Linux 多线程服务端编程
使用 muduo C++ 网络库

陈硕 (giantchen@gmail.com)

最后更新 2012-09-30

封面文案

示范在多核时代采用现代 C++ 编写

多线程 TCP 网络服务器的正规做法

内容简介

本书主要讲述采用现代 C++ 在 x86-64 Linux 上编写多线程 TCP 网络服务程序的主流常规技术，重点讲解一种适应性较强的多线程服务器的编程模型，即 one loop per thread。这是在 Linux 下以 native 语言编写用户态高性能网络程序最成熟的模式，掌握之后可顺利地开发各类常见的服务端网络应用程序。本书以 muduo 网络库为例，讲解这种编程模型的使用方法及注意事项。

本书的宗旨是贵精不贵多。掌握两种基本的同步原语就可以满足各种多线程同步的功能需求，还能写出更易用的同步设施。掌握一种进程间通信方式和一种多线程网络编程模型就足以应对日常开发任务，编写运行于公司内网环境的分布式服务系统。

作者简介

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

电子工业出版社

封底文案

看完了 W. Richard Stevens 的传世经典《UNIX 网络编程》，能照着例子用 Sockets API 编写 echo 服务，却仍然对稍微复杂一点的网络编程任务感到无从下手？学习网络编程有哪些好的练手项目？书中示例代码把业务逻辑和 Sockets 调用混在一起，似乎不利于将来扩展？网络编程中遇到一些具体问题该怎么办？例如：

- 程序在本机测试正常，放到网络上运行就经常出现数据收不全的情况？
- TCP 协议真的有所谓的“粘包问题”吗？该如何设计消息帧的协议？又该如何编码实现分包才不会掉到陷阱里？
- 带外数据（OOB）、信号驱动 IO 这些高级特性到底有没有用？
- 网络消息格式该怎么设计？发送 C struct 会有对齐方面的问题吗？对方不用 C/C++ 怎么通信？将来服务端软件升级，需要在消息中增加一个字段，现有的客户端就必须强制升级？
- 要处理成千上万的并发连接，似乎《UNIX 网络编程》介绍的传统 `fork()` 模型应付不过来，该用哪种并发模型呢？试试 `select(2)/poll(2)/epoll(4)` 这种 IO 复用模型吧，又感觉非阻塞 IO 陷阱重重，怎么程序的 CPU 使用率一直是 100%？
- 要不改用现成的 libevent 网络库吧，怎么查询一下数据库就把其他连接上的请求给耽误了？
- 再用个线程池吧。万一发回响应的时候对方已经断开连接了怎么办？会不会串话？

读过《UNIX 环境高级编程》，想用多线程来发挥多核 CPU 的性能潜力，但对程序该用哪种多线程模型感到一头雾水？有没有值得推荐的适用面广的多线程 IO 模型？互斥器、条件变量、读写锁、信号量这些底层同步原语哪些该用哪些不该用？有没有更高级的同步设施能简化开发？《UNIX 网络编程（第 2 卷）》介绍的那些琳琅满目的进程间通信（IPC）机制到底用哪个才能兼顾开发效率与可伸缩性？

网络编程和多线程编程的基础打得差不多，开始实际做项目了，更多问题扑面而来：

- 网上听人说服务端开发要做到 7×24 运行，为了防止内存碎片连动态内存分配都不能用，那岂不是连 C++ STL 也一并禁用了？硬件的可靠性高到值得去这么做吗？
- 传闻服务端开发主要通过日志来查错，那么日志里该写些什么？日志是写给谁看的？怎样写日志才不会影响性能？
- 分布式系统跟单机多进程到底有什么本质区别？心跳协议为什么是必需的，该如何实现？
- C++ 的大型工程该如何管理？库的接口如何设计才能保证升级的时候不破坏二进制兼容性？有没有更适合大规模分布式系统的部署方案？

这本《Linux 多线程服务端编程：使用 muduo C++ 网络库》中，作者凭借多年的工程实践经验试图解答以上疑问。当然，内容还远不止这些……

前言

本书主要讲述采用现代 C++ 在 x86-64 Linux 上编写多线程 TCP 网络服务程序的主流常规技术，这也是我对过去 5 年编写生产环境下的多线程服务端程序的经验总结。本书重点讲解多线程网络服务器的一种 IO 模型，即 one loop per thread。这是一种适应性较强的模型，也是 Linux 下以 native 语言编写用户态高性能网络程序最成熟的模式，掌握之后可顺利地开发各类常见的服务端网络应用程序。本书以 muduo 网络库为例，讲解这种编程模型的使用方法及注意事项。

muduo 是一个基于非阻塞 IO 和事件驱动的现代 C++ 网络库，原生支持 one loop per thread 这种 IO 模型。muduo 适合开发 Linux 下的面向业务的多线程服务端网络应用程序，其中“面向业务的网络编程”的定义见附录 A。“现代 C++”指的不是 C++11 新标准，而是 2005 年 TR1 发布之后的 C++ 语言和库。与传统 C++ 相比，现代 C++ 的变化主要有两方面：资源管理（见第 1 章）与事件回调（见第 449 页）。

本书不是多线程编程教程，也不是网络编程教程，更不是 C++ 教程。读者应该已经大致读过《UNIX 环境高级编程》、《UNIX 网络编程》、《C++ Primer》或与之内容相近的书籍。本书不谈 C++11，因为目前（2012 年）主流的 Linux 服务端发行版的 g++ 版本都还停留在 4.4，C++11 进入实用尚需一段时日。

本书适用的硬件环境是主流 x86-64 服务器，多路多核 CPU、几十 GB 内存、千兆以太网互联。除了第 5 章讲诊断日志之外，本书不涉及文件 IO。

本书分为四大部分，第 1 部分“C++ 多线程系统编程”考察多线程下的对象生命期管理、线程同步方法、多线程与 C++ 的结合、高效的多线程日志等。第 2 部分“muduo 网络库”介绍使用现成的非阻塞网络库编写网络应用程序的方法，以及 muduo 的设计与实现。第 3 部分“工程实践经验谈”介绍分布式系统的工程化开发方法和 C++ 在工程实践中的功能特性取舍。第 4 部分“附录”分享网络编程和 C++ 语言的学习经验。

本书的宗旨是贵精不贵多。掌握两种基本的同步原语就可以满足各种多线程同步的功能需求，还能写出更易用的同步设施。掌握一种进程间通信方式和一种多线程网络编程模型就足以应对日常开发任务，编写运行于公司内网环境的分布式服务系统。(本书不涉及分布式存储系统，也不涉及 UDP。)

术语与排版范例

本书大量使用英文术语，甚至有少量英文引文。设计模式的名字一律用英文，例如 Observer、Reactor、Singleton。在中文术语不够突出时，也会使用英文，例如 class、heap、event loop、STL algorithm 等。注意几个中文 C++ 术语：对象**实体**（instance）、函数重载**决议**（resolution）、模板**具现化**（instantiation）、**覆写**（override）虚函数、**提领**（dereference）指针。本书中的英语可数名词一般不用复数形式，例如两个 class，6 个 syscall；但有时会用 (s) 强调中文名词是复数。fd 是文件描述符（file descriptor）的缩写。“CPU 数目”一般指的是核（core）的数目。容量单位 kB、MB、GB 表示的字节数分别为 10^3、10^6、10^9，在特别强调准确数值时，会分别用 KiB、MiB、GiB 表示 2^{10}、2^{20}、2^{30} 字节。用诸如 §11.5 表示本书第 11.5 节，ℒ42 表示上下文中出现的第 42 行代码。[JCP]、[CC2e] 等是参考文献，见书末清单。

一般术语用普通罗马字体，如 mutex、socket；C++ 关键字用无衬线字体，如 class、this、mutable；函数名和 class 名用等宽字体，如 `fork(2)`、`muduo::EventLoop`，其中 `fork(2)` 表示系统函数 `fork()` 的文档位于 manpage 第 2 节，可以通过 `man 2 fork` 命令查看。如果函数名或类名过长，可能会折行，行末有连字号“-”，如 `EventLoop-ThreadPool`。文件路径和 URL 采用窄字体，例如 muduo/base/Date.h、http://chenshuo.com。用中文楷体表示引述别人的话。

代码

本书的示例代码以开源项目的形式发布在 GitHub 上，地址是 http://github.com/chenshuo/recipes/ 和 http://github.com/chenshuo/muduo/。本书配套页面提供全部源代码打包下载，正文中出现的类似 recipes/thread 的路径是压缩包内的相对路径，读者不难找到其对应的 GitHub URL。本书引用代码的形式如下，左侧数字是文件的行号，右侧的“muduo/base/Types.h”是文件路径[1]。例如下面这几行代码是 `muduo::string` 的 typedef。

muduo/base/Types.h

```
15 namespace muduo
16 {
17
18 #ifdef MUDUO_STD_STRING
19 using std::string;
20 #else  // !MUDUO_STD_STRING
21 typedef __gnu_cxx::__sso_string string;
22 #endif
```

muduo/base/Types.h

[1] 在第 6、7 两章的 muduo 示例代码中，路径 muduo/examples/XXX 会简写为 examples/XXX。此外，第 8 章会把 recipes/reactor/XXX 简写为 reactor/XXX。

本书假定读者熟悉 `diff -u` 命令的输出格式，用于表示代码的改动。

本书正文中出现的代码有时为了照顾排版而略有改写，例如改变缩进规则，去掉单行条件语句前后的花括号等。就编程风格而论，应以电子版代码为准。

联系方式

邮箱：giantchen@gmail.com
主页：http://chenshuo.com/book （正文和脚注中出现的 URL 可从这里找到。）
微博：http://weibo.com/giantchen
博客：http://blog.csdn.net/Solstice
代码：http://github.com/chenshuo

陈硕
中国 · 香港

内容一览

目录

第 1 部分

C++ 多线程系统编程

第 1 章

线程安全的对象生命期管理

编写线程安全的类不是难事，用同步原语（synchronization primitives）保护内部状态即可。但是对象的生与死不能由对象自身拥有的 mutex（互斥器）来保护。如何避免对象析构时可能存在的 race condition（竞态条件）是 C++ 多线程编程面临的基本问题，可以借助 Boost 库中的 `shared_ptr` 和 `weak_ptr` [1] 完美解决。这也是实现线程安全的 Observer 模式的必备技术。

本章源自 2009 年 12 月我在上海祝成科技举办的 C++ 技术大会的一场演讲《当析构函数遇到多线程》，读者应具有 C++ 多线程编程经验，熟悉互斥器、竞态条件等概念，了解智能指针，知道 Observer 设计模式。

1.1 当析构函数遇到多线程

与其他面向对象语言不同，C++ 要求程序员自己管理对象的生命期，这在多线程环境下显得尤为困难。当一个对象能被多个线程同时看到时，那么对象的销毁时机就会变得模糊不清，可能出现多种竞态条件（race condition）：

- 在即将析构一个对象时，从何而知此刻是否有别的线程正在执行该对象的成员函数？
- 如何保证在执行成员函数期间，对象不会在另一个线程被析构？
- 在调用某个对象的成员函数之前，如何得知这个对象还活着？它的析构函数会不会碰巧执行到一半？

解决这些 race condition 是 C++ 多线程编程面临的基本问题。本文试图以 `shared_ptr` 一劳永逸地解决这些问题，减轻 C++ 多线程编程的精神负担。

[1] 这两个 class 也是 TR1 的一部分，位于 `std::tr1` 命名空间；在 C++11 中，它们是标准库的一部分。

1.1.1 线程安全的定义

依据 [JCP]，一个线程安全的 class 应当满足以下三个条件：

- 多个线程同时访问时，其表现出正确的行为。
- 无论操作系统如何调度这些线程，无论这些线程的执行顺序如何交织（interleaving）。
- 调用端代码无须额外的同步或其他协调动作。

依据这个定义，C++ 标准库里的大多数 class 都不是线程安全的，包括 `std::string`、`std::vector`、`std::map` 等，因为这些 class 通常需要在外部加锁才能供多个线程同时访问。

1.1.2 MutexLock 与 MutexLockGuard

为了便于后文讨论，先约定两个工具类。我相信每个写 C++ 多线程程序的人都实现过或使用过类似功能的类，代码见 §2.4。

`MutexLock` 封装临界区（critical section），这是一个简单的资源类，用 RAII 手法 [CCS，条款 13] 封装互斥器的创建与销毁。临界区在 Windows 上是 struct `CRITICAL_SECTION`，是可重入的；在 Linux 下是 `pthread_mutex_t`，默认是不可重入的 [2]。`MutexLock` 一般是别的 class 的数据成员。

`MutexLockGuard` 封装临界区的进入和退出，即加锁和解锁。`MutexLockGuard` 一般是个栈上对象，它的作用域刚好等于临界区域。

这两个 class 都不允许拷贝构造和赋值，它们的使用原则见 §2.1。

1.1.3 一个线程安全的 Counter 示例

编写单个的线程安全的 class 不算太难，只需用同步原语保护其内部状态。例如下面这个简单的计数器类 `Counter`：

```
1  // A thread-safe counter
2  class Counter : boost::noncopyable
3  {
4    // copy-ctor and assignment should be private by default for a class.
5   public:
6    Counter() : value_(0) {}
```

[2] 可重入与不可重入的讨论见 §2.1.1。

```
7     int64_t value() const;
8     int64_t getAndIncrease();
9
10   private:
11    int64_t value_;
12    mutable MutexLock mutex_;
13  };
14
15  int64_t Counter::value() const
16  {
17    MutexLockGuard lock(mutex_);  // lock 的析构会晚于返回对象的构造，
18    return value_;                // 因此有效地保护了这个共享数据。
19  }
20
21  int64_t Counter::getAndIncrease()
22  {
23    MutexLockGuard lock(mutex_);
24    int64_t ret = value_++;
25    return ret;
26  }
27  // In a real world, atomic operations are preferred.
28  // 当然在实际项目中，这个 class 用原子操作更合理，这里用锁仅仅为了举例。
```

这个 class 很直白，一看就明白，也容易验证它是线程安全的。每个 Counter 对象有自己的 mutex_，因此不同对象之间不构成锁争用（lock contention）。即两个线程有可能同时执行 $\mathcal{L}$24，前提是它们访问的不是同一个 Counter 对象。注意到其 mutex_ 成员是 mutable 的，意味着 const 成员函数如 Counter::value() 也能直接使用 non-const 的 mutex_。思考：如果 mutex_ 是 static，是否影响正确性和/或性能？

尽管这个 Counter 本身毫无疑问是线程安全的，但如果 Counter 是动态创建的并通过指针来访问，前面提到的对象销毁的 race condition 仍然存在。

1.2 对象的创建很简单

对象构造要做到线程安全，唯一的要求是在构造期间不要泄露 this 指针，即

- 不要在构造函数中注册任何回调；
- 也不要在构造函数中把 this 传给跨线程的对象；
- 即便在构造函数的最后一行也不行。

之所以这样规定，是因为在构造函数执行期间对象还没有完成初始化，如果 this 被泄露（escape）给了其他对象（其自身创建的子对象除外），那么别的线程有可能访问这个半成品对象，这会造成难以预料的后果。

```
// 不要这么做（Don't do this.）
class Foo : public Observer  // Observer 的定义见第 10 页
{
 public:
  Foo(Observable* s)
  {
    s->register_(this);  // 错误，非线程安全
  }

  virtual void update();
};
```

