第 6 章

muduo 网络库简介

6.1 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：非阻塞 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。
- 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）。
- 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)`。
- 以上条件都满足时，可以考虑搭配 Google Protocol Buffers RPC。

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

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）[2]，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 11 月）的最新版本是 0.8.2。

为什么需要网络库

使用 Sockets API 进行网络编程是很容易上手的一项技术，花半天时间读完一两篇网上教程，相信不难写出能相互连通的网络程序。例如下面这个网络服务端和客户端程序，它用 Python 实现了一个简单的“Hello”协议，客户端发来姓名，服务端返回问候语和服务器的当前时间。

hello-server.py

```
#!/usr/bin/python

import socket, time

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('', 8888))
serversocket.listen(5)

while True:
    (clientsocket, address) = serversocket.accept()  # 等待客户端连接
    data = clientsocket.recv(4096)                   # 接收姓名
    datetime = time.asctime()+'\n'
    clientsocket.send('Hello ' + data)               # 发回问候
    clientsocket.send('My time is ' + datetime)      # 发送服务器当前时间
    clientsocket.close()                             # 关闭连接
```

hello-server.py

hello-client.py

```
# 省略 import 等
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((sys.argv[1], 8888))   # 服务器地址由命令行指定
sock.send(os.getlogin() + '\n')     # 发送姓名
message = sock.recv(4096)           # 接收响应
print message                       # 打印结果
sock.close()                        # 关闭连接
```

hello-client.py

上面两个程序使用了全部主要的 Sockets API，包括 `socket(2)`、`bind(2)`、`listen(2)`、`accept(2)`、`connect(2)`、`recv(2)`、`send(2)`、`close(2)`、`gethostbyname(3)` [3] 等，似乎网络编程一点也不难嘛。在同一台机器上运行上面的服务端和客户端，结果不出意料：

[2] 这个名字的由来见我的一篇访谈：http://www.oschina.net/question/28_61182。

[3] 代码中没有显式调用，而是在 *L22* 隐式调用。

```
$ ./hello-client.py localhost
Hello schen
My time is Sun May 13 12:56:44 2012
```

但是连接同一局域网的另外一台服务器时，收到的数据是不完整的。错在哪里？

```
$ ./hello-client.py atom
Hello schen
```

出现这种情况的原因是高级语言（Java、Python 等）的 Sockets 库并没有对 Sockets API 提供更高层的封装，直接用它编写网络程序很容易掉到陷阱里，因此我们需要一个好的网络库来降低开发难度。网络库的价值还在于能方便地处理并发连接（§6.6）。

6.2 安装

源文件 tar 包的下载地址：http://code.google.com/p/muduo/downloads/list，此处以 muduo-0.8.2-beta.tar.gz 为例。

muduo 使用了 Linux 较新的系统调用（主要是 `timerfd` 和 `eventfd`），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-bit 和 64-bit 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 [4]。

如果要在较旧的 Linux 2.6 内核 [5] 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）和 Raspberry Pi（ARM11）上成功运行了 muduo 的多个示例。代码只需略作改动，请参考 armlinux.diff。

muduo 采用 CMake [6] 为 build system，安装方法如下：

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

muduo 依赖于 Boost [7]，也很容易安装：

```
$ sudo apt-get install libboost-dev libboost-test-dev
```

[4] http://aur.archlinux.org/packages.php?ID=49251

[5] 例如 Debian 5.0 Lenny、Ubuntu 8.04、CentOS 5 等旧的发行版。

[6] 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。

[7] 核心库只依赖 TR1，示例代码用到了其他 Boost 库。

muduo 有三个非必需的依赖库：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。安装方法如下：

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

muduo 的编译方法很简单：

```
$ tar zxf muduo-0.8.2-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}，
以便 muduo-protorpc 和 muduo-udns 等库使用
```

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}，
以便 muduo-protorpc 和 muduo-udns 等库使用
```

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。

在自己的程序中使用 muduo

muduo 是静态链接[8]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 makefile 编译基于 muduo 的程序：https://github.com/chenshuo/muduo-tutorial。

[8] 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 11 章。

6.3 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（生产者消费者队列）
    |-- BoundedBlockingQueue.h    有界阻塞队列
    |-- Condition.h               条件变量，与 Mutex.h 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（§9.5）。

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的 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}
```

6.3.1 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
|           |-- Channel.h
|           |-- Endian.h
|           |-- EventLoop.h
|           |-- EventLoopThread.h
|           |-- InetAddress.h
|           |-- TcpClient.h
|           |-- TcpConnection.h
|           |-- TcpServer.h
|           |-- TimerId.h
|           |-- http                以下为网络附属库的头文件
|           |   |-- 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
```

图 6-1 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。

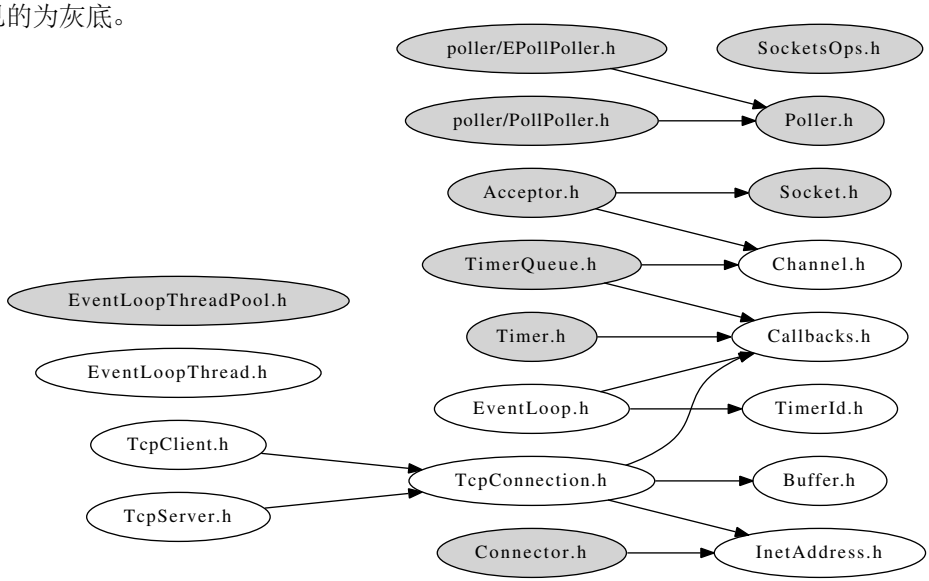


图 6-1

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，详细的介绍参见后文。

公开接口

- Buffer 仿 Netty ChannelBuffer 的 buffer class，数据的读写通过 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（§7.4）。

- `InetAddress` 封装 IPv4 地址（end point），注意，它不能解析域名，只认 IP 地址。因为直接用 `gethostbyname(3)` 解析域名会阻塞 IO 线程。
- `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，注意它不能发起连接。
- `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，注意它不拥有 file descriptor。它是 `Acceptor`、`Connector`、`EventLoop`、`TimerQueue`、`TcpConnection` 的成员，生命期由后者控制。
- `Socket` 是一个 RAII handle，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` class。
- `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。
- `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 $O(\log N)$，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。

图 6-2 是 muduo 的简化类图，`Buffer` 是 `TcpConnection` 的成员。

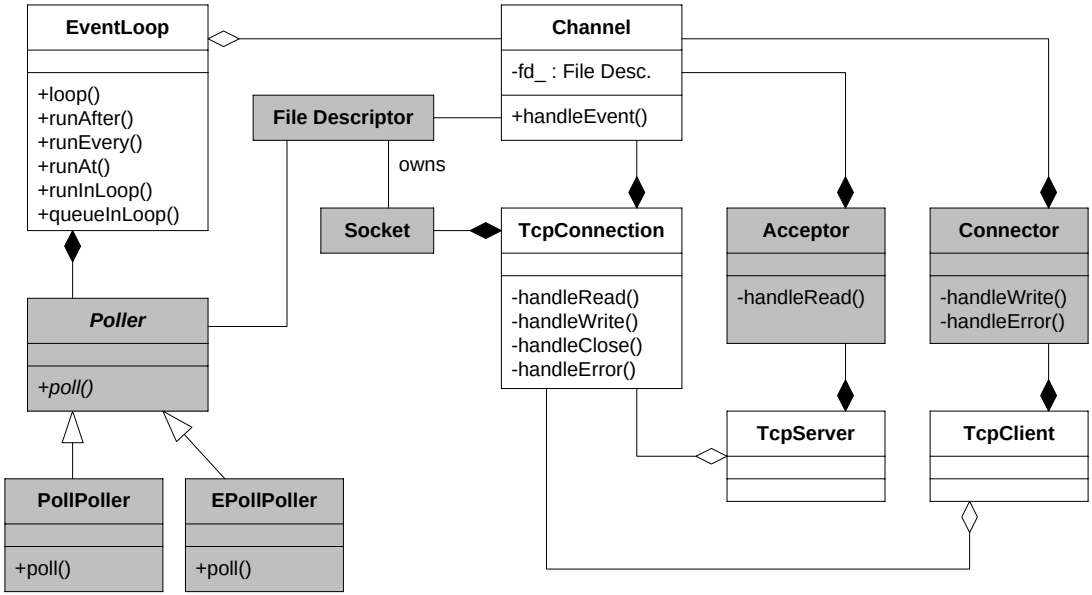


图 6-2

6.3.2 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。这些例子基本覆盖了常见的服务端网络编程功能点，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          5 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。[9]

6.3.3 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(2)` 与 `TcpConnection` 用同一个线程做 IO。
- 多线程，`accept(2)` 与 `EventLoop` 在同一个线程，另外创建一个 `EventLoopThreadPool`，新到的连接会按 round-robin 方式分配到线程池中。

后文 §6.6 还会以 Sudoku 服务器为例再次介绍 muduo 的多线程模型。

[9] 注意，目前 muduo-protorpc 与 Ubuntu Linux 12.04 中通过 `apt-get` 安装的 Protobuf 编译器无法配合，请从源码编译安装 Protobuf 2.4.1。

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 [10]），待日后慢慢改进吧。

6.4 使用教程

本节主要介绍 muduo 网络库的使用，其设计与实现将在第 8 章讲解。

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 [11]，我希望 muduo 能减少网络编程中的偶发复杂性（accidental complexity）。

6.4.1 TCP 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。

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`、`shutdown`）和被动断开（`read(2)` 返回 0）。

[10] Signal 也可以通过 `signalfd(2)` 融入 `EventLoop` 中，见 muduo-protorpc 中的 zurg slave 例子。

[11] http://www.cs.nott.ac.uk/~cah/G51ISS/Documents/NoSilverBullet.html

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，这里的“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需的，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 地）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）？[12] 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 40kB 数据，但是操作系统的 TCP 发送缓冲区只有 25kB 剩余空间，那么剩下的 15kB 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 15kB 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。如果应用程序随后又要发送 50kB 数据，而此时发送缓冲区中尚有未发送的数据（若干 kB），那么网络库应该将这 50kB 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 [13]。假如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 [14]。

[12] 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。

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

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

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，我们希望减少内存占用。如果有 10 000 个并发连接，每个连接一建立就分配各 50kB 的读写缓冲区 (s) 的话，将占用 1GB 内存，而大多数时候这些缓冲区的使用率很低。muduo 用 `readv(2)` 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。

6.4.2 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。

下面以经典的 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_;
23   muduo::net::TcpServer server_;
24 };
```

