>,
                      boost::noncopyable
{
 public:
```

```
  void setContext(const boost::any& context)
  { context_ = context; }

  boost::any& getContext()
  { return context_; }

  const boost::any& getContext() const
  { return context_; }

  // ...

 private:
  // ...
  boost::any context_;
};

typedef boost::shared_ptr<TcpConnection> TcpConnectionPtr;
```

muduo/net/TcpConnection.h

对于 Multiplexer，在 onClientConnection() 里调用 conn->setContext(id)，把 id 存到 TcpConnection 对象中。onClientMessage() 从 TcpConnection 对象中取得 id，连同数据一起发送给 backend，完整实现如下：

```
void onClientMessage(const TcpConnectionPtr& conn, Buffer* buf, Timestamp)
{
  if (!conn->getContext().empty())
  {
    int id = boost::any_cast<int>(conn->getContext());
    sendBackendBuffer(id, buf);
  }
  else
  {
    buf->retrieveAll();
  }
}
```

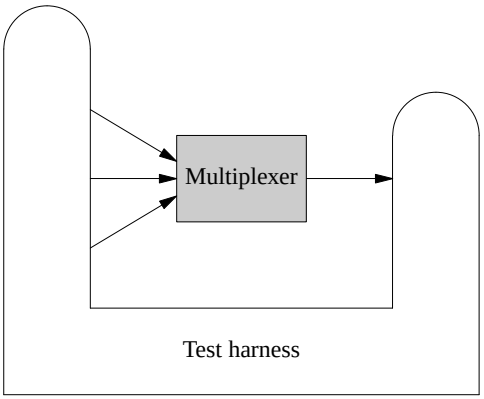
TcpConnection 的生命期如何管理？ 由于 Client Connection 是动态创建并销毁，其生与灭完全由客户决定，如何保证 backend 想向它发送数据的时候，这个 TcpConnection 对象还活着？解决思路是用 reference counting，当然，不用自己写，用 boost::shared_ptr 即可。TcpConnection 是 muduo 中唯一默认采用 shared_ptr 来管理生命期的对象，盖由其动态生命期的本质决定。更多内容请参考《当析构函数遇到多线程——C++ 中线程安全的对象回调》。

multiplexer 是二进制协议，如何测试呢？

2.12.1 自动化测试

Multiplexer 是 muduo 网络编程示例中第一个具有 non-trivial 业务逻辑的网络程序，根据陈硕《分布式程序的自动化回归测试》一文的思想，我为它编写了 test harness。代码见 examples/multiplexer/harness/。

这个 Test harness 采用 Java 编写，用的是 Netty 库。这个 test harness 要扮演 clients 和 backend，也就是既要主动发起连接，也要被动接受连接。而且，test harness 与 multiplexer 的启动顺序是任意的，如何做到这一点请阅读代码。结构如下：



Test harness 会把各种 event 汇聚到一个 blocking queue 里边，方便编写 test case。Test case 则操纵 test harness，发起连接、发送数据、检查收到的数据，例如以下是其中一个 test case testcase/TestOneClientSend.java

这里的几个 test cases 都以用 java 直接写的，如果有必要，也可以采用 Groovy 来编写，这样可以在不重启 test harness 的情况下随时修改添加 test cases。具体做法见陈硕《“过家家”版的移动离线计费系统实现》。