对象构造的正确方法：

```
// 要这么做（Do this.）
class Foo : public Observer
{
 public:
  Foo();
  virtual void update();

  // 另外定义一个函数，在构造之后执行回调函数的注册工作
  void observe(Observable* s)
  {
    s->register_(this);
  }
};

Foo* pFoo = new Foo;
Observable* s = getSubject();
pFoo->observe(s);  // 二段式构造，或者直接写 s->register_(pFoo);
```

这也说明，二段式构造——即构造函数 +initialize()——有时会是好办法，这虽然不符合 C++ 教条，但是多线程下别无选择。另外，既然允许二段式构造，那么构造函数不必主动抛异常，调用方靠 initialize() 的返回值来判断对象是否构造成功，这能简化错误处理。

即使构造函数的最后一行也不要泄露 this，因为 Foo 有可能是个基类，基类先于派生类构造，执行完 Foo::Foo() 的最后一行代码还会继续执行派生类的构造函数，这时 most-derived class 的对象还处于构造中，仍然不安全。

相对来说，对象的构造做到线程安全还是比较容易的，毕竟曝光少，回头率为零。而析构的线程安全就不那么简单，这也是本章关注的焦点。

1.3 销毁太难

对象析构，这在单线程里不构成问题，最多需要注意避免空悬指针和野指针[3]。而在多线程程序中，存在了太多的竞态条件。对一般成员函数而言，做到线程安全的办法是让它们顺次执行，而不要并发执行（关键是不要同时读写共享状态），也就是让每个成员函数的临界区不重叠。这是显而易见的，不过有一个隐含条件或许不是每个人都能立刻想到：成员函数用来保护临界区的互斥器本身必须是有效的。而析构函数破坏了这一假设，它会把 mutex 成员变量销毁掉。悲剧啊！

1.3.1 mutex 不是办法

mutex 只能保证函数一个接一个地执行，考虑下面的代码，它试图用互斥锁来保护析构函数：（注意代码中的 (1) 和 (2) 两处标记。）

```
Foo::~Foo()
{
  MutexLockGuard lock(mutex_);
  // free internal state  (1)
}
```

```
void Foo::update()
{
  MutexLockGuard lock(mutex_);  // (2)
  // make use of internal state
}
```

此时，有 A、B 两个线程都能看到 Foo 对象 x，线程 A 即将销毁 x，而线程 B 正准备调用 x->update()。

```
extern Foo* x; // visible by all threads
```

```
// thread A
delete x;
x = NULL;  // helpless
```

```
// thread B
if (x) {
  x->update();
}
```

尽管线程 A 在销毁对象之后把指针置为了 NULL，尽管线程 B 在调用 x 的成员函数之前检查了指针 x 的值，但还是无法避免一种 race condition：

1. 线程 A 执行到了析构函数的 (1) 处，已经持有了互斥锁，即将继续往下执行。
2. 线程 B 通过了 if (x) 检测，阻塞在 (2) 处。

接下来会发生什么，只有天晓得。因为析构函数会把 mutex_ 销毁，那么 (2) 处有可能永远阻塞下去，有可能进入“临界区”，然后 core dump，或者发生其他更糟糕的情况。

这个例子至少说明 delete 对象之后把指针置为 NULL 根本没用，如果一个程序要靠这个来防止二次释放，说明代码逻辑出了问题。

[3] 空悬指针（dangling pointer）指向已经销毁的对象或已经回收的地址，野指针（wild pointer）指的是未经初始化的指针（http://en.wikipedia.org/wiki/Dangling_pointer）。

1.3.2 作为数据成员的 mutex 不能保护析构

前面的例子说明，作为 class 数据成员的 MutexLock 只能用于同步本 class 的其他数据成员的读和写，它不能保护安全地析构。因为 MutexLock 成员的生命期最多与对象一样长，而析构动作可说是发生在对象身故之后（或者身亡之时）。另外，对于基类对象，那么调用到基类析构函数的时候，派生类对象的那部分已经析构了，那么基类对象拥有的 MutexLock 不能保护整个析构过程。再说，析构过程本来也不需要保护，因为只有别的线程都访问不到这个对象时，析构才是安全的，否则会有 §1.1 谈到的竞态条件发生。

另外如果要同时读写一个 class 的两个对象，有潜在的死锁可能。比方说有 swap() 这个函数：

```
void swap(Counter& a, Counter& b)
{
  MutexLockGuard aLock(a.mutex_);  // potential dead lock
  MutexLockGuard bLock(b.mutex_);
  int64_t value = a.value_;
  a.value_ = b.value_;
  b.value_ = value;
}
```

如果线程 A 执行 swap(a, b); 而同时线程 B 执行 swap(b, a);，就有可能死锁。operator=() 也是类似的道理。

```
Counter& Counter::operator=(const Counter& rhs)
{
  if (this == &rhs)
    return *this;

  MutexLockGuard myLock(mutex_);   // potential dead lock
  MutexLockGuard itsLock(rhs.mutex_);
  value_ = rhs.value_;  // 改成 value_ = rhs.value() 会死锁
  return *this;
}
```

一个函数如果要锁住相同类型的多个对象，为了保证始终按相同的顺序加锁，我们可以比较 mutex 对象的地址，始终先加锁地址较小的 mutex。

1.4 线程安全的 Observer 有多难

一个动态创建的对象是否还活着，光看指针是看不出来的（引用也一样看不出来）。指针就是指向了一块内存，这块内存上的对象如果已经销毁，那么就根本不能

访问 [CCS，条款 99]（就像 `free(3)` 之后的地址不能访问一样），既然不能访问又如何知道对象的状态呢？换句话说，判断一个指针是不是合法指针没有高效的办法，这是 C/C++ 指针问题的根源[4]。（万一原址又创建了一个新的对象呢？再万一这个新的对象的类型异于老的对象呢？）

在面向对象程序设计中，对象的关系主要有三种：composition、aggregation、association。composition（组合/复合）关系在多线程里不会遇到什么麻烦，因为对象 x 的生命期由其唯一的拥有者 owner 控制，owner 析构的时候会把 x 也析构掉。从形式上看，x 是 owner 的直接数据成员，或者 `scoped_ptr` 成员，抑或 owner 持有的容器的元素。

后两种关系在 C++ 里比较难办，处理不好就会造成内存泄漏或重复释放。association（关联/联系）是一种很宽泛的关系，它表示一个对象 a 用到了另一个对象 b，调用了后者的成员函数。从代码形式上看，a 持有 b 的指针（或引用），但是 b 的生命期不由 a 单独控制。aggregation（聚合）关系从形式上看与 association 相同，除了 a 和 b 有逻辑上的整体与部分关系。如果 b 是动态创建的并在整个程序结束前有可能被释放，那么就会出现 §1.1 谈到的竞态条件。

那么似乎一个简单的解决办法是：只创建不销毁。程序使用一个对象池来暂存用过的对象，下次申请新对象时，如果对象池里有存货，就重复利用现有的对象，否则就新建一个。对象用完了，不是直接释放掉，而是放回池子里。这个办法当然有其自身的很多缺点，但至少能避免访问失效对象的情况发生。

这种山寨办法的问题有：

- 对象池的线程安全，如何安全地、完整地把对象放回池子里，防止出现“部分放回”的竞态？（线程 A 认为对象 x 已经放回了，线程 B 认为对象 x 还活着。）
- 全局共享数据引发的 lock contention，这个集中化的对象池会不会把多线程并发的操作串行化？
- 如果共享对象的类型不止一种，那么是重复实现对象池还是使用类模板？
- 会不会造成内存泄漏与分片？因为对象池占用的内存只增不减，而且多个对象池不能共享内存（想想为何）。

回到正题上来，如果对象 x 注册了任何非静态成员函数回调，那么必然在某处持有了指向 x 的指针，这就暴露在了 race condition 之下。

[4] 在 Java 中，一个 reference 只要不为 null，它一定指向有效的对象。

一个典型的场景是 Observer 模式（代码见 recipes/thread/test/Observer.cc）。

```
class Observer // : boost::noncopyable
{
 public:
  virtual ~Observer();
  virtual void update() = 0;
  // ...
};

class Observable // : boost::noncopyable
{
 public:
  void register_(Observer* x);
  void unregister(Observer* x);

  void notifyObservers() {
    for (Observer* x : observers_) {  // 这行是 C++11
      x->update(); // (3)
    }
  }
 private:
  std::vector<Observer*> observers_;
};
```

当 Observable 通知每一个 Observer 时 (*L*17)，它从何得知 Observer 对象 x 还活着？要不试试在 Observer 的析构函数里调用 unregister() 来解注册？恐难奏效。

```
class Observer
{
  // 同前
  void observe(Observable* s) {
    s->register_(this);
    subject_ = s;
  }

  virtual ~Observer() {
    subject_->unregister(this);
  }

  Observable* subject_;
};
```

我们试着让 Observer 的析构函数去调用 unregister(this)，这里有两个 race conditions。其一：*L*32 如何得知 subject_ 还活着？其二：就算 subject_ 指向某个永久存在的对象，那么还是险象环生：

1. 线程 A 执行到 *L*32 之前，还没有来得及 unregister 本对象。
2. 线程 B 执行到 *L*17，x 正好指向是 *L*32 正在析构的对象。

这时悲剧又发生了，既然 x 所指的 Observer 对象正在析构，调用它的任何非静态成员函数都是不安全的，何况是虚函数[5]。更糟糕的是，Observer 是个基类，执行到 $\mathcal{L}32$ 时，派生类对象已经析构掉了，这时候整个对象处于将死未死的状态，core dump 恐怕是最幸运的结果。

这些 race condition 似乎可以通过加锁来解决，但在哪儿加锁，谁持有这些互斥锁，又似乎不是那么显而易见的。要是有什么活着的对象能帮帮我们就好了，它提供一个 isAlive() 之类的程序函数，告诉我们那个对象还在不在。可惜指针和引用都不是对象，它们是内建类型。

1.5 原始指针有何不妥

指向对象的原始指针（raw pointer）是坏的，尤其当暴露给别的线程时。Observable 应当保存的不是原始的 Observer*，而是别的什么东西，能分辨 Observer 对象是否存活。类似地，如果 Observer 要在析构函数里解注册（这虽然不能解决前面提到的 race condition，但是在析构函数里打扫战场还是应该的），那么 subject_ 的类型也不能是原始的 Observable*。

有经验的 C++ 程序员或许会想到用智能指针。没错，这是正道，但也没那么简单，有些关窍需要注意。这两处直接使用 shared_ptr 是不行的，会形成循环引用，直接造成资源泄漏。别着急，后文会一一讲到。

空悬指针

有两个指针 p1 和 p2，指向堆上的同一个对象 Object，p1 和 p2 位于不同的线程中（图 1-1 的左图）。假设线程 A 通过 p1 指针将对象销毁了（尽管把 p1 置为了 NULL），那 p2 就成了空悬指针（图 1-1 的右图）。这是一种典型的 C/C++ 内存错误。

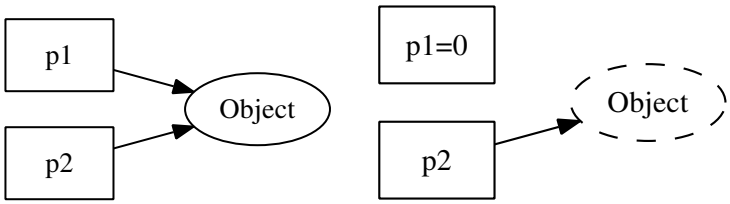


图 1-1

要想安全地销毁对象，最好在别人（线程）都看不到的情况下，偷偷地做。（这正是垃圾回收的原理，所有人都用不到的东西一定是垃圾。）

[5] C++ 标准对在构造函数和析构函数中调用虚函数的行为有明确规定，但是没有考虑并发调用的情况。

一个“解决办法”

一个解决空悬指针的办法是，引入一层间接性，让 p1 和 p2 所指的对象永久有效。比如图 1-2 中的 proxy 对象，这个对象，持有一个指向 `Object` 的指针。（从 C 语言的角度，p1 和 p2 都是二级指针。）

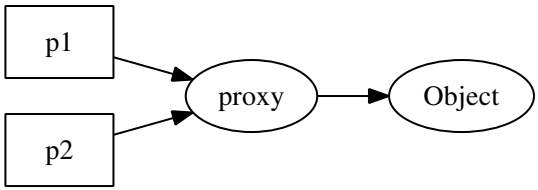


图 1-2

当销毁 `Object` 之后，proxy 对象继续存在，其值变为 0（见图 1-3）。而 p2 也没有变成空悬指针，它可以通过查看 proxy 的内容来判断 Object 是否还活着。

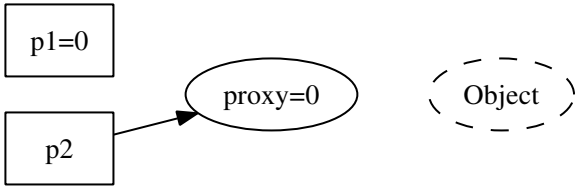


图 1-3

要线程安全地释放 `Object` 也不是那么容易，race condition 依旧存在。比如 p2 看第一眼的时候 proxy 不是零，正准备去调用 `Object` 的成员函数，期间对象已经被 p1 给销毁了。

问题在于，何时释放 proxy 指针呢？

一个更好的解决办法

为了安全地释放 proxy，我们可以引入引用计数（reference counting），再把 p1 和 p2 都从指针变成对象 sp1 和 sp2。proxy 现在有两个成员，指针和计数器。

1. 一开始，有两个引用，计数值为 2（见图 1-4）。

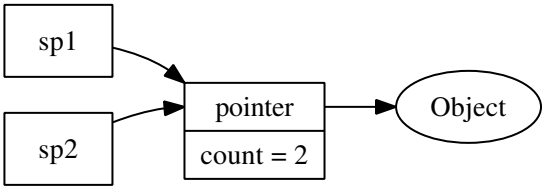


图 1-4

2. sp1 析构了，引用计数的值减为 1（见图 1-5）。

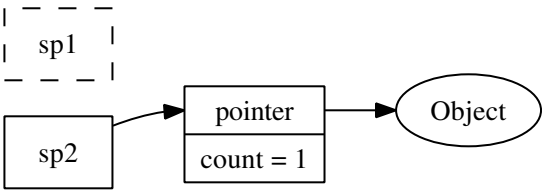


图 1-5

3. sp2 也析构了，引用计数降为 0，可以安全地销毁 proxy 和 `Object` 了（见图 1-6）。

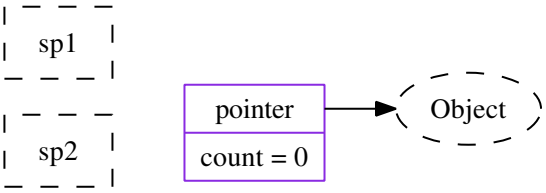


图 1-6

慢着！这不正是引用计数型智能指针吗？

一个万能的解决方案

引入另外一层间接性（another layer of indirection）[6]，用对象来管理共享资源（如果把 `Object` 看作资源的话），亦即 handle/body 惯用技法（idiom）。当然，编写线程安全、高效的引用计数 handle 的难度非凡，作为一名谦卑的程序员[7]，用现成的库就行。万幸，C++ 的 TR1 标准库里提供了一对“神兵利器”，可助我们完美解决这个头疼的问题。

1.6 神器 shared_ptr/weak_ptr

`shared_ptr` 是引用计数型智能指针，在 Boost 和 `std::tr1` 里均提供，也被纳入 C++11 标准库，现代主流的 C++ 编译器都能很好地支持。`shared_ptr<T>` 是一个类模板（class template），它只有一个类型参数，使用起来很方便。引用计数是自动化