examples/simple/echo/echo.h

在构造函数里注册回调函数。

examples/simple/echo/echo.cc

```
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()`。

examples/simple/echo/echo.cc

```
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->retrieveAllAsString());
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

$\mathcal{L}$37 和 $\mathcal{L}$40 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 $\mathcal{L}$40 的 `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。

6.4.3 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();
  }
}

int main()
{
  EventLoop loop;
  TcpServer server(&loop, InetAddress(1079), "Finger");
  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。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（£30），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 £39。

```
--- 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(1)` 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。

6.5 性能评测

我在一开始编写 muduo 的时候并没有以高性能为首要目标。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。我很惊奇地发现，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。

注意这里的测试只是简单地比较了平均值；其实在严肃的性能对比中至少还应该考虑分布和百分位数（percentile）的值 [15] [16]。限于篇幅，此处从略。

6.5.1 muduo 与 Boost.Asio、libevent2 的吞吐量对比

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 的吞吐量比 Boost.Asio 高 15% 以上；比 libevent2 高 18% 以上，个别情况甚至达到 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/，未做更改。
- 我自己编写了 libevent2 的 ping pong 测试代码，路径是 recipes/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 [17] 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
```

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 16GiB。

软件：操作系统为 Ubuntu Linux Server 10.04.1 LTS x86_64，编译器是 g++ 4.4.3。

[15] http://zedshaw.com/essays/programmer_stats.html

[16] http://www.percona.com/files/presentations/VELOCITY2012-Beyond-the-Numbers.pdf

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

测试方法

依据 asio 性能测试[18] 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 1/10/100/1000/10 000 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。）

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

在同一台机器测试吞吐量的原因如下：

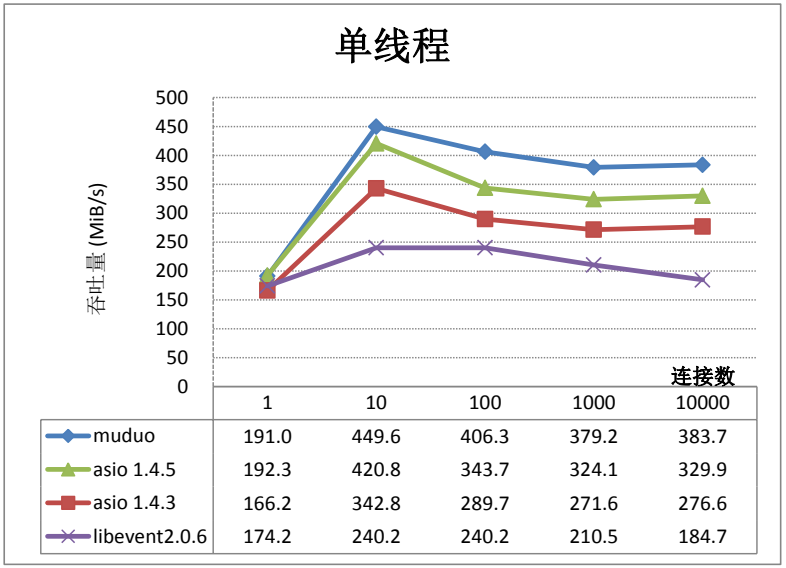
现在的 CPU 很快，即便是单线程单 TCP 连接也能把千兆以太网的带宽跑满。如果用两台机器，所有的吞吐量测试结果都将是 110MiB/s，失去了对比的意义。（用 Python 也能跑出同样的吞吐量，或许可以对比哪个库占的 CPU 少。）

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。也就是说在单线程下，服务端和客户端各占满 1 个 CPU，比较哪个库的吞吐量高。