将来的改进

有了这个自动化的 test harness，我们可以比较方便且安全地修改（甚至重新设计）multiplexer。例如

- 增加“backend 发送指令断开 client connection”的功能。有了自动化测试，这个新功能可以被单独测试（指开发者测试），而不需要真正的 backend 参与进来。

- 将 Multiplexer 改用多线程重写。有了自动化回归测试，我们不用担心破坏原有的功能，可以放心大胆地重写。而且由于 test harness 是从外部测试，不是单元测试，重写 multiplexer 的时候不用动 test cases，这样保证了测试的稳定性。另外，这个 test harness 稍作改进还可以进行 stress testing，既可用于验证多线程 multiplexer 的正确性，亦可对比其相对单线程版的效率提升。

2.13 socks4a 代理服务器

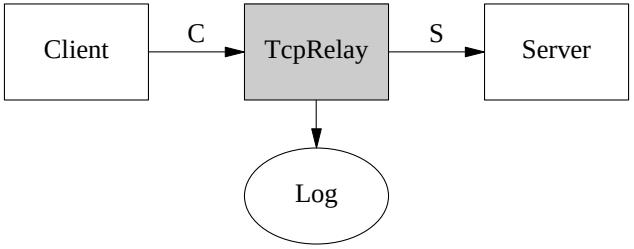
本文介绍用 muduo 实现一个简单的 socks4a 代理服务器，代码见 examples/socks4a/ 。

2.13.1 TCP 中继器

在实现 socks4a proxy 之前，我们先写一个功能更简单的网络程序——TCP 中继器 (TCP relay)，或者叫做穷人的 tcpdump (poor man's tcpdump)。

一般情况下，客户端程序直接连接服务端，如下图。

有时候，我们想在 client 和 server 之间放一个中继器 (relay)，把 client 与 server 之间的通信内容记录下来。这时用 tcpdump 是最方便省事的，但是 tcpdump 需要 root 权限，万一拿不到权限呢？穷人有穷人的办法，自己写一个 relay，让 client 连接 relay，再让 relay 连接 server，如下图中的 T 型结构，relay 扮演了类似 proxy 的角色。



TcpRelay 是我们自己写的，可以动动手脚。除了记录通信内容，还可以制造延时，或者故意翻转 1 bit 数据以模拟 router 硬件故障。

TcpRelay 的功能（业务逻辑）看上去很简单，无非是把连接 C 上收到的数据发给连接 S，同时把连接 S 上收到的数据发给连接 C。但仔细考虑起来，细节其实不那么简单：

1. 建立连接。为了真实模拟 client，TcpRelay 在 accept 连接 C 之后才向 server 发起连接 S，那么在 S 建立起来之前，从 C 收到数据怎么办？要不要暂存起来？
2. 并发连接的管理。上图中只画出了一个 client，实际上 TcpRelay 可以服务多个 clients，左右两边这些并发连接如何管理，如何防止串话 (cross talk)？
3. 连接断开。Client 和 Server 都可能主动断开连接。当 Client 主动断开连接 C 时，TcpRelay 应该立刻断开 S。当 Server 主动断开连接 S 时，TcpRelay 应立刻断开 C。这样才能比较精确地模拟 Client 和 Server 的行为。在关闭连接的刹那，又有新的 client 连接进来，复用了刚刚 close 的 fd 号码，会不会造成串话？万一 Client 和 Server 几乎同时主动断开连接，TcpRelay 如何应对？
4. 速度不匹配。如果连接 C 的带宽是 100KB/s，而连接 S 的带宽是 10MB/s，不巧 Server 是个 chargen 服务，会全速发送数据，那么会不会撑爆 TcpRelay 的 buffer？如何限速？特别是在使用 non-blocking IO 和 level-trigger polling 的时候如何限制读取数据的速度？

在看 muduo 的实现之前，请读者思考：如果用 Sockets API 来实现 TcpRelay，如何解决以上这些问题。

如果用传统多线程阻塞 IO 的方式来实现 TcpRelay，代码如下。功能上没有问题，但是并发度就高不到哪儿去了。