[6] http://en.wikipedia.org/wiki/Abstraction_layer

[7] 参见 Edsger W. Dijkstra 的著名演讲《The Humble Programmer》（http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html）。

资源管理的常用手法，当引用计数降为 0 时，对象（资源）即被销毁。weak_ptr 也是一个引用计数型智能指针，但是它不增加对象的引用次数，即弱（weak）引用。

shared_ptr 的基本用法和语意请参考手册或教程，本书从略。谈几个关键点。

- shared_ptr 控制对象的生命期。shared_ptr 是强引用（想象成用铁丝绑住堆上的对象），只要有一个指向 x 对象的 shared_ptr 存在，该 x 对象就不会析构。当指向对象 x 的最后一个 shared_ptr 析构或 reset() 的时候，x 保证会被销毁。
- weak_ptr 不控制对象的生命期，但是它知道对象是否还活着（想象成用棉线轻轻拴住堆上的对象）。如果对象还活着，那么它可以提升（promote）为有效的 shared_ptr；如果对象已经死了，提升会失败，返回一个空的 shared_ptr。“提升/lock()”行为是线程安全的。
- shared_ptr/weak_ptr 的“计数”在主流平台上是原子操作，没有用锁，性能不俗。
- shared_ptr/weak_ptr 的线程安全级别与 std::string 和 STL 容器一样，后面还会讲。

孟岩在《垃圾收集机制批判》[8] 中一针见血地点出智能指针的优势：“C++ 利用智能指针达成的效果是：一旦某对象不再被引用，系统刻不容缓，立刻回收内存。这通常发生在关键任务完成后的清理（clean up）时期，不会影响关键任务的实时性，同时，内存里所有的对象都是有用的，绝对没有垃圾空占内存。”

1.7 插曲：系统地避免各种指针错误

我同意孟岩说的[9] “大部分用 C 写的上规模的软件都存在一些内存方面的错误，需要花费大量的精力和时间把产品稳定下来。”举例来说，就像 Nginx 这样成熟且广泛使用的 C 语言产品都会不时暴露出低级的内存错误[10]。

内存方面的问题在 C++ 里很容易解决，我第一次也是最后一次见到别人的代码里有内存泄漏是在 2004 年实习那会儿，我自己写的 C++ 程序从来没有出现过内存方面的问题。

[8] http://blog.csdn.net/myan/article/details/1906

[9]《Java 替代 C 语言的可能性》（http://blog.csdn.net/myan/article/details/1482614）。

[10] http://trac.nginx.org/nginx/ticket/{134,135,162}

C++ 里可能出现的内存问题大致有这么几个方面：

1. 缓冲区溢出（buffer overrun）。
2. 空悬指针/野指针。
3. 重复释放（double delete）。
4. 内存泄漏（memory leak）。
5. 不配对的 new[]/delete。
6. 内存碎片（memory fragmentation）。

正确使用智能指针能很轻易地解决前面 5 个问题，解决第 6 个问题需要别的思路，我会在 §9.2.1 和 §A.1.8 探讨。

1. 缓冲区溢出：用 `std::vector<char>/std::string` 或自己编写 `Buffer` class 来管理缓冲区，自动记住用缓冲区的长度，并通过成员函数而不是裸指针来修改缓冲区。
2. 空悬指针/野指针：用 `shared_ptr/weak_ptr`，这正是本章的主题。
3. 重复释放：用 `scoped_ptr`，只在对象析构的时候释放一次。
4. 内存泄漏：用 `scoped_ptr`，对象析构的时候自动释放内存。
5. 不配对的 new[]/delete：把 new[] 统统替换为 `std::vector/scoped_array`。

正确使用上面提到的这几种智能指针并不难，其难度大概比学习使用 `std::vector/std::list` 这些标准库组件还要小，与 `std::string` 差不多，只要花一周的时间去适应它，就能信手拈来。我认为，在现代的 C++ 程序中一般不会出现 delete 语句，资源（包括复杂对象本身）都是通过对象（智能指针或容器）来管理的，不需要程序员还为此操心。

在这几种错误里边，内存泄漏相对危害性较小，因为它只是借了东西不归还，程序功能在一段时间内还算正常。其他如缓冲区溢出或重复释放等致命错误可能会造成安全性（security 和 data safety）方面的严重后果。

需要注意一点：`scoped_ptr/shared_ptr/weak_ptr` 都是值语意，要么是栈上对象，或是其他对象的直接数据成员，或是标准库容器里的元素。几乎不会有下面这种用法：

```
shared_ptr<Foo>* pFoo = new shared_ptr<Foo>(new Foo);  // WRONG semantic
```

还要注意，如果这几种智能指针是对象 x 的数据成员，而它的模板参数 T 是个 incomplete 类型，那么 x 的析构函数不能是默认的或内联的，必须在 .cpp 文件里边显式定义，否则会有编译错或运行错（原因见 §10.3.2）。

1.8 应用到 Observer 上

既然通过 weak_ptr 能探查对象的生死，那么 Observer 模式的竞态条件就很容易解决，只要让 Observable 保存 weak_ptr<Observer> 即可：

recipes/thread/test/Observer_safe.cc

```
39  class Observable  // not 100% thread safe!
40  {
41   public:
42    void register_(weak_ptr<Observer> x);  // 参数类型可用 const weak_ptr<Observer>&
43    // void unregister(weak_ptr<Observer> x);  // 不需要它
44    void notifyObservers();
45
46   private:
47    mutable MutexLock mutex_;
48    std::vector<weak_ptr<Observer> > observers_;
49    typedef std::vector<weak_ptr<Observer> >::iterator Iterator;
50  };
51
52  void Observable::notifyObservers()
53  {
54    MutexLockGuard lock(mutex_);
55    Iterator it = observers_.begin();  // Iterator 的定义见第 49 行
56    while (it != observers_.end())
57    {
58      shared_ptr<Observer> obj(it->lock());  // 尝试提升，这一步是线程安全的
59      if (obj)
60      {
61        // 提升成功，现在引用计数值至少为 2 （想想为什么？）
62        obj->update();  // 没有竞态条件，因为 obj 在栈上，对象不可能在本作用域内销毁
63        ++it;
64      }
65      else
66      {
67        // 对象已经销毁，从容器中拿掉 weak_ptr
68        it = observers_.erase(it);
69      }
70    }
71  }
```

recipes/thread/test/Observer_safe.cc

就这么简单。前文代码 (3) 处（p. 10 的 £17）的竞态条件已经弥补了。思考：如果把 £48 改为 vector<shared_ptr<Observer> > observers_;，会有什么后果？

解决了吗

把 Observer* 替换为 weak_ptr<Observer> 部分解决了 Observer 模式的线程安全，但还有以下几个疑点。这些问题留到本章 §1.14 中去探讨，每个都是能解决的。

侵入性 强制要求 Observer 必须以 shared_ptr 来管理。

不是完全线程安全 Observer 的析构函数会调用 subject_->unregister(this)，万一 subject_ 已经不复存在了呢？为了解决它，又要求 Observable 本身是用 shared_ptr 管理的，并且 subject_ 多半是个 weak_ptr<Observable>。

锁争用（lock contention） 即 Observable 的三个成员函数都用了互斥器来同步，这会造成 register_() 和 unregister() 等待 notifyObservers()，而后者的执行时间是无上限的，因为它同步回调了用户提供的 update() 函数。我们希望 register_() 和 unregister() 的执行时间不会超过某个固定的上限，以免殃及无辜群众。

死锁 万一 £62 的 update() 虚函数中调用了 (un)register 呢？如果 mutex_ 是不可重入的，那么会死锁；如果 mutex_ 是可重入的，程序会面临迭代器失效（core dump 是最好的结果），因为 vector observers_ 在遍历期间被意外地修改了。这个问题乍看起来似乎没有解决办法，除非在文档里做要求。（一种办法是：用可重入的 mutex_，把容器换为 std::list，并把 ++it 往前挪一行。）

我个人倾向于使用不可重入的 mutex，例如 Pthreads 默认提供的那个，因为“要求 mutex 可重入”本身往往意味着设计上出了问题（§2.1.1）。Java 的 intrinsic lock 是可重入的，因为要允许 synchronized 方法相互调用（派生类调用基类的同名 synchronized 方法），我觉得这也是无奈之举。

1.9 再论 shared_ptr 的线程安全

虽然我们借 shared_ptr 来实现线程安全的对象释放，但是 shared_ptr 本身不是 100% 线程安全的。它的引用计数本身是安全且无锁的，但对象的读写则不是，因为 shared_ptr 有两个数据成员，读写操作不能原子化。根据文档 [11]，shared_ptr 的线程安全级别和内建类型、标准库容器、std::string 一样，即：

- 一个 shared_ptr 对象实体可被多个线程同时读取；
- 两个 shared_ptr 对象实体可以被两个线程同时写入，“析构”算写操作；
- 如果要从多个线程读写同一个 shared_ptr 对象，那么需要加锁。

请注意，以上是 shared_ptr 对象本身的线程安全级别，不是它管理的对象的线程安全级别。

[11] http://www.boost.org/doc/libs/release/libs/smart_ptr/shared_ptr.htm#ThreadSafety

要在多个线程中同时访问同一个 shared_ptr，正确的做法是用 mutex 保护：

```
MutexLock mutex; // No need for ReaderWriterLock
shared_ptr<Foo> globalPtr;

// 我们的任务是把 globalPtr 安全地传给 doit()
void doit(const shared_ptr<Foo>& pFoo);
```

globalPtr 能被多个线程看到，那么它的读写需要加锁。注意我们不必用读写锁，而只用最简单的互斥锁，这是为了性能考虑。因为临界区非常小，用互斥锁也不会阻塞并发读。

为了拷贝 globalPtr，需要在读取它的时候加锁，即：

```
void read()
{
  shared_ptr<Foo> localPtr;
  {
    MutexLockGuard lock(mutex);
    localPtr = globalPtr;  // read globalPtr
  }
  // use localPtr since here, 读写 localPtr 也无须加锁
  doit(localPtr);
}
```

写入的时候也要加锁：

```
void write()
{
  shared_ptr<Foo> newPtr(new Foo);  // 注意，对象的创建在临界区之外
  {
    MutexLockGuard lock(mutex);
    globalPtr = newPtr;  // write to globalPtr
  }
  // use newPtr since here, 读写 newPtr 无须加锁
  doit(newPtr);
}
```

注意到上面的 read() 和 write() 在临界区之外都没有再访问 globalPtr，而是用了一个指向同一 Foo 对象的栈上 shared_ptr local copy。下面会谈到，只要有这样的 local copy 存在，shared_ptr 作为函数参数传递时不必复制，用 reference to const 作为参数类型即可。另外注意到上面的 new Foo 是在临界区之外执行的，这种写法通常比在临界区内写 globalPtr.reset(new Foo) 要好，因为缩短了临界区长度。如果要销毁对象，我们固然可以在临界区内执行 globalPtr.reset()，但是这样往往会让对象析构发生在临界区以内，增加了临界区的长度。一种改进办法是像上面一样定义一个 localPtr，用它在临界区内与 globalPtr 交换（swap()），这样能保证把对象的销毁推迟到临界区之外。练习：在 write() 函数中，globalPtr = newPtr; 这一句有可能会在临界区内销毁原来 globalPtr 指向的 Foo 对象，设法将销毁行为移出临界区。

1.10 shared_ptr 技术与陷阱

意外延长对象的生命期 shared_ptr 是强引用（“铁丝”绑的），只要有一个指向 x 对象的 shared_ptr 存在，该对象就不会析构。而 shared_ptr 又是允许拷贝构造和赋值的（否则引用计数就无意义了），如果不小心遗留了一个拷贝，那么对象就永世长存了。例如前面提到如果把 p. 16 中 £48 observers_ 的类型改为 vector<shared_ptr<Observer> >，那么除非手动调用 unregister()，否则 Observer 对象永远不会析构。即便它的析构函数会调用 unregister()，但是不去 unregister() 就不会调用 Observer 的析构函数，这变成了鸡与蛋的问题。这也是 Java 内存泄漏的常见原因。

另外一个出错的可能是 boost::bind，因为 boost::bind 会把实参拷贝一份，如果参数是个 shared_ptr，那么对象的生命期就不会短于 boost::function 对象：

```
class Foo
{
  void doit();
};

shared_ptr<Foo> pFoo(new Foo);
boost::function<void()> func = boost::bind(&Foo::doit, pFoo);  // long life foo
```

这里 func 对象持有了 shared_ptr<Foo> 的一份拷贝，有可能会在不经意间延长倒数第二行创建的 Foo 对象的生命期。

函数参数 因为要修改引用计数（而且拷贝的时候通常要加锁），shared_ptr 的拷贝开销比拷贝原始指针要高，但是需要拷贝的时候并不多。多数情况下它可以以 const reference 方式传递，一个线程只需要在最外层函数有一个实体对象，之后都可以用 const reference 来使用这个 shared_ptr。例如有几个函数都要用到 Foo 对象：

```
void save(const shared_ptr<Foo>& pFoo);     // pass by const reference
void validateAccount(const Foo& foo);

bool validate(const shared_ptr<Foo>& pFoo)  // pass by const reference
{
  validateAccount(*pFoo);
  // ...
}
```

那么在通常情况下，我们可以传常引用（pass by const reference）：

```
void onMessage(const string& msg)
{
  shared_ptr<Foo> pFoo(new Foo(msg));  // 只要在最外层持有一个实体，安全不成问题
  if (validate(pFoo)) {  // 没有拷贝 pFoo
    save(pFoo);          // 没有拷贝 pFoo
  }
}
```

遵照这个规则，基本上不会遇到反复拷贝 shared_ptr 导致的性能问题。另外由于 pFoo 是栈上对象，不可能被别的线程看到，那么读取始终是线程安全的。

析构动作在创建时被捕获 这是一个非常有用的特性，这意味着：

- 虚析构不再是必需的。
- shared_ptr<void> 可以持有任何对象，而且能安全地释放。
- shared_ptr 对象可以安全地跨越模块边界，比如从 DLL 里返回，而不会造成从模块 A 分配的内存在模块 B 里被释放这种错误。
- 二进制兼容性，即便 Foo 对象的大小变了，那么旧的客户代码仍然可以使用新的动态库，而无须重新编译。前提是 Foo 的头文件中不出现访问对象的成员的 inline 函数，并且 Foo 对象的由动态库中的 Factory 构造，返回其 shared_ptr。
- 析构动作可以定制。

最后这个特性的实现比较巧妙，因为 shared_ptr<T> 只有一个模板参数，而“析构行为”可以是函数指针、仿函数（functor）或者其他什么东西。这是泛型编程和面向对象编程的一次完美结合。有兴趣的读者可以参考 Scott Meyers 的文章 [12]。这个技术在后面的对象池中还会用到。

析构所在的线程 对象的析构是同步的，当最后一个指向 x 的 shared_ptr 离开其作用域的时候，x 会同时在同一个线程析构。这个线程不一定是对象诞生的线程。这个特性是把双刃剑：如果对象的析构比较耗时，那么可能会拖慢关键线程的速度（如果最后一个 shared_ptr 引发的析构发生在关键线程）；同时，我们可以用一个单独的线程来专门做析构，通过一个 BlockingQueue<shared_ptr<void> > 把对象的析构都转移到那个专用线程，从而解放关键线程。