测试结果

单线程测试的结果（见图 6-3），数字越大越好。

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

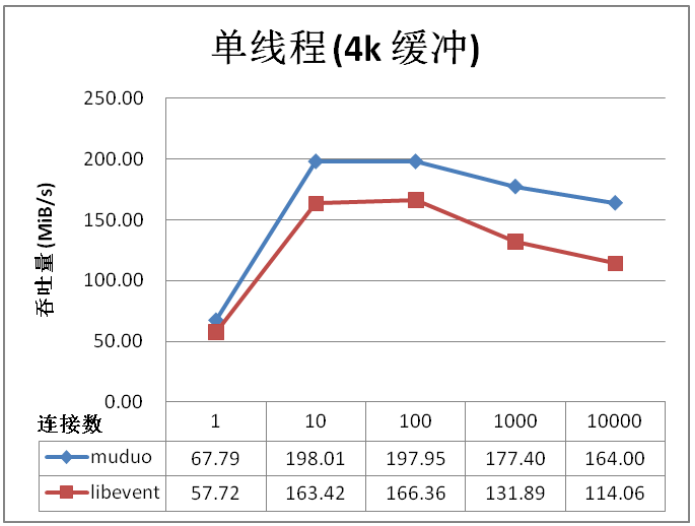


	1	10	100	1000	10000
muduo	191.0	449.6	406.3	379.2	383.7
asio 1.4.5	192.3	420.8	343.7	324.1	329.9
asio 1.4.3	166.2	342.8	289.7	271.6	276.6
libevent2.0.6	174.2	240.2	240.2	210.5	184.7

图 6-3

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 16 384 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（见图 6-4）。



	1	10	100	1000	10000
muduo	67.79	198.01	197.95	177.40	164.00
libevent	57.72	163.42	166.36	131.89	114.06

图 6-4

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

多线程测试的结果（见图 6-5），数字越大越好。

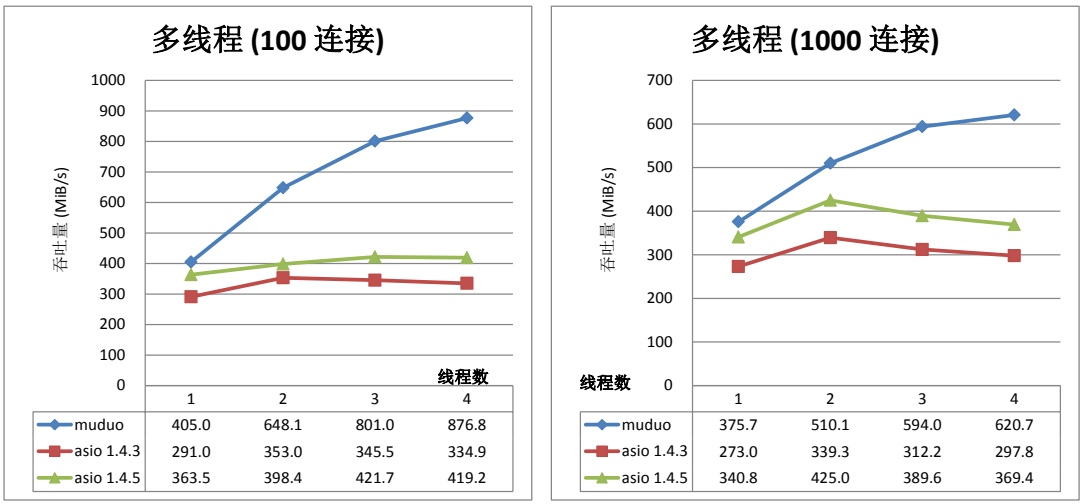


图 6-5

测试结果表明 muduo 的吞吐量平均比 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，其性能应该有所提高。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，这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。

6.5.2 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 并不是为高吞吐量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 笔记本电脑上运行了测试，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)` 或 `pipe(2)`），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 ~ 100 000（10 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 256 000。

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 [19]，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[20] 是 libev 的作者 Marc Lehmann 写的，我略做改用，用于测试 muduo 和 libevent2。

第一轮的结果（见图 6-6），请先只看“+”线（实线）和“×”线（粗虚线）。“×”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。

[19] http://libev.schmorp.de/bench.html

[20] recipes/pingpong/libevent/run_bench.sh

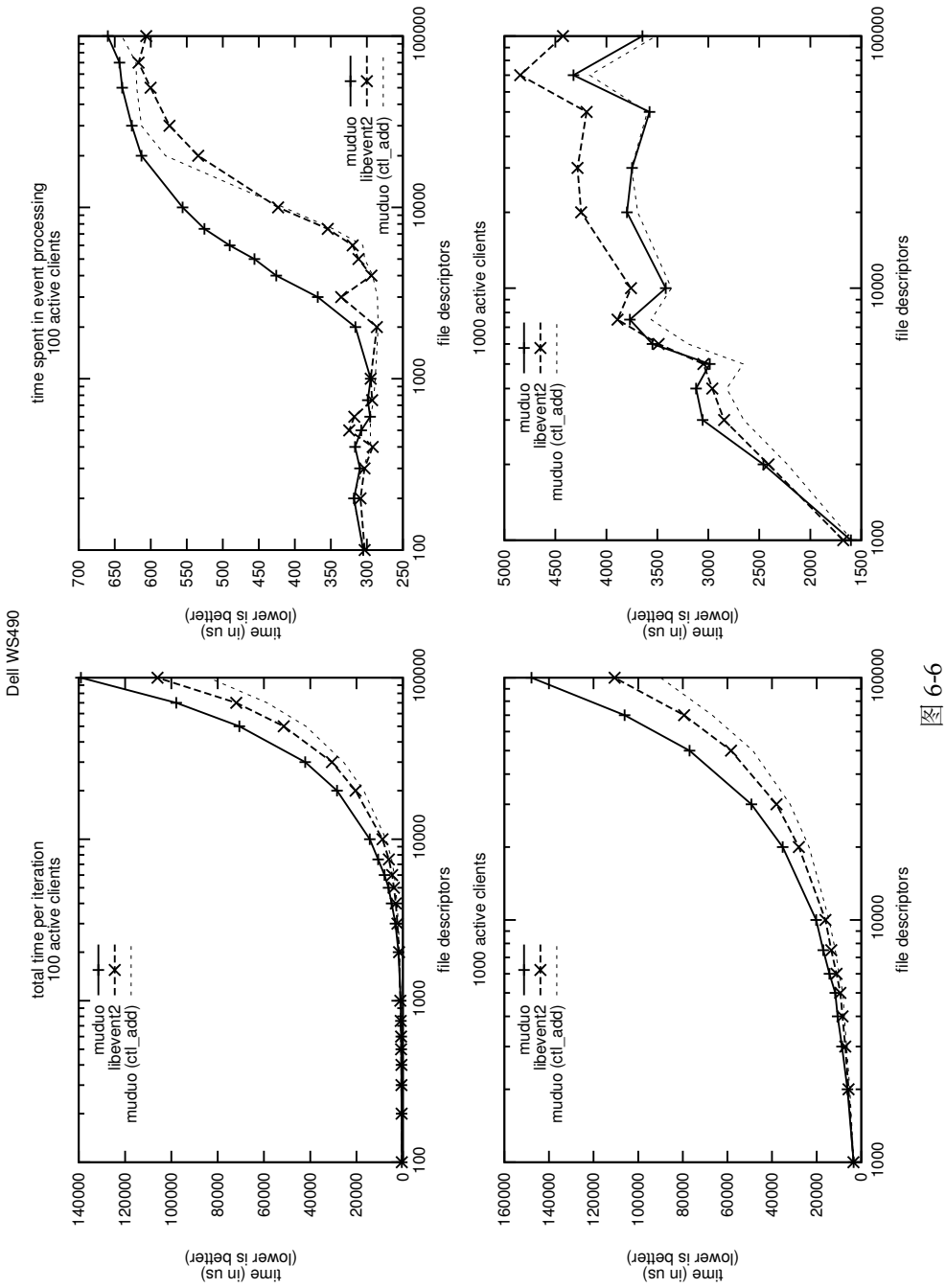

Dell WS490
total time per iteration
100 active clients
time spent in event processing
100 active clients
1000 active clients
1000 active clients
time (in us)
(lower is better)
file descriptors
muduo
libevent2
muduo (ctl_add)

图 6-6

从两条线的对比可以看出：

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 30 000 时，二者性能差不多；
      当总连接数大于 1000 或小于 30 000 时，libevent2 明显领先。
   b. 在 1000 个活动连接的情况下，
      当并发数小于 10 000 时，libevent2 和 muduo 得分接近；
      当并发数大于 10 000 时，muduo 明显占优。