```
#!/usr/bin/python

import socket, thread, time

listen_port = 3007
connect_addr = ('localhost', 2007)
sleep_per_byte = 0.0001

def forward(source, destination):
    source_addr = source.getpeername()
    while True:
        data = source.recv(4096)
        if data:
            for i in data:
                destination.sendall(i)
                time.sleep(sleep_per_byte)
        else:
```

```
            print 'disconnect', source_addr
            destination.shutdown(socket.SHUT_WR)
            break

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
serversocket.bind(('', listen_port))
serversocket.listen(5)

while True:
    (clientsocket, address) = serversocket.accept()
    print 'accepted', address
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(connect_addr)
    print 'connected', sock.getpeername()
    thread.start_new_thread(forward, (clientsocket, sock))
    thread.start_new_thread(forward, (sock, clientsocket))
```

TcpRelay 的实现很简单，只有几十行代码 examples/socks4a/tcprelay.cc，主要逻辑都在 Tunnel class 里 examples/socks4a/tunnel.h 。这个实现很好地解决了前三个问题，第四个问题的解决办法比较粗暴，是用的 high water mark callback，如果发送缓冲区堆积的数据大于 1MB，就断开连接。

2.13.2 Socks4a 代理服务器

Socks4a 的功能与 TcpRelay 非常相似，也是把连接 C 上收到的数据发给连接 S，同时把连接 S 上收到的数据发给连接 C。它与 TcpRelay 的区别在于，TcpRelay 固定连到某个 server 地址，而 socks4a 允许 client 指定要连哪个 server。在 accept 连接 C 之后，Socks4a server 会读几个字节，以了解 server 的地址，再发起连接 S。

Socks4a 的协议非常简单，请参考维基百科 [34] 。

muduo 的 socks4a 代理服务器的实现在 examples/socks4a/socks4a.cc，它也使用了 Tunnel class。与 TcpRelay 相比，只多了解析 server 地址这一步骤。目前 DNS 地址解析这一步用的是阻塞的 gethostbyname() 函数，在真正的系统中，应该换成非阻塞的 DNS 解析，可参考第 2.15 节。

muduo 这个 socks4a 是个标准的网络服务，可以供 Web 浏览器使用（我正是这么测试它的）。

[34] http://en.wikipedia.org/wiki/SOCKS#SOCKS_4a

2.13.3 n:1 与 1:n 连接转发

云风在《写了一个 proxy 用途你懂的》[35] 中写了一个 TCP 隧道 tunnel，程序由三部分组成：n:1 连接转发服务，1:n 连接转发服务，socks 代理服务。

我仿照他的思路，用 muduo 实现了这三个程序。不同的是，我没有做数据混淆，所以不能用来翻传说中的墙。

- n:1 连接转发服务就是前一节中的 multiplexer (数据选择器)。
- 1:n 连接转发服务是该文提到的 backend，一个数据分配器 (demultiplexer)，代码在 examples/multiplexer/demux.cc
- socks 代理服务正是本文实现的 socks4a。

有兴趣的读者可以把这三个程序级联起来试一试。

2.14 短址服务

muduo 内置了一个简陋的 http 服务器，可以处理简单的 http 请求。这个 http 服务器是面向内网的暴露进程状态的监控端口，不是面向公网的功能完善且健壮的 httpd。不过我们可以拿它来实现一个简单的短 URL 转发服务，以简单说明其用法。代码位于 examples/shorturl/shorturl.cc，接口与 J2EE 的 HttpServlet 有几分类似。

examples/shorturl/shorturl.cc

```
std::map<string, string> redirections;

void onRequest(const HttpRequest& req, HttpResponse* resp)
{
  LOG_INFO << "Headers " << req.methodString() << " " << req.path();

  // TODO: support PUT and DELETE to create new redirections on-the-fly.

  std::map<string, string>::const_iterator it = redirections.find(req.path());
  if (it != redirections.end())
  {
    resp->setStatusCode(HttpResponse::k301MovedPermanently);
    resp->setStatusMessage("Moved Permanently");
    resp->addHeader("Location", it->second);
    // resp->setCloseConnection(true);
  }
  // ...
}
```

[35] http://blog.codingnow.com/2011/05/xtunnel.html

```
int main()
{
  redirections["/1"] = "http://chenshuo.com";
  redirections["/2"] = "http://blog.csdn.net/Solstice";

  EventLoop loop;
  HttpServer server(&loop, InetAddress(8000), "shorturl");
  server.setHttpCallback(onRequest);
  server.start();
  loop.loop();
}
```

examples/shorturl/shorturl.cc

需要说明的是，muduo 不是为短连接 TCP 服务优化，无法发挥多核优势。

2.15 与其他库集成

待续。

2.15.1 UDNS

2.15.2 c-ares DNS

2.15.3 curl

2.15.4 更多

microhttp, libpg, libdrizzle, quickfix