现成的 RAII handle 我认为 RAII （资源获取即初始化）是 C++ 语言区别于其他所有编程语言的最重要的特性，一个不懂 RAII 的 C++ 程序员不是一个合格的 C++ 程序员。初学 C++ 的教条是“new 和 delete 要配对，new 了之后要记着 delete”；如果使用 RAII [CCS，条款 13]，要改成“每一个明确的资源配置动作（例如 new）都应该在单一语句中执行，并在该语句中立刻将配置获得的资源交给 handle 对象（如 shared_ptr），程序中一般不出现 delete”。shared_ptr 是管理共享资源的利器，需要注意避免循环引用，通常的做法是 owner 持有指向 child 的 shared_ptr，child 持有指向 owner 的 weak_ptr。

[12] http://www.artima.com/cppsource/top_cpp_aha_moments.html

1.11 对象池

假设有 Stock 类，代表一只股票的价格。每一只股票有一个唯一的字符串标识，比如 Google 的 key 是 "NASDAQ:GOOG"，IBM 是 "NYSE:IBM"。Stock 对象是个主动对象，它能不断获取新价格。为了节省系统资源，同一个程序里边每一只出现的股票只有一个 Stock 对象，如果多处用到同一只股票，那么 Stock 对象应该被共享。如果某一只股票没有再在任何地方用到，其对应的 Stock 对象应该析构，以释放资源，这隐含了“引用计数”。

为了达到上述要求，我们可以设计一个对象池 StockFactory [13]。它的接口很简单，根据 key 返回 Stock 对象。我们已经知道，在多线程程序中，既然对象可能被销毁，那么返回 shared_ptr 是合理的。自然地，我们写出如下代码（可惜是错的）。

```
// version 1: questionable code
class StockFactory : boost::noncopyable
{
 public:
  shared_ptr<Stock> get(const string& key);

 private:
  mutable MutexLock mutex_;
  std::map<string, shared_ptr<Stock> > stocks_;
};
```

get() 的逻辑很简单，如果在 stocks_ 里找到了 key，就返回 stocks_[key]；否则新建一个 Stock，并存入 stocks_[key]。

细心的读者或许已经发现这里有一个问题，Stock 对象永远不会被销毁，因为 map 里存的是 shared_ptr，始终有“铁丝”绑着。那么或许应该仿照前面 Observable 那样存一个 weak_ptr？比如

```
// // version 2: 数据成员修改为 std::map<string, weak_ptr<Stock> > stocks_;
shared_ptr<Stock> StockFactory::get(const string& key)
{
  shared_ptr<Stock> pStock;
  MutexLockGuard lock(mutex_);
  weak_ptr<Stock>& wkStock = stocks_[key];  // 如果 key 不存在，会默认构造一个
  pStock = wkStock.lock();  // 尝试把“棉线”提升为“铁丝”
  if (!pStock) {
    pStock.reset(new Stock(key));
    wkStock = pStock;  // 这里更新了 stocks_[key]，注意 wkStock 是个引用
  }
  return pStock;
}
```

[13] recipes/thread/test/Factory.cc 包含这里提到的各个版本。

这么做固然 Stock 对象是销毁了，但是程序却出现了轻微的内存泄漏，为什么？

因为 stocks_ 的大小只增不减，stocks_.size() 是曾经存活过的 Stock 对象的总数，即便活的 Stock 对象数目降为 0。或许有人认为这不算泄漏，因为内存并不是彻底遗失不能访问了，而是被某个标准库容器占用了。我认为这也算内存泄漏，毕竟是“战场”没有打扫干净。

其实，考虑到世界上的股票数目是有限的，这个内存不会一直泄漏下去，大不了把每只股票的对象都创建一遍，估计泄漏的内存也只有几兆字节。如果这是一个其他类型的对象池，对象的 key 的集合不是封闭的，内存就会一直泄漏下去。

解决的办法是，利用 shared_ptr 的定制析构功能。shared_ptr 的构造函数可以有一个额外的模板类型参数，传入一个函数指针或仿函数 d，在析构对象时执行 d(ptr)，其中 ptr 是 shared_ptr 保存的对象指针。shared_ptr 这么设计并不是多余的，因为反正要在创建对象时捕获释放动作，始终需要一个 bridge。

```
template<class Y, class D> shared_ptr::shared_ptr(Y* p, D d);
template<class Y, class D> void shared_ptr::reset(Y* p, D d);
// 注意 Y 的类型可能与 T 不同，这是合法的，只要 Y* 能隐式转换为 T*。
```

那么我们可以利用这一点，在析构 Stock 对象的同时清理 stocks_。

```
// version 3
class StockFactory : boost::noncopyable
{
  // 在 get() 中，将 pStock.reset(new Stock(key)); 改为：
  // pStock.reset(new Stock(key),
  //              boost::bind(&StockFactory::deleteStock, this, _1));  // ***

 private:
  void deleteStock(Stock* stock)
  {
    if (stock) {
      MutexLockGuard lock(mutex_);
      stocks_.erase(stock->key());
    }
    delete stock;  // sorry, I lied
  }
  // assuming StockFactory lives longer than all Stock's ...
  // ...
```

这里我们向 pStock.reset() 传递了第二个参数，一个 boost::function，让它在析构 Stock* p 时调用本 StockFactory 对象的 deleteStock 成员函数。

警惕的读者可能已经发现问题，那就是我们把一个原始的 StockFactory this 指针保存在了 boost::function 里（*** 处），这会有线程安全问题。如果这个 StockFactory 先于 Stock 对象析构，那么会 core dump。正如 Observer 在析构函数里去调用 Observable::unregister()，而那时 Observable 对象可能已经不存在了。

当然这也是能解决的，要用到 §1.11.2 介绍的弱回调技术。

1.11.1 enable_shared_from_this

StockFactory::get() 把原始指针 this 保存到了 boost::function 中（*** 处），如果 StockFactory 的生命期比 Stock 短，那么 Stock 析构时去回调 StockFactory::deleteStock 就会 core dump。似乎我们应该祭出惯用的 shared_ptr 大法来解决对象生命期问题，但是 StockFactory::get() 本身是个成员函数，如何获得一个指向当前对象的 shared_ptr<StockFactory> 对象呢？

有办法，用 enable_shared_from_this。这是一个以其派生类为模板类型实参的基类模板 [14]，继承它，this 指针就能变身为 shared_ptr。

```
class StockFactory : public boost::enable_shared_from_this<StockFactory>,
                     boost::noncopyable
{ /* ... */ };
```

为了使用 shared_from_this()，StockFactory 不能是 stack object，必须是 heap object 且由 shared_ptr 管理其生命期，即：

```
shared_ptr<StockFactory> stockFactory(new StockFactory);
```

万事俱备，可以让 this 摇身一变，化为 shared_ptr<StockFactory> 了。

```
// version 4
shared_ptr<Stock> StockFactory::get(const string& key)
{
  // change
  // pStock.reset(new Stock(key),
  //              boost::bind(&StockFactory::deleteStock, this, _1));
  // to
  pStock.reset(new Stock(key),
               boost::bind(&StockFactory::deleteStock,
                           shared_from_this(),
                           _1));
  // ...
```

这样一来，boost::function 里保存了一份 shared_ptr<StockFactory>，可以保证调用 StockFactory::deleteStock 的时候那个 StockFactory 对象还活着。

注意一点，shared_from_this() 不能在构造函数里调用，因为在构造 StockFactory 的时候，它还没有被交给 shared_ptr 接管。

最后一个问题，StockFactory 的生命期似乎被意外延长了。

[14] http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

1.11.2 弱回调

把 shared_ptr 绑（boost::bind）到 boost:function 里，那么回调的时候 StockFactory 对象始终存在，是安全的。这同时也延长了对象的生命期，使之不短于绑得的 boost:function 对象。

有时候我们需要“如果对象还活着，就调用它的成员函数，否则忽略之”的语意，就像 Observable::notifyObservers() 那样，我称之为“弱回调”。这也是可以实现的，利用 weak_ptr，我们可以把 weak_ptr 绑到 boost::function 里，这样对象的生命期就不会被延长。然后在回调的时候先尝试提升为 shared_ptr，如果提升成功，说明接受回调的对象还健在，那么就执行回调；如果提升失败，就不必劳神了。

使用这一技术的完整 StockFactory 代码如下：

```
class StockFactory : public boost::enable_shared_from_this<StockFactory>,
                     boost::noncopyable
{
 public:
  shared_ptr<Stock> get(const string& key)
  {
    shared_ptr<Stock> pStock;
    MutexLockGuard lock(mutex_);
    weak_ptr<Stock>& wkStock = stocks_[key];  // 注意 wkStock 是引用
    pStock = wkStock.lock();
    if (!pStock)
    {
      pStock.reset(new Stock(key),
                   boost::bind(&StockFactory::weakDeleteCallback,
                               boost::weak_ptr<StockFactory>(shared_from_this()),
                               _1));
      // 上面必须强制把 shared_from_this() 转型为 weak_ptr, 才不会延长生命期,
      // 因为 boost::bind 拷贝的是实参类型, 不是形参类型
      wkStock = pStock;
    }
    return pStock;
  }

 private:
  static void weakDeleteCallback(const boost::weak_ptr<StockFactory>& wkFactory,
                                 Stock* stock)
  {
    shared_ptr<StockFactory> factory(wkFactory.lock());  // 尝试提升
    if (factory)   // 如果 factory 还在, 那就清理 stocks_
    {
      factory->removeStock(stock);
    }
    delete stock;  // sorry, I lied
  }
```

```
  void removeStock(Stock* stock)
  {
    if (stock)
    {
      MutexLockGuard lock(mutex_);
      stocks_.erase(stock->key());
    }
  }

 private:
  mutable MutexLock mutex_;
  std::map<string, weak_ptr<Stock> > stocks_;
};
```

两个简单的测试：

```
void testLongLifeFactory()
{
  shared_ptr<StockFactory> factory(new StockFactory);
  {
    shared_ptr<Stock> stock = factory->get("NYSE:IBM");
    shared_ptr<Stock> stock2 = factory->get("NYSE:IBM");
    assert(stock == stock2);
    // stock destructs here
  }
  // factory destructs here
}

void testShortLifeFactory()
{
  shared_ptr<Stock> stock;
  {
    shared_ptr<StockFactory> factory(new StockFactory);
    stock = factory->get("NYSE:IBM");
    shared_ptr<Stock> stock2 = factory->get("NYSE:IBM");
    assert(stock == stock2);
    // factory destructs here
  }
  // stock destructs here
}
```

这下完美了，无论 `Stock` 和 `StockFactory` 谁先挂掉都不会影响程序的正确运行。这里我们借助 `shared_ptr` 和 `weak_ptr` 完美地解决了两个对象相互引用的问题。

当然，通常 Factory 对象是个 singleton，在程序正常运行期间不会销毁，这里只是为了展示弱回调技术 [15]，这个技术在事件通知中非常有用。

本节的 `StockFactory` 只有针对单个 `Stock` 对象的操作，如果程序需要遍历整个 `stocks_`，稍不注意就会造成死锁或数据损坏（§2.1），请参考 §2.8 的解决办法。

[15] 通用的弱回调封装见 recipes/thread/WeakCallback.h，用到了 C++11 的 variadic template 和 rvalue reference。

1.12 替代方案

除了使用 shared_ptr/weak_ptr，要想在 C++ 里做到线程安全的对象回调与析构，可能的办法有以下一些。

1. 用一个全局的 façade 来代理 Foo 类型对象访问，所有的 Foo 对象回调和析构都通过这个 façade 来做，也就是把指针替换为 objId/handle，每次要调用对象的成员函数的时候先 check-out，用完之后再 check-in [16]。这样理论上能避免 race condition，但是代价很大。因为要想把这个 façade 做成线程安全的，那么必然要用互斥锁。这样一来，从两个线程访问两个不同的 Foo 对象也会用到同一个锁，让本来能够并行执行的函数变成了串行执行，没能发挥多核的优势。当然，可以像 Java 的 ConcurrentHashMap 那样用多个 buckets，每个 bucket 分别加锁，以降低 contention。
2. §1.4 提到的“只创建不销毁”手法，实属无奈之举。
3. 自己编写引用计数的智能指针 [17]。本质上是重新发明轮子，把 shared_ptr 实现一遍。正确实现线程安全的引用计数智能指针不是一件容易的事情，而高效的实现就更加困难。既然 shared_ptr 已经提供了完整的解决方案，那么似乎没有理由抗拒它。
4. 将来在 C++11 里有 unique_ptr，能避免引用计数的开销，或许能在某些场合替换 shared_ptr。

其他语言怎么办

有垃圾回收就好办。Google 的 Go 语言教程明确指出，没有垃圾回收的并发编程是困难的（Concurrency is hard without garbage collection）。但是由于指针算术的存在，在 C/C++ 里实现全自动垃圾回收更加困难。而那些天生具备垃圾回收的语言在并发编程方面具有明显的优势，Java 是目前支持并发编程最好的主流语言，它的 util.concurrent 库和内存模型是 C++11 效仿的对象。

1.13 心得与小结

学习多线程程序设计远远不是看看教程了解 API 怎么用那么简单，这最多“主要是为了读懂别人的代码，如果自己要写这类代码，必须专门花时间严肃、认真、系

[16] 这是 Jeff Grossman 在《A technique for safe deletion with object locking》一文中提出的办法 [Gr00]。

[17] 见 http://blog.csdn.net/solstice/article/details/5238671#comments 后面的评论。

统地学习，严禁半桶水上阵”（孟岩）[18]。一般的多线程教程上都会提到要让加锁的区域足够小，这没错，问题是如何找出这样的区域并加锁，本章 §1.9 举的安全读写 shared_ptr 可算是一个例子。

据我所知，目前 C++ 没有特别好的多线程领域专著，但 C 语言有，Java 语言也有。《Java Concurrency in Practice》[JCP] 是我读过的写得最好的书，内容足够新，可读性和可操作性俱佳。C++ 程序员反过来要向 Java 学习，多少有些讽刺。除了编程书，操作系统教材也是必读的，至少要完整地学习一本经典教材的相关章节，可从《操作系统设计与实现》、《现代操作系统》、《操作系统概念》任选一本，了解各种同步原语、临界区、竞态条件、死锁、典型的 IPC 问题等等，防止闭门造车。

分析可能出现的 race condition 不仅是多线程编程的基本功，也是设计分布式系统的基本功，需要反复历练，形成一定的思考范式，并积累一些经验教训，才能少犯错误。这是一个快速发展的领域，要不断吸收新知识，才不会落伍。单 CPU 时代的多线程编程经验到了多 CPU 时代不一定有效，因为多 CPU 能做到真正的并行执行，每个 CPU 看到的事件发生顺序不一定完全相同。正如狭义相对论所说的每个观察者都有自己的时钟，在不违反因果律的前提下，可能发生十分违反直觉的事情。

尽管本章通篇在讲如何安全地使用（包括析构）跨线程的对象，但我建议尽量减少使用跨线程的对象，我赞同水木网友 ilovecpp 说的：“用流水线，生产者消费者，任务队列这些有规律的机制，最低限度地共享数据。这是我所知最好的多线程编程的建议了。”

不用跨线程的对象，自然不会遇到本章描述的各种险态。如果迫不得已要用，希望本章内容能对你有帮助。

小结

- 原始指针暴露给多个线程往往会造成 race condition 或额外的簿记负担。
- 统一用 shared_ptr/scoped_ptr 来管理对象的生命期，在多线程中尤其重要。