这里有两个问题值得探讨：

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[21]，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。

第二轮测试结果见图 6-6 的细虚线，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，结果如图 6-7 所示。（我的笔记本电脑的 CPU 主频是 2.4 GHz，高于台式机的 1.86 GHz，所以用时较少。）

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

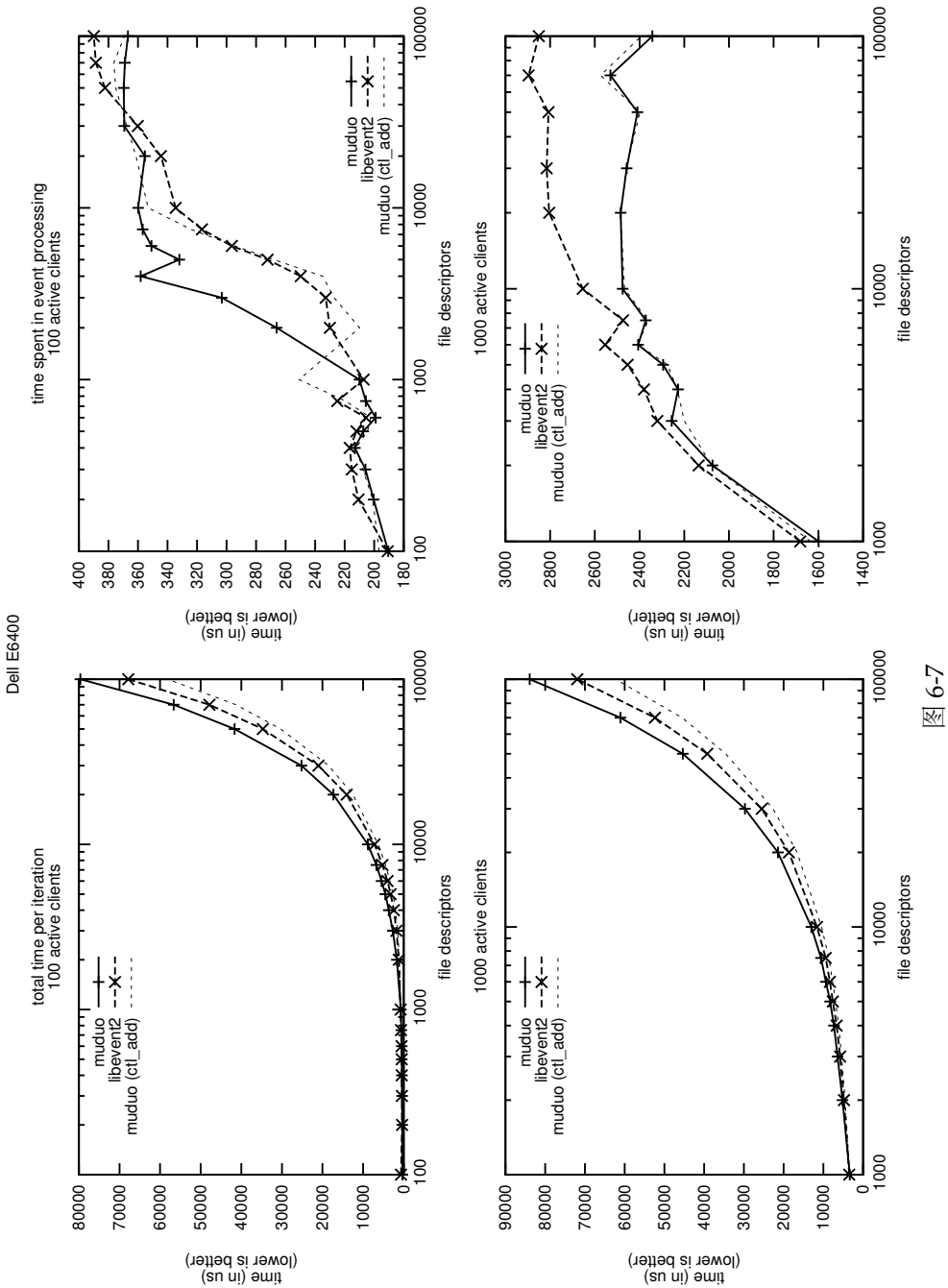

Dell E6400
total time per iteration
100 active clients
time spent in event processing
100 active clients
1000 active clients
1000 active clients
muduo
libevent2
muduo (ctl_add)
time (in us)
(lower is better)
file descriptors

图 6-7

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 10 000），muduo 略微占优。

6.5.3 muduo 与 Nginx 的吞吐量对比

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 [22] 和 weighttp [23]，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，这里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;
```

[22] http://httpd.apache.org/docs/2.4/programs/ab.html

[23] 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 [24] 来实现直接返回数据。配置文件如下：

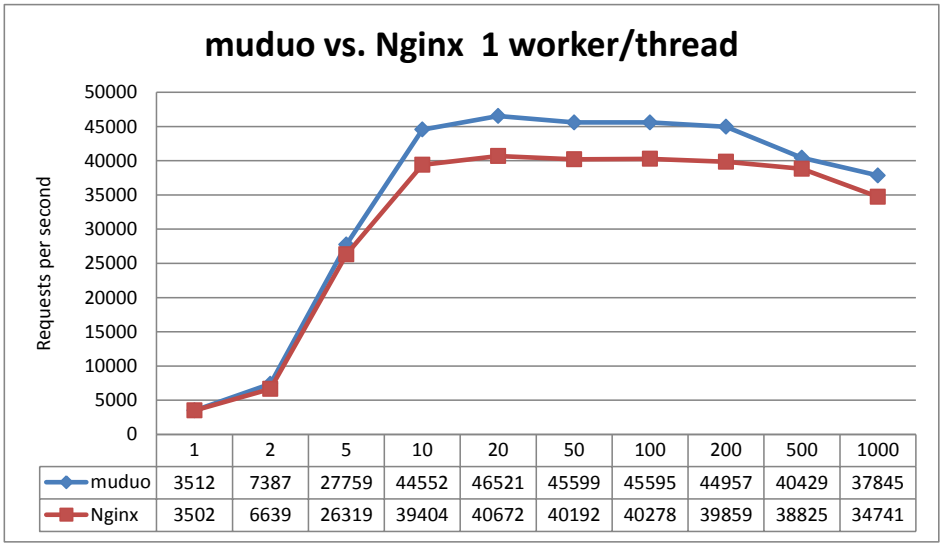
```
#user  nobody;
worker_processes  4;

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!";
        }
    }
}
```

[24] http://wiki.nginx.org/HttpEchoModule，配置文件 https://gist.github.com/1967026。

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
```

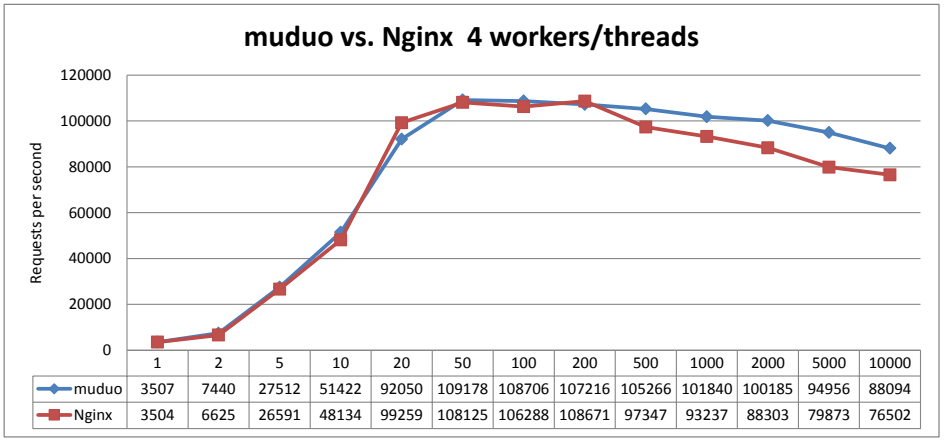
先测试单线程的性能（见图 6-8），横轴是并发连接数，纵轴为每秒完成的 HTTP 请求响应数目，下同。在测试期间，ab 的 CPU 使用率低于 70%，客户端游刃有余。