[18] 孟岩《快速掌握一个语言最常用的 50%》博客，这篇博客（http://blog.csdn.net/myan/article/details/3144661）的其他文字也很有趣味：“粗粗看看语法，就撸起袖子开干，边查 Google 边学习”这种路子也有问题，在对于这种语言的脾气秉性还没有了解的情况下大刀阔斧地拼凑代码，写出来的东西肯定不入流。说穿新鞋走老路，新瓶装旧酒，那都是小问题，真正严重的是这样的程序员可以在短时间内堆积大量充满缺陷的垃圾代码。由于通常开发阶段的测试完备程度有限，这些垃圾代码往往能通过这个阶段，从而潜伏下来，在后期成为整个项目的“毒瘤”，反反复复让后来的维护者陷入西西弗斯困境。……其实真正写程序不怕完全不会，最怕一知半解地去攒解决方案。因为你完全不会，就自然会去认真查书学习，如果学习能力好的话，写出来的代码质量不会差。而一知半解，自己动手“土法炼钢”，那搞出来的基本上都是“废铜烂铁”。

- shared_ptr 是值语意，当心意外延长对象的生命期。例如 boost::bind 和容器都可能拷贝 shared_ptr。
- weak_ptr 是 shared_ptr 的好搭档，可以用作弱回调、对象池等。
- 认真阅读一遍 boost::shared_ptr 的文档，能学到很多东西：
  http://www.boost.org/doc/libs/release/libs/smart_ptr/shared_ptr.htm
- 保持开放心态，留意更好的解决办法，比如 C++11 引入的 unique_ptr。忘掉已被废弃的 auto_ptr。

shared_ptr 是 TR1 的一部分，即 C++ 标准库的一部分，值得花一点时间去学习掌握 [19]，对编写现代的 C++ 程序有莫大的帮助。我个人的经验是，一周左右就能基本掌握各种用法与常见陷阱，比学 STL 还快。网络上有一些对 shared_ptr 的批评，那可以算作故意误用的例子，就好比故意访问失效的迭代器来证明 std::vector 不安全一样。

正确使用标准库（含 shared_ptr）作为自动化的内存/资源管理器，解放大脑，从此告别内存错误。

1.14 Observer 之谬

本章 §1.8 把 shared_ptr/weak_ptr 应用到 Observer 模式中，部分解决了其线程安全问题。我用 Observer 举例，因为这是一个广为人知的设计模式，但是它有本质的问题。

Observer 模式的本质问题在于其面向对象的设计。换句话说，我认为正是面向对象（OO）本身造成了 Observer 的缺点。Observer 是基类，这带来了非常强的耦合，强度仅次于友元（friend）。这种耦合不仅限制了成员函数的名字、参数、返回值，还限制了成员函数所属的类型（必须是 Observer 的派生类）。

Observer class 是基类，这意味着如果 Foo 想要观察两个类型的事件（比如时钟和温度），需要使用多继承。这还不是最糟糕的，如果要重复观察同一类型的事件（比如 1 秒一次的心跳和 30 秒一次的自检），就要用到一些伎俩来 work around，因为不能从一个 Base class 继承两次。

[19] 孟岩在《垃圾收集机制批判》中说：在 C++ 中，new 出来的对象没有 delete，这就导致了 memory leak。但是 C++ 早就有了克服这一问题的办法——smart pointer。通过使用标准库里设计精致的各种 STL 容器，还有例如 Boost 库（差不多是个准标准库了）中的 4 个 smart pointers，C++ 程序员只要花上一个星期的时间学习最新的资料，就可以拍着胸脯说：“我写的程序没有 memory leak!”。

现在的语言一般可以绕过 Observer 模式的限制，比如 Java 可以用匿名内部类，Java 8 用 Closure，C# 用 delegate，C++ 用 `boost::function/ boost::bind` [20] 。

在 C++ 里为了替换 Observer，可以用 Signal/Slots，我指的不是 QT 那种靠语言扩展的实现，而是完全靠标准库实现的 thread safe、race condition free、thread contention free 的 Signal/Slots，并且不强制要求 `shared_ptr` 来管理对象，也就是说完全解决了 §1.8 列出的 Observer 遗留问题。这会用到 §2.8 介绍的“借 `shared_ptr` 实现 copy-on-write”技术。

在 C++11 中，借助 variadic template，实现最简单（trivial）的一对多回调可谓不费吹灰之力，代码如下。

recipes/thread/SignalSlotTrivial.h

```
template<typename Signature>
class SignalTrivial;

// NOT thread safe !!!
template <typename RET, typename... ARGS>
class SignalTrivial<RET(ARGS...)>
{
 public:
  typedef std::function<void (ARGS...)> Functor;

  void connect(Functor&& func)
  {
    functors_.push_back(std::forward<Functor>(func));
  }

  void call(ARGS&&... args)
  {
    for (const Functor& f: functors_)
    {
      f(args...);
    }
  }

 private:
  std::vector<Functor> functors_;
};
```

recipes/thread/SignalSlotTrivial.h

我们不难把以上基本实现扩展为线程安全的 Signal/Slots，并且在 `Slot` 析构时自动 unregister。有兴趣的读者可仔细阅读完整实现的代码（recipes/thread/SignalSlot.h）。

[20] 见 §11.5“以 `boost::function` 和 `boost::bind` 取代虚函数”，还有孟岩的《function/bind 的救赎（上）》（http://blog.csdn.net/myan/article/details/5928531）。

结语

《C++ 沉思录》（*Ruminations on C++* 中文版）的附录是王曦和孟岩对作者夫妇二人的采访，在被问到“请给我们三个你们认为最重要的建议”时，Koenig 和 Moo 的第一个建议是“避免使用指针”。我 2003 年读到这段时，理解不深，觉得固然使用指针容易造成内存方面的问题，但是完全不用也是做不到的，毕竟 C++ 的多态要通过指针或引用来起效。6 年之后重新拾起来，发现大师的观点何其深刻，不免掩卷长叹。

这本书详细地介绍了 handle/body idiom，这是编写大型 C++ 程序的必备技术，也是实现物理隔离的“法宝”，值得细读。

目前来看，用 `shared_ptr` 来管理资源在国内 C++ 界似乎并不是一种主流做法，很多人排斥智能指针，视其为“洪水猛兽”（这或许受了 `auto_ptr` 的垃圾设计的影响）。据我所知，很多 C++ 项目还是手动管理内存和资源，因此我觉得有必要把我认为好的做法分享出来，让更多的人尝试并采纳。我觉得 `shared_ptr` 对于编写线程安全的 C++ 程序是至关重要的，不然就得“土法炼钢”，自己“重新发明轮子[21]”。这让我想起了 2001 年前后 STL 刚刚传入国内，大家也是很犹豫，觉得它性能不高，使用不便，还不如自己造的容器类。10 年过去了，现在 STL 已经是主流，大家也适应了迭代器、容器、算法、适配器、仿函数这些“新”名词、“新”技术，开始在项目中普遍使用（至少用 `vector` 代替数组嘛）。我希望，几年之后人们回头看本章内容，觉得“怎么讲的都是常识”，那我的写作目的也就达到了。

[21] http://en.wikipedia.org/wiki/Reinventing_the_wheel

第 2 部分

muduo 网络库

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。

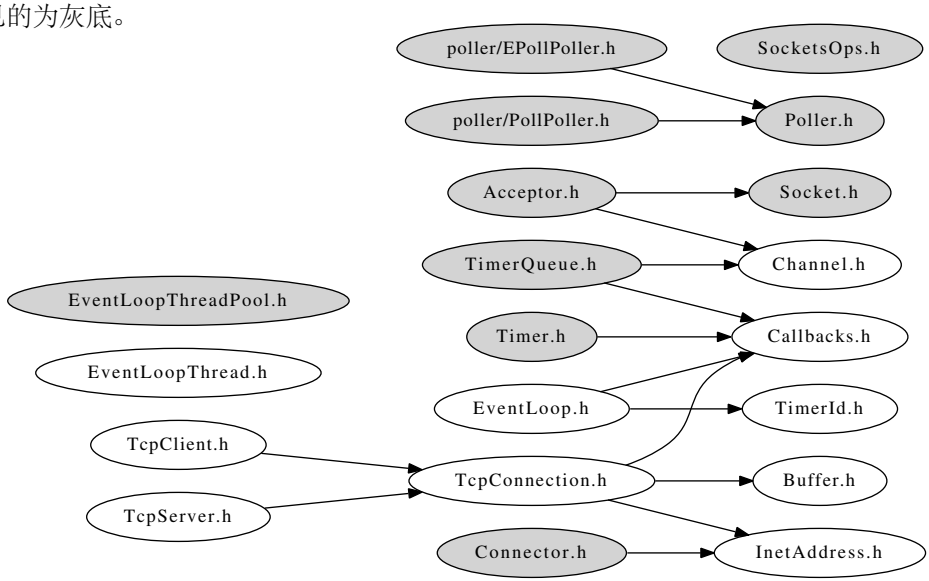


图 6-1

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。

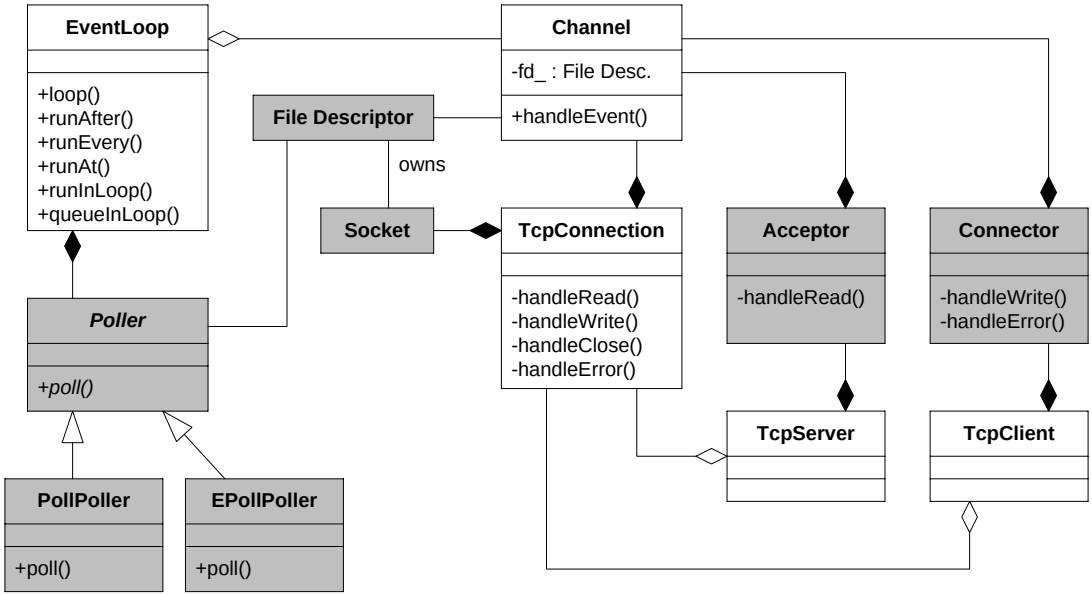


图 6-2

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：

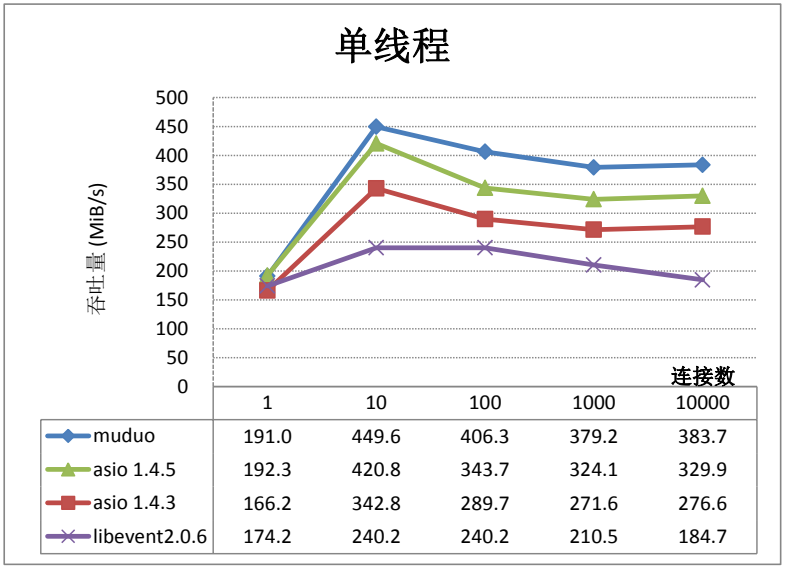
现在的 CPU 很快，即便是单线程单 TCP 连接也能把千兆以太网的带宽跑满。如果用两台机器，所有的吞吐量测试结果都将是 110MiB/s，失去了对比的意义。（用 Python 也能跑出同样的吞吐量，或许可以对比哪个库占的 CPU 少。）

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

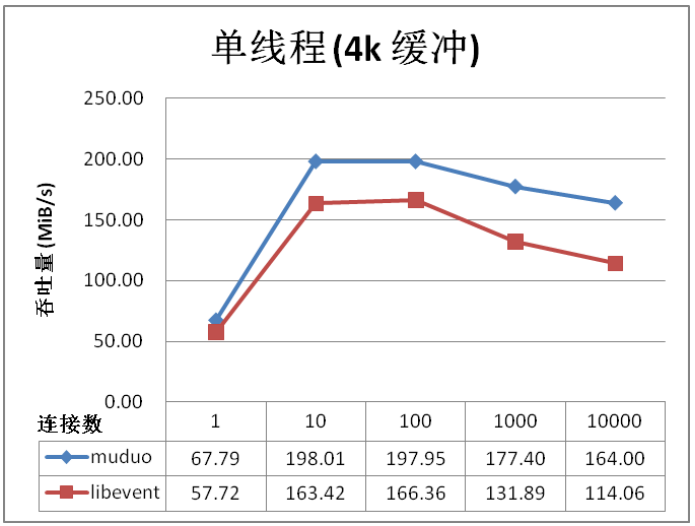


	1	10	100	1000	10000
muduo	191.0	449.6	406.3	379.2	383.7
asio 1.4.5	192.3	420.8	343.7	324.1	329.9
asio 1.4.3	166.2	342.8	289.7	271.6	276.6
libevent2.0.6	174.2	240.2	240.2	210.5	184.7