	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

图 6-8

再对比 muduo 4 线程和 Nginx 4 工作进程的性能（见图 6-9）。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

图 6-9

CPU 使用率对比（百分比是 top(1) 显示的数值）：

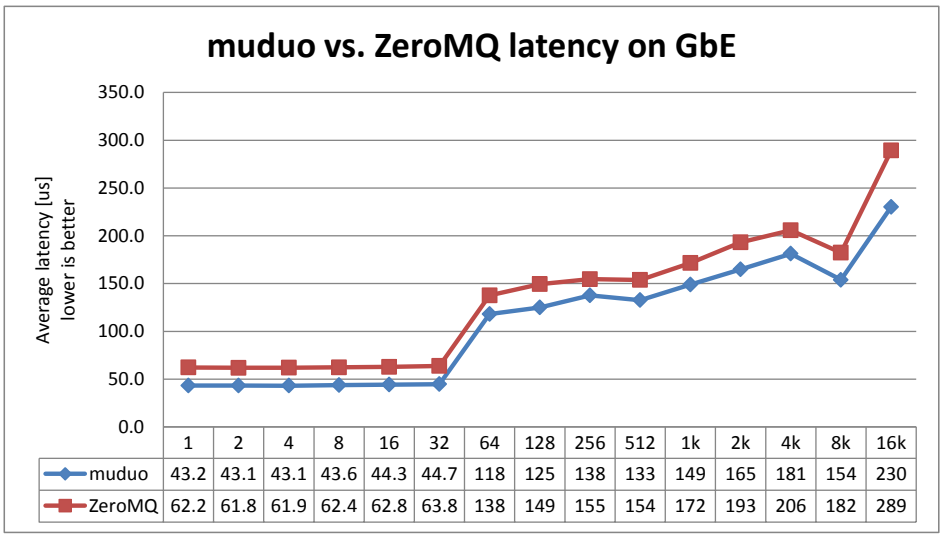
- 10 000 并发连接，4 workers/threads，muduo 是 4 × 83%，Nginx 是 4 × 75%
- 1000 并发连接，4 workers/threads，muduo 是 4 × 85%，Nginx 是 4 × 78%

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（每秒请求数）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。

6.5.4 muduo 与 ZeroMQ 的延迟对比

本节我们用 ZeroMQ 自带的延迟和吞吐量测试 [25] 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 §6.5.1 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。测试结果如图 6-10 所示，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 16KiB 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

图 6-10

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

6.6 详解 muduo 多线程模型

本节以一个 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。本节代码参见：examples/sudoku/。

6.6.1 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”（§9.8）、“muduo `Buffer` 类的设计与使用”（§7.4）、“‘多线程服务器的适用场合’例释与答疑”（§3.6）等处，它也可以看成是 echo 服务的一个变种（附录 A “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）》[26]，或者见我写的《分布式系统的工程化开发方法》[27] 第 54 页关于 out-of-order RPC 的介绍。

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

[27] http://blog.csdn.net/solstice/article/details/5950190

<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。

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。Sudoku Solver 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）》[28] 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 §6.4.2 “echo 服务的实现”中出现的 EchoServer 为蓝本，稍加修改就能得到 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。这个函数应该能正确处理 TCP 分包。

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

examples/sudoku/server_basic.cc

```
const int kCells = 81;  // 81 个格子

void onMessage(const TcpConnectionPtr& conn, Buffer* buf, Timestamp)
{
  LOG_DEBUG << conn->name();
  size_t len = buf->readableBytes();
  while (len >= kCells + 2)  // 反复读取数据，2 为回车换行字符
  {
    const char* crlf = buf->findCRLF();
    if (crlf)  // 如果找到了一条完整的请求
    {
      string request(buf->peek(), crlf);  // 取出请求
      string id;
      buf->retrieveUntil(crlf + 2);       // retrieve 已读取的数据
      string::iterator colon = find(request.begin(), request.end(), ':');
      if (colon != request.end())  // 如果找到了 id 部分
      {
        id.assign(request.begin(), colon);
        request.erase(request.begin(), colon+1);
      }
      if (request.size() == implicit_cast<size_t>(kCells))  // 请求的长度合法
      {
        string result = solveSudoku(request);  // 求解数独，然后发回响应
        if (id.empty())
        {
          conn->send(result+"\r\n");
        }
        else
        {
          conn->send(id+":"+result+"\r\n");
        }
      }
      else  // 非法请求，断开连接
      {
        conn->send("Bad Request!\r\n");
        conn->shutdown();
      }
    }
    else  // 请求不完整，退出消息处理函数
    {
      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 §7.4 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。

6.6.2 常见的并发网络服务程序设计方案

W. Richard Stevens 的《UNIX 网络编程（第 2 版）》第 27 章“Client-Server Design Alternatives”介绍了十来种当时（20 世纪 90 年代末）流行的编写并发网络程序的方案。[UNP] 第 3 版第 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 已经进行了相当全面的总结，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

表 6-1 是笔者总结的 12 种常见方案。其中“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 也是附录 A 中举的三大 TCP 网络编程案例之一）。对于 echo/httpd/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 类服务却至关重要。“顺序性”指的是在 httpd/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）。

表 6-1

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	特点
0	accept+read/write	0	否	否	是	否	否	无	否	低	否	是	常	一次服务一个客户
1	accept+fork	1	是	否	是	否	是	低	是	高	否	是	变	process-per-connection
2	accept+thread	6	否	是	是	否	是	中	是	中	是	是	变	thread-per-connection
3	prefork	2/3/4/5	是	否	是	否	是	低	是	高	否	是	变	见[UNP]
4	pre threaded	7/8	否	是	是	否	是	中	是	中	是	是	变	见[UNP]
5	poll (reactor)	6.8 节	否	否	否	是	是	高	否	低	是	是	常	单线程 reactor
6	reactor + thread-per-task	无	否	是	否	是	是	中	是	中	是	否	变	thread-per-request
7	reactor + worker thread	无	否	是	否	是	是	中	是	中	是	是	变	worker-thread-per-connection
8	reactor + thread poll	无	否	是	否	是	是	高	是	低	是	否	常	主线程 IO，工作线程计算
9	reactors in threads	无	否	是	否	是	是	高	是	低	是	是	常	one loop per thread
10	reactors in processes	无	是	否	否	是	是	高	是	低	否	是	常	Nginx
11	reactors + thread pool	无	否	是	否	是	是	高	是	低	是	否	常	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。方案 6 和方案 7 其实不是实用的方案，只是作为过渡品。方案 8 和方案 9 是本文重点介绍的方案，其实这两个方案已经在 §3.3 “多线程服务器的常用编程模型”中提到过，只不过当时没有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()`: 534.7μs。
- `pthread_create()+pthread_join()`: 42.5μs，其中创建线程用了 26.1μs。
- push/pop a blocking queue : 11.5μs。
- Sudoku resolve: 100us （根据题目难度不同，浮动范围 20~200μs）。

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] 中的 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，倒是很适合 daytime 这种 write-only 短连接服务。以下 Python 代码展示用方案 0 实现 echo server 的大致做法（本章的 Python 代码均没有考虑错误处理）：

recipes/python/echo-iterative.py

```
3  import socket
4
5  def handle(client_socket, client_address):
6      while True:
7          data = client_socket.recv(4096)
8          if data:
9              sent = client_socket.send(data)    # sendall?
10         else:
11             print "disconnect", client_address
12             client_socket.close()
13             break
14
15 if __name__ == "__main__":
16     listen_address = ("0.0.0.0", 2007)
17     server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
18     server_socket.bind(listen_address)
19     server_socket.listen(5)
20
21     while True:
22         (client_socket, client_address) = server_socket.accept()
23         print "got connection from", client_address
24         handle(client_socket, client_address)
```

recipes/python/echo-iterative.py

ℒ6~ℒ13 是 echo 服务的“业务逻辑循环”，从 ℒ21~ℒ24 可以看出它一次只能服务一个客户连接。后面列举的方案都是在保持这个循环的功能不变的情况下，设法能高

效地同时服务多个客户端。L9 代码值得商榷，或许应该用 sendall() 函数，以确保完整地发回数据。

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。

Python 示例如下，注意其中 L9~L16 正是前面的业务逻辑循环，self.request 代替了前面的 client_socket。ForkingTCPServer 会对每个客户连接新建一个子进程，在子进程中调用 EchoHandler.handle()，从而同时服务多个客户端。在这种编程方式中，业务逻辑已经初步从网络框架分离出来，但是仍然和 IO 紧密结合。

recipes/python/echo-fork.py

```
1  #!/usr/bin/python
2
3  from SocketServer import BaseRequestHandler, TCPServer
4  from SocketServer import ForkingTCPServer, ThreadingTCPServer
5
6  class EchoHandler(BaseRequestHandler):
7      def handle(self):
8          print "got connection from", self.client_address
9          while True:
10             data = self.request.recv(4096)
11             if data:
12                 sent = self.request.send(data)    # sendall?
13             else:
14                 print "disconnect", self.client_address
15                 self.request.close()
16                 break
17
18 if __name__ == "__main__":
19     listen_address = ("0.0.0.0", 2007)
20     server = ForkingTCPServer(listen_address, EchoHandler)
21     server.serve_forever()
```

recipes/python/echo-fork.py

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 网络服务多采用这种方案。它的初始化开销比方案 1 要小很多，但与求解 Sudoku 的用时差不多，仍然不适合短连接服务。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。

Python 示例如下，只改动了一行代码。ThreadingTCPServer 会对每个客户连接新建一个线程，在该线程中调用 EchoHandler.handle()。

```
$ diff -U2 echo-fork.py echo-thread.py
 if __name__ == "__main__":
     listen_address = ("0.0.0.0", 2007)
-    server = ForkingTCPServer(listen_address, EchoHandler)
+    server = ThreadingTCPServer(listen_address, EchoHandler)
     server.serve_forever()
```

这里再次体现了将“并发策略”与业务逻辑（`EchoHandler.handle()`）分离的思路。用同样的思路重写方案 0 的代码，可得到：

```
$ diff -U2 echo-fork.py echo-single.py
 if __name__ == "__main__":
     listen_address = ("0.0.0.0", 2007)
-    server = ForkingTCPServer(listen_address, EchoHandler)
+    server = TCPServer(listen_address, EchoHandler)
     server.serve_forever()
```

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(2)` “惊群”问题（thundering herd）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。方案 3 和方案 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，程序流程（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 b 收到的数据发给 a，那么到底读哪个？（proxy 是附录 A 讲的三大 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。§7.13 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 §7.4 。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）。

这里先用一小段 Python 代码简要地回顾“以 IO multiplexing 方式实现并发 echo server”的基本做法[29]。为了简单起见，以下代码并没有开启 non-blocking，也没有考虑数据发送不完整（$\mathcal{L}$28）等情况。首先定义一个从文件描述符到 socket 对象的映射（$\mathcal{L}$14），程序的主体是一个事件循环（$\mathcal{L}$15~$\mathcal{L}$32），每当有 IO 事件发生时，就针对不同的文件描述符（fileno）执行不同的操作（$\mathcal{L}$16, $\mathcal{L}$17）。对于 listening fd，接受（accept）新连接，并注册到 IO 事件关注列表（watch list），然后把连接添加到 connections 字典中（$\mathcal{L}$18~$\mathcal{L}$23）。对于客户连接，则读取并回显数据，并处理连接的关闭（$\mathcal{L}$24~$\mathcal{L}$32）。对于 echo 服务而言，真正的业务逻辑只有 $\mathcal{L}$28：将收到的数据原样发回客户端。

recipes/python/echo-poll.py

```
6  server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7  server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
8  server_socket.bind(('', 2007))
9  server_socket.listen(5)
10 # server_socket.setblocking(0)
11 poll = select.poll() # epoll() should work the same
12 poll.register(server_socket.fileno(), select.POLLIN)
13
14 connections = {}
15 while True:
16     events = poll.poll(10000)  # 10 seconds
17     for fileno, event in events:
18         if fileno == server_socket.fileno():
19             (client_socket, client_address) = server_socket.accept()
20             print "got connection from", client_address
21             # client_socket.setblocking(0)
22             poll.register(client_socket.fileno(), select.POLLIN)
23             connections[client_socket.fileno()] = client_socket
24         elif event & select.POLLIN:
25             client_socket = connections[fileno]
26             data = client_socket.recv(4096)
27             if data:
28                 client_socket.send(data) # sendall() partial?
29             else:
30                 poll.unregister(fileno)
31                 client_socket.close()
32                 del connections[fileno]
```

recipes/python/echo-poll.py

注意以上代码不是功能完善的 IO multiplexing 范本，它没有考虑错误处理，也

[29] 这个例子参照了 http://scotdoyle.com/python-epoll-howto.html#async-examples。

没有实现定时功能，而且只适合侦听（listen）一个端口的网络服务程序。如果需要侦听多个端口，或者要同时扮演客户端，那么代码的结构需要推倒重来。

这个代码骨架可用于实现多种 TCP 服务器。例如写一个聊天服务只需改动 3 行代码，如下所示。业务逻辑是 £28~£30：将本连接收到的数据转发给其他客户连接。