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。

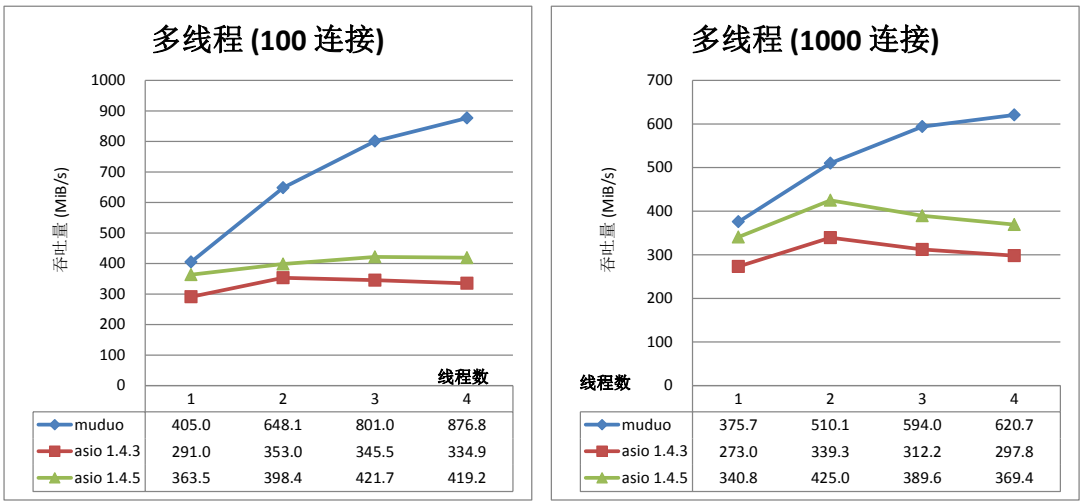


图 6-5

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

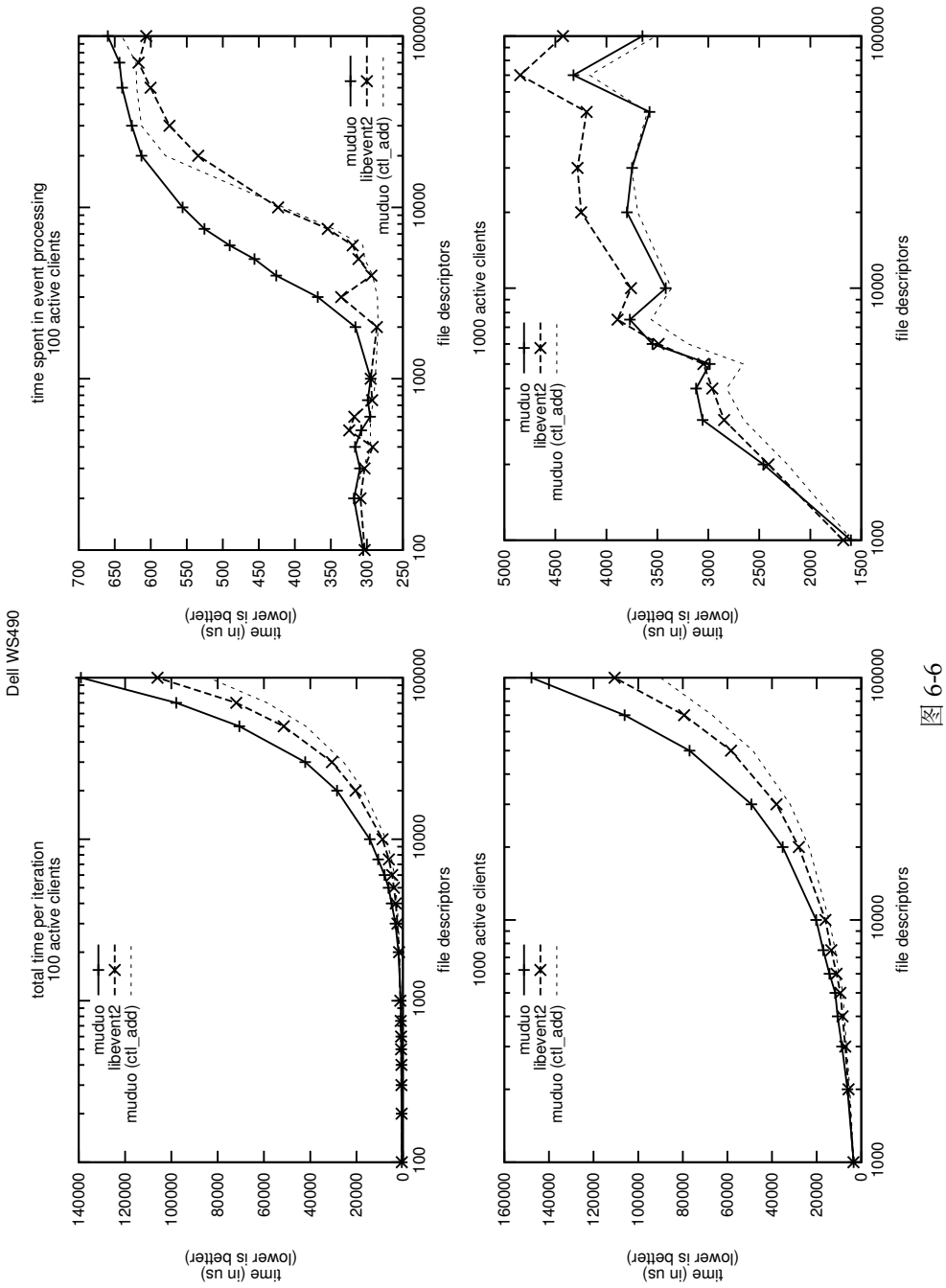

Dell WS490
total time per iteration
100 active clients
time spent in event processing
100 active clients
1000 active clients
1000 active clients
time (in us)
(lower is better)
file descriptors
muduo
libevent2
muduo (ctl_add)

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

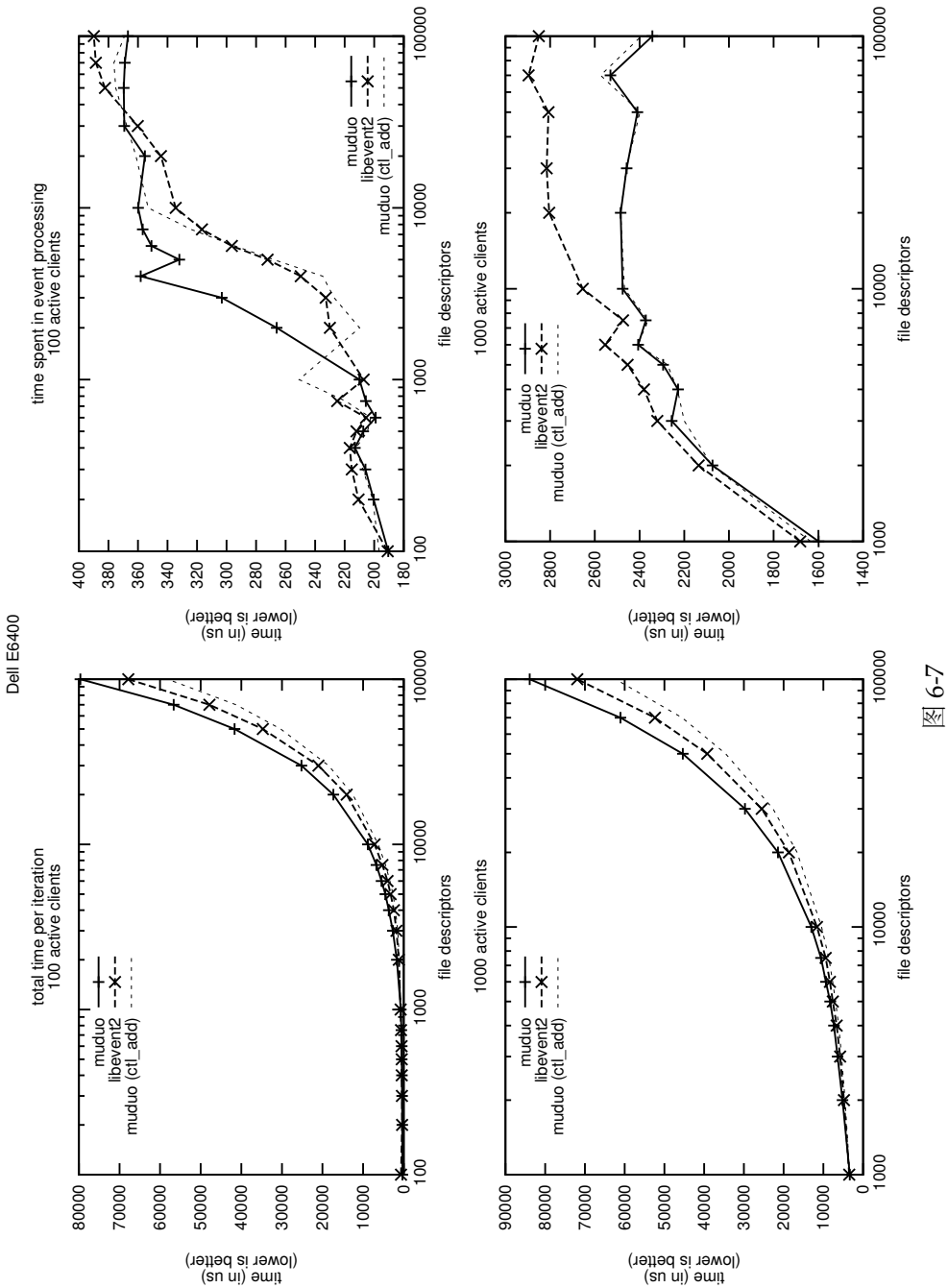

Dell E6400
total time per iteration
100 active clients
time spent in event processing
100 active clients
1000 active clients
1000 active clients
muduo
libevent2
muduo (ctl_add)
time (in us)
(lower is better)
file descriptors

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：

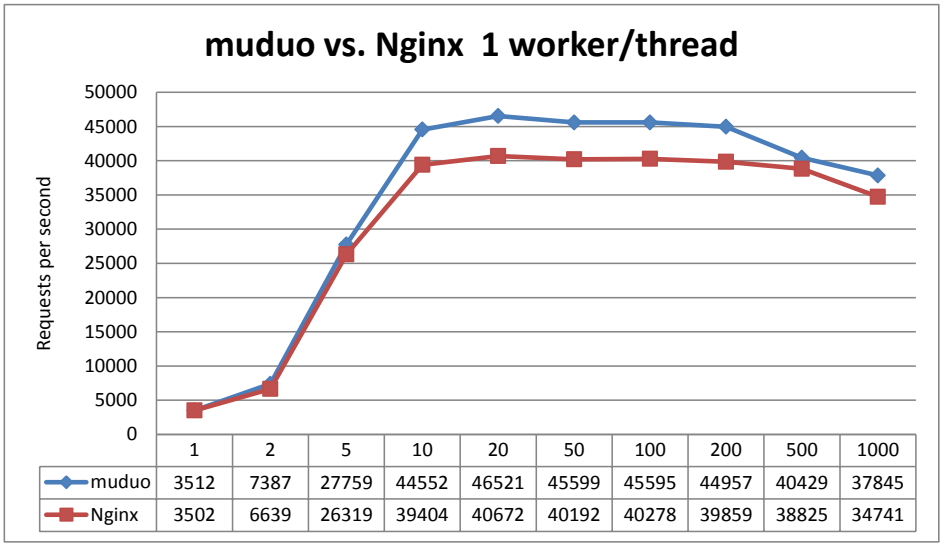
```
#user  nobody;
worker_processes  4;

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
```

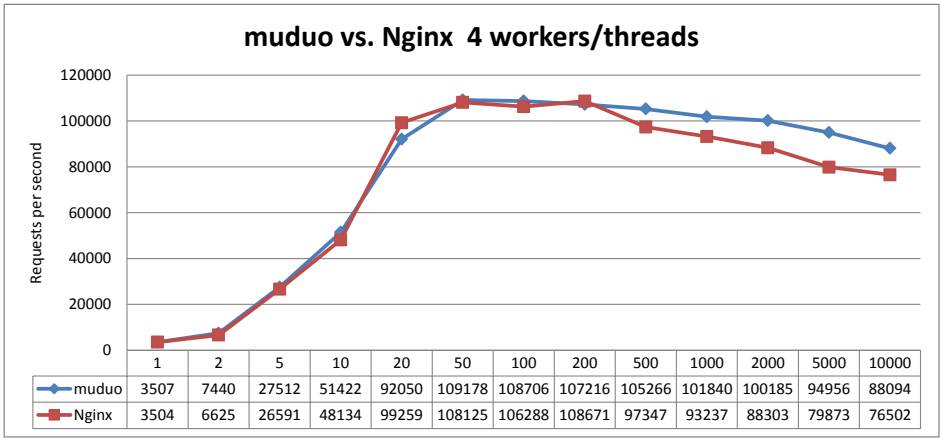
先测试单线程的性能（见图 6-8），横轴是并发连接数，纵轴为每秒完成的 HTTP 请求响应数目，下同。在测试期间，ab 的 CPU 使用率低于 70%，客户端游刃有余。



	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）：

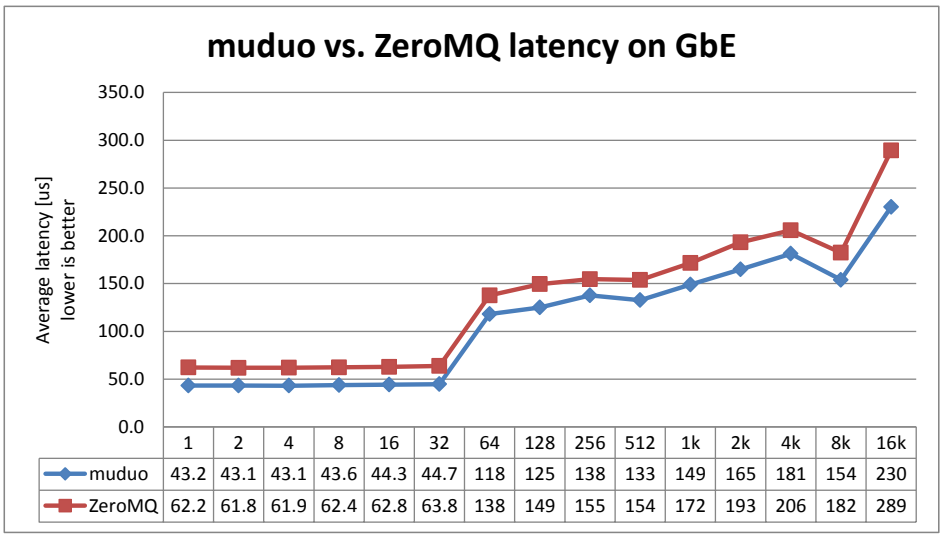
- 10 000 并发连接，4 workers/threads，muduo 是 4 × 83%，Nginx 是 4 × 75%
- 1000 并发连接，4 workers/threads，muduo 是 4 × 85%，Nginx 是 4 × 78%