```
$ diff echo-poll.py chat-poll.py -U4
--- echo-poll.py  2012-08-20 08:50:49.000000000 +0800
+++ chat-poll.py  2012-08-20 08:50:49.000000000 +0800
          elif event & select.POLLIN:
              clientsocket = connections[fileno]
              data = clientsocket.recv(4096)
              if data:
-                 clientsocket.send(data) # sendall() partial?
+                 for (fd, othersocket) in connections.iteritems():
+                     if othersocket != clientsocket:
+                         othersocket.send(data) # sendall() partial?
              else:
                  poll.unregister(fileno)
                  clientsocket.close()
                  del connections[fileno]
```

但是这种把业务逻辑隐藏在一个大循环中的做法其实不利于将来功能的扩展，我们能不能设法把业务逻辑抽取出来，与网络基础代码分离呢？

Doug Schmidt 指出，其实网络编程中有很多是事务性（routine）的工作，可以提取为公用的框架或库，而用户只需要填上关键的业务逻辑代码，并将回调注册到框架中，就可以实现完整的网络服务，这正是 Reactor 模式的主要思想。

如果用传统 Windows GUI 消息循环来做一个类比，那么我们前面展示 IO multiplexing 的做法相当于把程序的全部逻辑都放到了窗口过程（WndProc）的一个巨大的 switch-case 语句中，这种做法无疑是不利于扩展的。（各种 GUI 框架在此各显神通。）

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  switch (message)
  {
    case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
    // many more cases
  }
  return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

而 Reactor 的意义在于将消息（IO 事件）分发到用户提供的处理函数，并保持网络部分的通用代码不变，独立于用户的业务逻辑。

单线程 Reactor 的程序执行顺序如图 6-11 （左图）所示。在没有事件的时候，线程等待在 `select/poll/epoll_wait` 等函数上。事件到达后由网络库处理 IO，再把消息通知（回调）客户端代码。Reactor 事件循环所在的线程通常叫 IO 线程。通常由网络库负责读写 socket，用户代码负载解码、计算、编码。

注意由于只有一个线程，因此事件是顺序处理的，一个线程同时只能做一件事情。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（见图 6-11 的右图）。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。

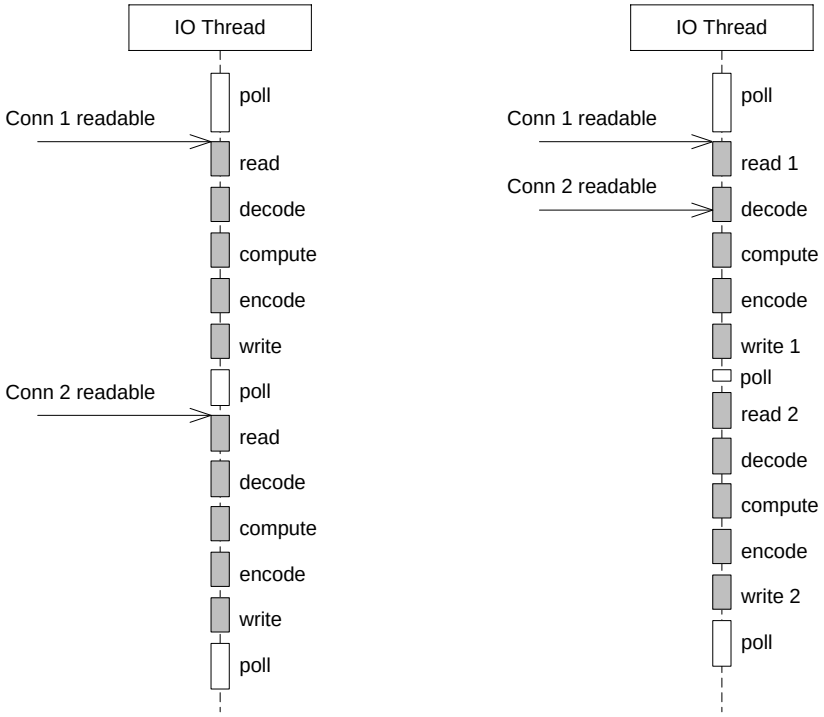


图 6-11

方案 5 基本的单线程 Reactor 方案（见图 6-11），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。

这里用一小段 Python 代码展示 Reactor 模式的雏形。为了节省篇幅，这里直接使用了全局变量，也没有处理异常。程序的核心仍然是事件循环（L42~L46），与前面不同的是，事件的处理通过 handlers 转发到各个函数中，不再集中在一坨。例如 listening fd 的处理函数是 handle_accept，它会注册客户连接的 handler。普通客户连接的处理函数是 handle_request，其中又把连接断开和数据到达这两个事件分开，后者由 handle_input 处理。业务逻辑位于单独的 handle_input 函数，实现了分离。

recipes/python/echo-reactor.py

```
6  server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7  server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
8  server_socket.bind(('', 2007))
9  server_socket.listen(5)
10 # serversocket.setblocking(0)
11
12 poll = select.poll() # epoll() should work the same
13 connections = {}
14 handlers = {}
15
16 def handle_input(socket, data):
17     socket.send(data) # sendall() partial?
18
19 def handle_request(fileno, event):
20     if event & select.POLLIN:
21         client_socket = connections[fileno]
22         data = client_socket.recv(4096)
23         if data:
24             handle_input(client_socket, data)
25         else:
26             poll.unregister(fileno)
27             client_socket.close()
28             del connections[fileno]
29             del handlers[fileno]
30
31 def handle_accept(fileno, event):
32     (client_socket, client_address) = server_socket.accept()
33     print "got connection from", client_address
34     # client_socket.setblocking(0)
35     poll.register(client_socket.fileno(), select.POLLIN)
36     connections[client_socket.fileno()] = client_socket
37     handlers[client_socket.fileno()] = handle_request
38
39 poll.register(server_socket.fileno(), select.POLLIN)
40 handlers[server_socket.fileno()] = handle_accept
41
42 while True:
43     events = poll.poll(10000)  # 10 seconds
44     for fileno, event in events:
45         handler = handlers[fileno]
46         handler(fileno, event)
```

recipes/python/echo-reactor.py

如果要改成聊天服务，重新定义 handle_input 函数即可，程序的其余部分保持不变。

```
$ diff echo-reactor.py chat-reactor.py  -U1
 def handle_input(socket, data):
-    socket.send(data) # sendall() partial?
+    for (fd, other_socket) in connections.iteritems():
+        if other_socket != socket:
+            other_socket.send(data) # sendall() partial?
```

必须说明的是，完善的非阻塞 IO 网络库远比上面的玩具代码复杂，需要考虑各种错误场景。特别是要真正接管数据的收发，而不是像上面的示例那样直接在事件处理回调函数中发送网络数据。

注意在使用非阻塞 IO ＋事件驱动方式编程的时候，一定要注意避免在事件回调中执行耗时的操作，包括阻塞 IO 等，否则会影响程序的响应。这和 Windows GUI 消息循环非常类似。

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。这也是我们在一开始设计协议的时候使用了 id 的原因，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，程序结构如图 6-12 所示。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。代码参见：examples/sudoku/server_threadpool.cc。

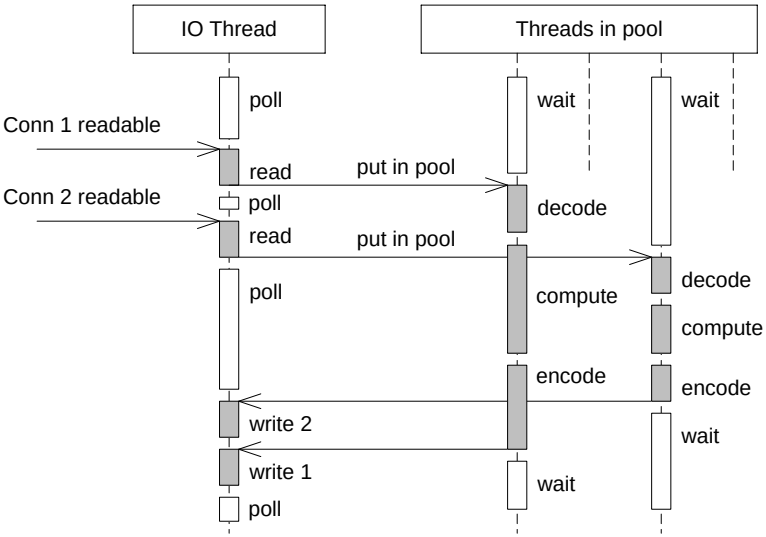


图 6-12

方案 8 使用线程池的代码与单线程 Reactor 的方案 5 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 有乱序返回的可能，客户端要根据 id 来匹配响应。

```
$ diff server_basic.cc server_threadpool.cc -u
--- server_basic.cc             2012-04-20 20:19:56.000000000 +0800
+++ server_threadpool.cc        2012-06-10 22:15:02.000000000 +0800
@@ -96,16 +100,7 @@  void onMessage(const TcpConnectionPtr& conn, ...

     if (puzzle.size() == implicit_cast<size_t>(kCells))
     {
-      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));
     }

@@ -114,17 +109,40 @@

+  static void solve(const TcpConnectionPtr& conn,
+                    const string& puzzle,
+                    const string& id)
+  {
```

```
+    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_;
   Timestamp startTime_;
 };
```

线程池的另外一个作用是执行阻塞操作。比如有的数据库的客户端只提供同步访问，那么可以把数据库查询放到线程池中，可以避免阻塞 IO 线程，不会影响其他客户连接，就像 Java Servlet 2.x 的做法一样。另外也可以用线程池来调用一些阻塞的 IO 函数，例如 `fsync(2)/fdatasync(2)`，这两个函数没有非阻塞的版本 [30]。

如果 IO 的压力比较大，一个 Reactor 处理不过来，可以试试方案 9，它采用多个 Reactor 来分担负载。

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(2)` 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 数目确定，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（见图 6-13）。

[30] 不过目前 Linux 内核的实现仍然会阻塞其他线程的磁盘 IO，见 http://antirez.com/post/fsync-different-thread-useless.html。

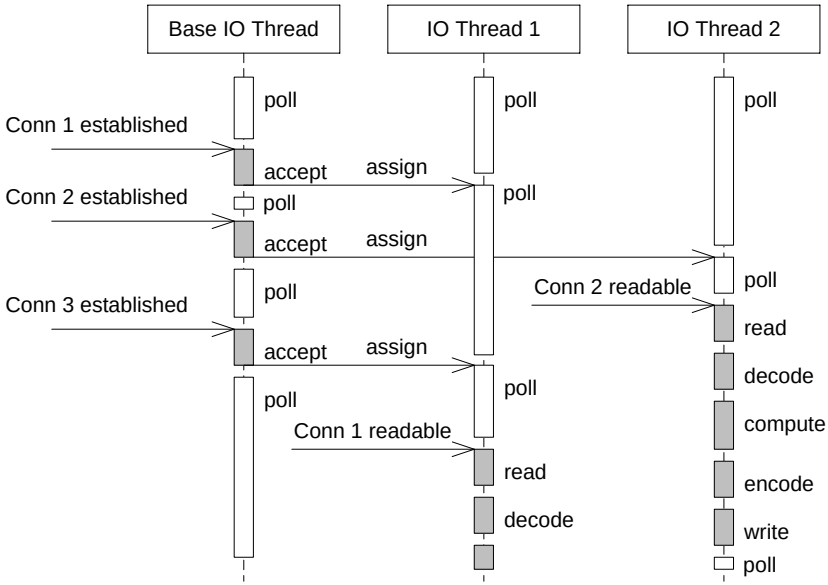


图 6-13

方案 9 代码见：examples/sudoku/server_multiloop.cc。它与 server_basic.cc 的区别很小，最关键的只有一行代码：`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 @@ class SudokuServer

-  SudokuServer(EventLoop* loop, const InetAddress& listenAddr)
+  SudokuServer(EventLoop* loop, const InetAddress& listenAddr, int numThreads)
     : loop_(loop),
       server_(loop, listenAddr, "SudokuServer"),
       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 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 Reactor 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），见图 6-14。

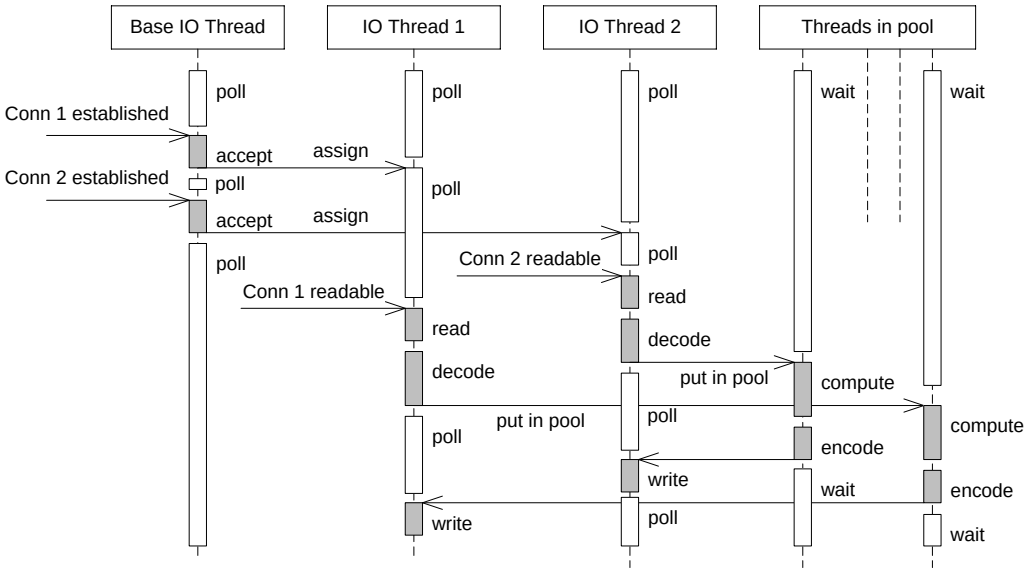


图 6-14

这种方案看起来复杂，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 [31]，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量，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 event loop 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

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

10 个数据请求连接，都归属同一个 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 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。

结语

我在 §3.3 曾写道：

总结起来，我推荐的 C++ 多线程服务端编程模式为：one loop per thread + thread pool。

- event loop 用作 non-blocking IO 和定时器。
- thread pool 用来做计算，具体可以是任务队列或生产者消费者队列。

当时（2010 年 2 月）写这篇博客时我还说：“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自己的思想完整地表达出来了。归纳一下 [32]，实用的方案有 5 种，muduo 直接支持后 4 种，见表 6-2。

表 6-2

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 6-2 中的 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。

[32] 此表参考了《Characteristics of multithreading models for high-performance IO driven network applications》一文（http://arxiv.org/ftp/arxiv/papers/0909/0909.4934.pdf）。

我再用银行柜台办理业务为比喻，简述各种模型的特点。银行有旋转门，办理业务的客户人员从旋转门进出（IO）；银行也有柜台，客户在柜台办理业务（计算）。要想办理业务，客户要先通过旋转门进入银行；办理完之后，客户要再次通过旋转门离开银行。一个客户可以办理多次业务，每次都必须从旋转门进出（TCP 长连接）。另外，旋转门一次只允许一个客户通过（无论进出），因为 `read()/write()` 只能同时调用其中一个。

方案 5：这间小银行有一个旋转门、一个柜台，每次只允许一名客户办理业务。而且当有人在办理业务时，旋转门是锁住的（计算和 IO 在同一线程）。为了维持工作效率，银行要求客户应该尽快办理业务，最好不要在取款的时候打电话去问家里人密码，也不要在通过旋转门的时候停下来系鞋带，这都会阻塞其他堵在门外的客户。如果客户很少，这是很经济且高效的方案；但是如果场地较大（多核），则这种布局就浪费了不少资源，只能并发（concurrent）不能并行（parallel）。如果确实一次办不完，应该离开柜台，到门外等着，等银行通知再来继续办理（分阶段回调）。

方案 8：这间银行有一个旋转门，一个或多个柜台。银行进门之后有一个队列，客户在这里排队到柜台（线程池）办理业务。即在单线程 Reactor 后面接了一个线程池用于计算，可以利用多核。旋转门基本是不锁的，随时都可以进出。但是排队会消耗一点时间，相比之下，方案 5 中客户一进门就能立刻办理业务。另外一种做法是线程池里的每个线程有自己的任务队列，而不是整个线程池共用一个任务队列。这样的好处是避免全局队列的锁争用，坏处是计算资源有可能分配不平均，降低并行度。

方案 9：这间大银行相当于包含方案 5 中的多家小银行，每个客户进大门的时候就被固定分配到某一间小银行中，他的业务只能由这间小银行办理，他每次都要进出小银行的旋转门。但总体来看，大银行可以同时服务多个客户。这时同样要求办理业务时不能空等（阻塞），否则会影响分到同一间小银行的其他客户。而且必要的时候可以为 VIP 客户单独开一间或几间小银行，优先办理 VIP 业务。这跟方案 5 不同，当普通客户在办理业务的时候，VIP 客户也只能在门外等着（见图 6-11 的右图）。这是一种适应性很强的方案，也是 muduo 原生的多线程 IO 模型。

方案 11：这间大银行有多个旋转门，多个柜台。旋转门和柜台之间没有一一对应关系，客户进大门的时候就被固定分配到某一旋转门中（奇怪的安排，易于实现线程安全的 IO，见 §4.6），进入旋转门之后，有一个队列，客户在此排队到柜台办理业务。这种方案的资源利用率可能比方案 9 更高，一个客户不会被同一小银行的其他客户阻塞，但延迟也比方案 9 略大。