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。

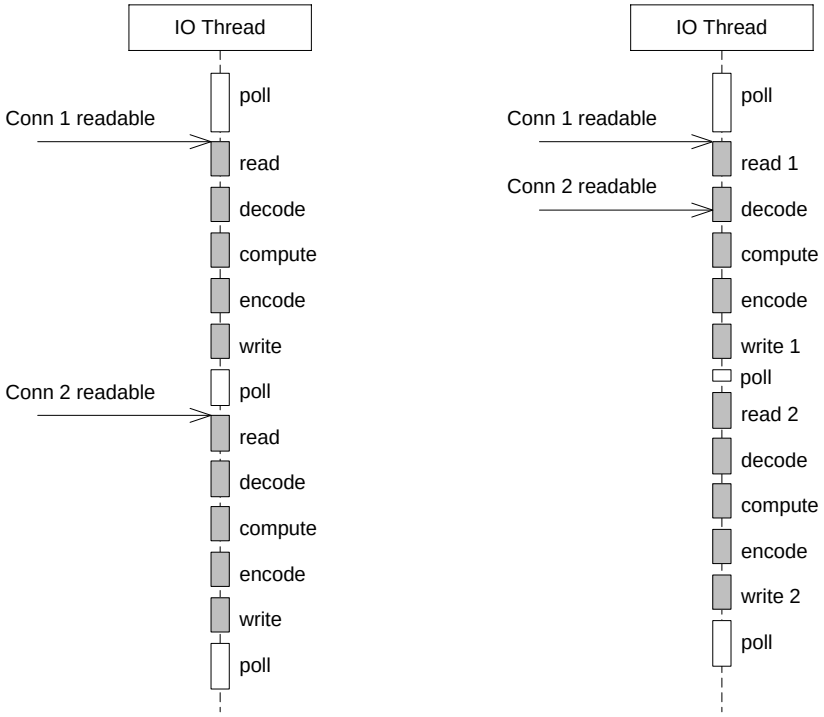


图 6-11

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。

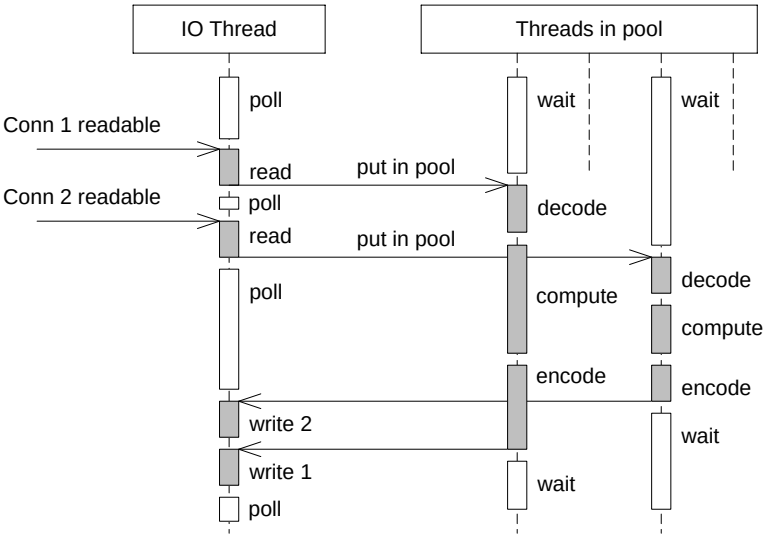


图 6-12

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。

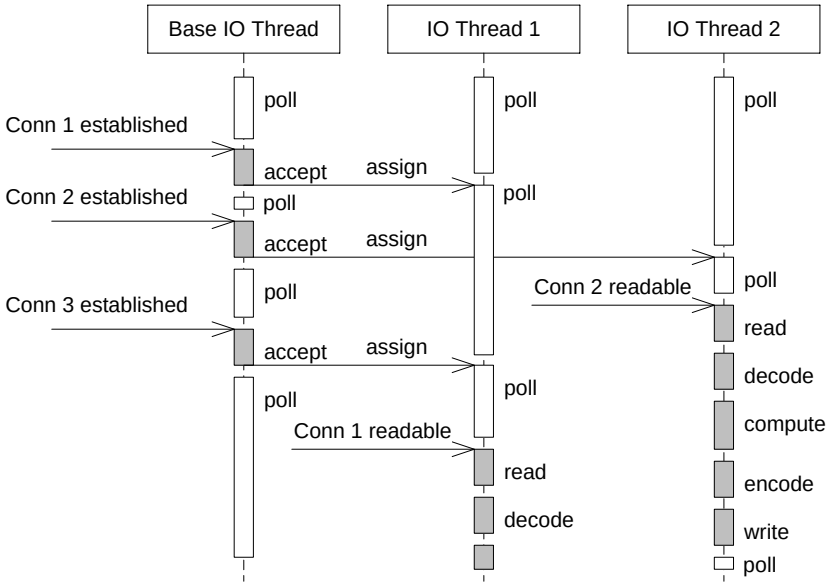


图 6-13

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。

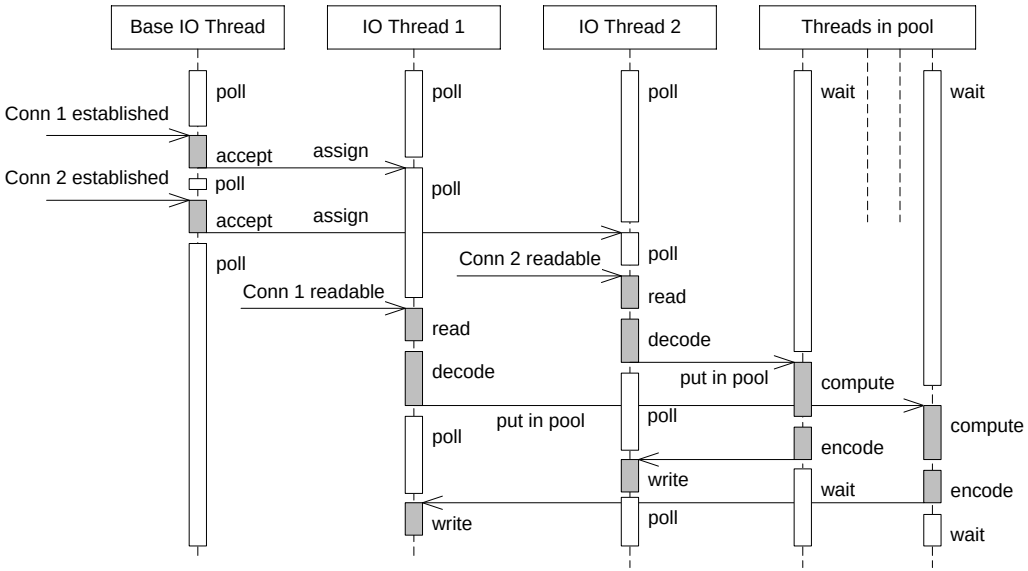


图 6-14

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。

第 3 部分

工程实践经验谈

采用 pimpl 多了一道 explicit forward 的手续，带来的好处是可扩展性与二进制兼容性，这通常是划算的。pimpl 扮演了编译器防火墙的作用。

pimpl 不仅 C++ 语言可以用，C 语言的库同样可以用，一样带来二进制兼容性的好处，比如 libevent2 中的 struct event_base 是个 opaque pointer，客户端看不到其成员，都是通过 libevent 的函数和它打交道，这样库的版本升级比较容易做到二进制兼容。

为什么 non-virtual 函数比 virtual 函数更健壮？因为 virtual function 是 bind-by-vtable-offset，而 non-virtual function 是 bind-by-name。加载器（loader）会在程序启动时做决议（resolution），通过 mangled name 把可执行文件和动态库链接到一起。就像使用 Internet 域名比使用 IP 地址更能适应变化一样。

万一要跨语言怎么办？很简单，暴露 C 语言的接口。Java 有 JNI 可以调用 C 语言的代码；Python/Perl/Ruby 等的解释器都是 C 语言编写的，使用 C 函数也不在话下。C 函数是 Linux 下的万能接口。

本节只谈了使用 class 为接口，其实用 free function 有时候更好（比如 muduo/base/Timestamp.h 除了定义 class Timestamp 外，还定义了 muduo::timeDifference() 等 free function），这也是 C++ 比 Java 等纯面向对象语言优越的地方。

11.5 以 boost::function 和 boost::bind 取代虚函数

本节的中心思想是“面向对象的继承就像一条贼船，上去就下不来了”，而借助 boost::function 和 boost::bind，大多数情况下，你都不用上“贼船”。

boost::function 和 boost::bind 已经纳入了 std::tr1，这或许是 C++11 最值得期待的功能，它将彻底改变 C++ 库的设计方式，以及应用程序的编写方式。

Scott Meyers 的 [EC3，条款 35] 提到了以 boost::function 和 boost:bind 取代虚函数的做法，另见孟岩的《function/bind 的救赎（上）》[11]、《回复几个问题》[12] 中的“四个半抽象”，这里谈谈我自己使用的感受。

我对面向对象的“继承”和“多态”的态度是能不用就不用，因为很难纠正错误。如果有一棵类型继承树（class hierarchy），人们在一开始设计时就得考虑各个 class 在树上的位置。随着时间的推衍，原来正确的决定有可能变成错误的。但是更正这个错误的代价可能很高。要想把这个 class 在继承树上从一个节点挪到另一个节点，可

[11] http://blog.csdn.net/myan/archive/2010/10/09/5928531.aspx

[12] http://blog.csdn.net/myan/archive/2010/09/14/5884695.aspx

能要触及所有用到这个 class 的客户代码，所有用到其各层基类的客户代码，以及从这个 class 派生出来的全部 class 的代码。简直是牵一发而动全身，在 C++ 缺乏良好重构工具的语言下，有时候只好保留错误，用些 wrapper 或者 adapter 来掩盖之。久而久之，设计越来越烂，最后只好推倒重来 [13]。解决办法之一就是不采用基于继承的设计，而是写一些容易使用也容易修改的具体类。

总之，继承和虚函数是万恶之源，这条“贼船”上去就不容易下来。不过还好，在 C++ 里我们有别的办法: 以 `boost::function` 和 `boost::bind` 取代虚函数。

用“继承树”这种方式来建模，确实是基于概念分类的思想。“分类”似乎是西方哲学一早就有的思想，影响深远，这种思想估计可以上溯到古希腊时期。

- 比如电影，可以分为科幻片、爱情片、伦理片、战争片、灾难片、恐怖片等等。
- 比如生物，按小学知识可以分为动物和植物，动物又可以分为有脊椎动物和无脊椎动物，有脊椎动物又分为鱼类、两栖类、爬行类、鸟类和哺乳类等。
- 又比如技术书籍分为电子类、通信类、计算机类等等，计算机书籍又可分为编程语言、操作系统、数据结构、数据库、网络技术等等。

这种分类法或许是早期面向对象方法的模仿对象。这种思考方式的本质困难在于：某些物体很难准确分类，似乎有不止一个分类适合它。而且不同的人看法可能不同，比如一部科幻悬疑片到底科幻的成分重还是悬疑的成分重，到底该归入哪一类。

在编程方面，情况更糟，因为这个“物体 x”是变化的，一开始分入 A 类可能是合理的（x ″is-a″ A），随着功能演化，分入 B 类或许更合适（x is more like a B），但是这种改动对现有代码的代价已经太高了（特别对于 C++）。

在传统的面向对象语言中，可以用继承多个 interfaces 来缓解分错类的代价，使得一物多用。但是某些语言限制了基类只能有一个，在新增类型时可能会遇到麻烦，见星巴克卖鸳鸯奶茶的例子 [14]。

现代编程语言这一步走得更远，Ruby 的 duck typing 和 Google Go 的无继承 [15] 都可以看作以 tag 取代分类（层次化的类型）的代表。一个 object 只要提供了相应的 operations，就能当做某种东西来用，不需要显式地继承或实现某个接口。这确实是一种进步。

[13] Linus 在 2007 年炮轰 C++ 时说：“（C++ 面向对象）导致低效的抽象编程模型，可能在两年之后你会注意到有些抽象效果不怎么样，但是所有代码已经依赖于围绕它设计的‘漂亮’对象模型了，如果不重写应用程序，就无法改正。”（译文引自 http://blog.csdn.net/turingbook/article/details/1775488）

[14] http://www.cnblogs.com/Solstice/archive/2011/04/22/2024791.html

[15] http://golang.org/doc/go_lang_faq.html#inheritance

对于 C++ 的四种范式，我现在基本只把它当 better C 和 data abstraction 来用。OO 和 GP 可以在非常小的范围内使用，只要暴露的接口是 object based（甚至 global function）就行。

以上谈了设计层面，再来说一说实现层面。

在传统的 C++ 程序中，事件回调是通过虚函数进行的。网络库往往会定义一个或几个抽象基类（Handler class），其中声明了一些（纯）虚函数，如 onConnect()、onDisconnect()、onMessage()、onTimer() 等等。使用者需要继承这些基类，并覆写（override）这些虚函数，以获得事件回调通知。由于 C++ 的动态绑定只能通过指针和引用实现，使用者必须把派生类（MyHandler）对象的指针或引用隐式转换为基类（Handler）的指针或引用，再注册到网络库中。MyHandler 对象通常是动态创建的，位于堆上，用完后需要 delete。网络库调用基类的虚函数，通过动态绑定机制实际调用的是用户在派生类中 override 的虚函数，这也是各种 OO framework 的通行做法。这种方式在 Java 这种纯面向对象语言中是正当做法 [16]。但是在 C++ 这种非 GC 语言中，使用虚函数作为事件回调接口有其本质困难，即如何管理派生类对象的生命期。在这种接口风格中，MyHandler 对象的所有权和生命期很模糊，到底谁（用户还是网络库）有权力释放它呢？有的网络库甚至出现了 delete this; 这种代码，让人捏一把汗：如何才能保证此刻程序的其他地方没有保存着这个即将销毁的对象的指针呢？另外，如果网络库需要自己创建 MyHandler 对象（比方说需要为每个 TCP 连接创建一个 MyHandler 对象），那么就得定义另外一个抽象基类 HandlerFactory，用户要从它派生出 MyHandlerFactory，再把后者的指针或引用注册到网络库中。以上这些都是面向对象编程的常规思路，或许大家已经习以为常。

在现代 C++ 中（指 2005 年 TR1 之后，不是最新的 C++11），事件回调有了新的推荐做法，即 boost::function + boost::bind （即 std::tr1::function + std::tr1::bind，也是最新 C++11 中的 std::function + std::bind），这种方式的一个明显优点是不必担心对象的生存期。muduo 正是用 boost::function 来表示事件回调的，包括 TCP 网络编程的三个半 IO 事件和定时器事件等。用户代码可以传入签名相同的全局函数，也可以借助 boost::bind 把对象的成员函数传给网络库作为事件回调的接受方。这种接口方式对用户代码的 class 类型没有限制（不必从特定的基类派生），对成员函数名也没有限制，只对函数签名有部分限制。这样自然也解决了空悬指针的难题，因为传给网络库的都是具有值语义的 boost::function 对象。从这个意义上说，muduo 不是一个面向对象的库，而是一个基于对象的库。因为 muduo 暴露的接口都是一个个的具体类，完全没有虚函数（无论是调用还是回调）。

[16] Java 8 也有新的 Closure 语法，C# 从一诞生就有 delegate。

言归正传，说说 boost::function 和 boost::bind 取代虚函数的具体做法。

11.5.1 基本用途

boost::function 就像 C# 里的 delegate，可以指向任何函数，包括成员函数。当用 bind 把某个成员函数绑到某个对象上时，我们得到了一个 closure（闭包）。例如：

```
class Foo
{
 public:
  void methodA();
  void methodInt(int a);
  void methodString(const string& str);
};

class Bar
{
 public:
  void methodB();
};

boost::function<void()> f1; // 无参数，无返回值

Foo foo;
f1 = boost::bind(&Foo::methodA, &foo);
f1(); // 调用 foo.methodA();

Bar bar;
f1 = boost::bind(&Bar::methodB, &bar);
f1(); // 调用 bar.methodB();

f1 = boost::bind(&Foo::methodInt, &foo, 42);
f1(); // 调用 foo.methodInt(42);

f1 = boost::bind(&Foo::methodString, &foo, "hello");
f1(); // 调用 foo.methodString("hello")
// 注意，bind 拷贝的是实参类型 (const char*)，不是形参类型 (string)
// 这里形参中的 string 对象的构造发生在调用 f1 的时候，而非 bind 的时候，
// 因此要留意 bind 的实参 (cosnt char*) 的生命期，它应该不短于 f1 的生命期。
// 必要时可通过 bind(&Foo::methodString, &foo, string(aTempBuf)) 来保证安全

boost::function<void(int)> f2; // int 参数，无返回值
f2 = boost::bind(&Foo::methodInt, &foo, _1);
f2(53); // 调用 foo.methodInt(53);
```

如果没有 boost::bind，那么 boost::function 就什么都不是；而有了 bind，“同一个类的不同对象可以 delegate 给不同的实现，从而实现不同的行为”（孟岩），简直就无敌了。

11.5.2 对程序库的影响

程序库的设计不应该给使用者带来不必要的限制（耦合），而继承是第二强的一种耦合（最强耦合的是友元）。如果一个程序库限制其使用者必须从某个 class 派生，那么我觉得这是一个糟糕的设计。不巧的是，目前不少 C++ 程序库就是这么做的。

例 1：线程库

常规 OO 设计 写一个 Thread base class，含有（纯）虚函数 Thread::run()，然后应用程序派生一个 derived class，覆写 run()。程序里的每一种线程对应一个 Thread 的派生类。例如 Java 的 Thread class 可以这么用。

缺点：如果一个 class 的三个 method 需要在三个不同的线程中执行，就得写 helper class(es) 并玩一些 OO 把戏。

基于 boost::function 的设计 令 Thread 是一个具体类，其构造函数接受 ThreadCallback 对象。应用程序只需提供一个能转换为 ThreadCallback 的对象（可以是函数），即可创建一份 Thread 实体，然后调用 Thread::start() 即可。Java 的 Thread 也可以这么用，传入一个 Runnable 对象。C# 的 Thread 只支持这一种用法，构造函数的参数是 delegate ThreadStart。boost::thread 也只支持这种用法。

```
// 一个基于 boost::function 的 Thread class 基本结构
class Thread
{
 public:
  typedef boost::function<void()> ThreadCallback;

  Thread(ThreadCallback cb)
    : cb_(cb)
  { }

  void start()
  {
    /* some magic to call run() in new created thread */
  }

 private:

  void run()
  {
    cb_();
  }

  ThreadCallback cb_;
  // ...
};
```

使用方式：

```
class Foo // 不需要继承
{
 public:
  void runInThread();
  void runInAnotherThread(int)
};

Foo foo;
Thread thread1(boost::bind(&Foo::runInThread, &foo));
Thread thread2(boost::bind(&Foo::runInAnotherThread, &foo, 43));
thread1.start();  // 在两个线程中分别运行两个成员函数
thread2.start();
```

例 2：网络库

以 boost::function 作为桥梁，NetServer class 对其使用者没有任何类型上的限制，只对成员函数的参数和返回类型有限制。使用者 EchoService 也完全不知道 NetServer 的存在，只要在 main() 里把两者装配到一起，程序就跑起来了。[17]

network library

```
class Connection;
class NetServer : boost::noncopyable
{
 public:
  typedef boost::function<void (Connection*)> ConnectionCallback;
  typedef boost::function<void (Connection*, const void*, int len)> MessageCallback;

  NetServer(uint16_t port);
  ~NetServer();
  void registerConnectionCallback(const ConnectionCallback&);
  void registerMessageCallback(const MessageCallback&);
  void sendMessage(Connection*, const void* buf, int len);

 private:
  // ...
};
```

network library

user code

```
class EchoService
{
 public:
  // 符合 NetServer::sendMessage 的原型
  typedef boost::function<void(Connection*, const void*, int)> SendMessageCallback;

  EchoService(const SendMessageCallback& sendMsgCb)
    : sendMessageCb_(sendMsgCb) // 保存 boost::function
  { }
```

[17] 本小节内容写得比较早，那会儿我还没有开始写 muduo，所以该例子与现在的代码有些脱节。

```
  // 符合 NetServer::MessageCallback 的原型
  void onMessage(Connection* conn, const void* buf, int size)
  {
    printf("Received Msg from Connection %d: %.*s\n",
           conn->id(), size, (const char*)buf);
    sendMessageCb_(conn, buf, size); // echo back
  }

  // 符合 NetServer::ConnectionCallback 的原型
  void onConnection(Connection* conn)
  {
    printf("Connection from %s:%d is %s\n", conn->ipAddr(), conn->port(),
           conn->connected() ? "UP" : "DOWN");
  }

 private:
  SendMessageCallback sendMessageCb_;
};

// 扮演上帝的角色，把各部件拼起来
int main()
{
  NetServer server(7);
  EchoService echo(bind(&NetServer::sendMessage, &server, _1, _2, _3));
  server.registerMessageCallback(
        bind(&EchoService::onMessage, &echo, _1, _2, _3));
  server.registerConnectionCallback(
        bind(&EchoService::onConnection, &echo, _1));
  server.run();
}
```

user code

11.5.3 对面向对象程序设计的影响

一直以来，我对面向对象都有一种厌恶感，叠床架屋，绕来绕去的，一拳拳打在棉花上，不解决实际问题。面向对象的三要素是封装、继承和多态。我认为封装是根本的，继承和多态则是可有可无的。用 class 来表示 concept，这是根本的；至于继承和多态，其耦合性太强，往往不划算。

继承和多态不仅规定了函数的名称、参数、返回类型，还规定了类的继承关系。在现代的 OO 编程语言里，借助反射和 attribute/annotation，已经大大放宽了限制。举例来说，JUnit 3.x 是用反射，找出派生类里的名字符合 `void test*()` 的函数来执行的，这里就没继承什么事，只是对函数的名称有部分限制（继承是全面限制，一字不差）。至于 JUnit 4.x 和 NUnit 2.x 则更进一步，以 annotation/attribute 来标明 test case，更没继承什么事了。

我的猜测是，当初提出面向对象的时候，closure 还没有一个通用的实现，所以它没能算作基本的抽象工具之一。现在既然 closure 已经这么方便了，或许我们应该重新审视面向对象设计，至少不要那么滥用继承。

自从找到了 boost::function+boost::bind 这对“神兵利器”，不用再考虑 class 之间的继承关系，只需要基于对象的设计（object-based），拳拳到肉，程序写起来顿时顺手了很多。

对面向对象设计模式的影响

既然虚函数能用 closure 代替，那么很多 OO 设计模式，尤其是行为模式，就失去了存在的必要。另外，既然没有继承体系，那么很多创建型模式似乎也没啥用了（比如 Factory Method 可以用 boost::function<Base* ()> 替代）。

最明显的是 Strategy，不用累赘的 Strategy 基类和 ConcreteStrategyA、ConcreteStrategyB 等派生类，一个 boost::function 成员就能解决问题。另外一个例子是 Command 模式，有了 boost::function，函数调用可以直接变成对象，似乎就没 Command 什么事了。同样的道理，Template Method 可以不必使用基类与继承，只要传入几个 boost::function 对象，在原来调用虚函数的地方换成调用 boost::function 对象就能解决问题。

在《设计模式》这本书中提到了 23 个模式，在我看来其更多的是弥补了 C++ 这种静态类型语言在动态性方面的不足。在动态语言中，由于语言内置了一等公民的类型和函数 [18]，这使得很多模式失去了存在的必要 [19]。或许它们解决了面向对象中的常见问题，不过要是我的程序里连面向对象（指继承和多态）都不用，那似乎也不用叨扰面向对象设计模式了。

或许基于 closure 的编程将作为一种新的编程范式（paradigm）而流行起来。

依赖注入与单元测试

前面的 EchoService 可算是依赖注入的例子。EchoService 需要一个什么东西来发送消息，它对这个“东西”的要求只是函数原型满足 SendMessageCallback，而并不关心数据到底发到网络上还是发到控制台。在正常使用的时候，数据应该发给网络；而在做单元测试的时候，数据应该发给某个 DataSink。

[18] “一等公民”指类型和函数可以像普通变量一样使用（赋值，传参），既可以用一个变量表示一个类型，通过该变量构造其代表的类型的对象；也可以用一个变量表示一个函数，通过该变量调用其代表的函数。

[19] http://norvig.com/design-patterns/

按照面向对象的思路，先写一个 AbstractDataSink interface，包含 sendMessage() 这个虚函数，然后派生出两个 class：NetDataSink 和 MockDataSink，前面那个干活用，后面那个单元测试用。EchoService 的构造函数应该以 AbstractDataSink* 为参数，这样就实现了所谓的接口与实现分离。

我认为这么做纯粹是多此一举，因为直接传入一个 SendMessageCallback 对象就能解决问题。在单元测试的时候，可以 boost::bind() 到 MockServer 上，或某个全局函数上，完全不用继承和虚函数，也不会影响现有的设计。

什么时候使用继承

如果是指 OO 中的 public 继承，即为了接口与实现分离，那么我只会在派生类的数目和功能完全确定的情况下使用。换句话说，不为将来的扩展考虑，这时候面向对象或许是一种不错的描述方法。一旦要考虑扩展，什么办法都没用，还不如把程序写简单点，将来好大改或重写。

如果是功能继承，那么我会考虑继承 boost::noncopyable 或 boost::enable_shared_from_this，§1.11 讲到了 enable_shared_from_this 在实现多线程安全的对象回调时的妙用。

例如，IO multiplexing 在不同的操作系统下有不同的推荐实现，最通用的 select()、POSIX 的 poll()、Linux 的 epoll()、FreeBSD 的 kqueue() 等，数目固定，功能也完全确定，不用考虑扩展。那么设计一个 NetLoop base class 加若干具体 classes 就是不错的解决办法。换句话说，用多态来代替 switch-case 以达到简化代码的目的。

基于接口的设计

这个问题来自那个经典的讨论：不会飞的企鹅（Penguin）究竟应不应该继承自鸟（Bird），如果 Bird 定义了 virtual function fly() 的话。讨论的结果是，把具体的行为提出来，作为 interface，比如 Flyable（能飞的），Runnable（能跑的），然后让企鹅实现 Runnable，麻雀实现 Flyable 和 Runnable。（其实麻雀只能双脚跳，不能跑，这里不作深究。）

进一步的讨论表明，interface 的粒度应足够小，或许包含一个 method 就够了，那么 interface 实际上退化成了给类型打的标签（tag）。在这种情况下，完全可以使用 boost::function 来代替，比如：

```
class Penguin // 企鹅能游泳，也能跑
{
 public:
  void run();
  void swim();
};

class Sparrow // 麻雀能飞，也能跑
{
 public:
  void fly();
  void run();
};

// 以 boost::function 作为接口
typedef boost::function<void()> FlyCallback;
typedef boost::function<void()> RunCallback;
typedef boost::function<void()> SwimCallback;

// 一个既用到 run，也用到 fly 的客户 class
class Foo
{
 public:
  Foo(FlyCallback flyCb, RunCallback runCb)
   : flyCb_(flyCb), runCb_(runCb)
  { }

 private:
  FlyCallback flyCb_;
  RunCallback runCb_;
};

// 一个既用到 run，也用到 swim 的客户 class
class Bar
{
 public:
  Bar(SwimCallback swimCb, RunCallback runCb)
   : swimCb_(swimCb), runCb_(runCb)
  { }

 private:
  SwimCallback swimCb_;
  RunCallback runCb_;
};

int main()
{
  Sparrow s;
  Penguin p;
  // 装配起来，Foo 要麻雀，Bar 要企鹅。
  Foo foo(bind(&Sparrow::fly, &s), bind(&Sparrow::run, &s));
  Bar bar(bind(&Penguin::swim, &p), bind(&Penguin::run, &p));
}
```

第 4 部分

附录

附录 A

谈一谈网络编程学习经验

本文谈一谈我在学习网络编程方面的一些个人经验。“网络编程”这个术语的范围很广，本文指用 Sockets API 开发基于 TCP/IP 的网络应用程序，具体定义见§A.1.5 “网络编程的各种任务角色”。

受限于本人的经历和经验，本附录的适应范围是：

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

本文可能不适合：

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

本文分两个部分：

1. 网络编程的一些“胡思乱想”，以自问自答的形式谈谈我对这一领域的认识。
2. 几本必看的书，基本上还是 W. Richard Stevents 的那几本。

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

A.1 网络编程的一些“胡思乱想”

以下大致列出我对网络编程的一些想法，前后无关联。

A.1.1 网络编程是什么

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

第一次是 2005 年在学校做一个羽毛球赛场计分系统：我用 C# 编写运行在 PC 上的软件，负责比分的显示；再用 C# 写了运行在 PDA 上的计分界面，记分员拿着 PDA 记录比分；这两部分程序通过 TCP 协议相互通信。这其实是个简单的分布式系统，体育馆有几片场地，每个场地都有一名拿 PDA 的记分员，每个场地都有两台显示比分的 PC （显示器是 42 寸平板电视，放在场地的对角，这样两边看台的观众都能看到比分)。这两台 PC 的功能不完全一样，一台只负责显示当前比分，另一台还要负责与 PDA 通信，并更新数据库里的比分信息。此外，还有一台 PC 负责周期性地从数据库读出全部 7 片场地的比分，显示在体育馆墙上的大屏幕上。这台 PC 上还运行着一个程序，负责生成比分数据的静态页面，通过 FTP 上传发布到某门户网站的体育频道。系统中还有一个录入赛程（参赛队、运动员、出场顺序等）数据库的程序，运行在数据库服务器上。算下来整个系统有十来个程序，运行在二十多台设备 (PC 和 PDA）上，还要考虑可靠性，避免 single point of failure。

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

第二次是 2010 年编写 muduo 网络库，我再次拿起了 Sockets API，写了一个基于 Reactor 模式的 C++ 网络库。写这个库的目的之一就是想让日常的网络编程从 Sockets API 的琐碎细节中解脱出来，让程序员专注于业务逻辑，把时间用在刀刃上。muduo 网络库的示例代码包含了几十个网络程序，这些示例程序都没有直接使用 Sockets API。

在此之外，无论是实习还是工作，虽然我写的程序都会通过 TCP 协议与其他程序打交道，但我没有直接使用过 Sockets API。对于 TCP 网络编程，我认为核心是处理“三个半事件”，见 §6.4.1 “TCP 网络编程本质论”。程序员的主要工作是在事件处理函数中实现业务逻辑，而不是和 Sockets API“较劲”。

这里还是没有说清楚“网络编程”是什么，请继续阅读后文 §A.1.5“网络编程的各种任务角色”。

A.1.2 学习网络编程有用吗

以上说的是比较底层的网络编程，程序代码直接面对从 TCP 或 UDP 收到的数据以及构造数据包发出去。在实际工作中，另一种常见的情况是通过各种 client library 来与服务端打交道，或者在现成的框架中填空来实现 server，或者采用更上层的通信方式。比如用 libmemcached 与 memcached 打交道，使用 libpq 来与 PostgreSQL 打交道，编写 Servlet 来响应 HTTP 请求，使用某种 RPC 与其他进程通信，等等。这些情况都会发生网络通信，但不一定算作“网络编程”。如果你的工作是前面列举的这些，学习 TCP/IP 网络编程还有用吗？

我认为还是有必要学一学，至少在 troubleshooting 的时候有用。无论如何，这些 library 或 framework 都会调用底层的 Sockets API 来实现网络功能。当你的程序遇到一个线上问题时，如果你熟悉 Sockets API，那么从 strace 不难发现程序卡在哪里，尽管可能你没有直接调用这些 Sockets API。另外，熟悉 TCP/IP 协议、会用 tcpdump 也非常有助于分析解决线上网络服务问题。

A.1.3 在什么平台上学习网络编程

对于服务端网络编程，我建议在 Linux 上学习。

如果在 10 年前，这个问题的答案或许是 FreeBSD，因为 FreeBSD“根正苗红”，在 2000 年那一次互联网浪潮中扮演了重要角色，是很多公司首选的免费服务器操作系统。2000 年那会儿 Linux 还远未成熟，连 epoll 都还没有实现。（FreeBSD 在 2001 年发布 4.1 版，加入了 kqueue，从此 C10k 不是问题。）

10 年后的今天，事情起了一些变化，Linux 成为市场份额最大的服务器操作系统[1]。在 Linux 这种大众系统上学网络编程，遇到什么问题会比较容易解决。因为用的人多，你遇到的问题别人多半也遇到过；同样因为用的人多，如果真的有什么内核 bug，很快就会得到修复，至少有 work around 的办法。如果用别的系统，可能一个问题发到论坛上半个月都不会有人理。从内核源码的风格看，FreeBSD 更干净整洁，注释到位，但是无奈它的市场份额远不如 Linux，学习 Linux 是更好的技术投资。

A.1.4 可移植性重要吗

写网络程序要不要考虑移植性？要不要跨平台？这取决于项目需要，如果贵公司做的程序要卖给其他公司，而对方可能使用 Windows、Linux、FreeBSD、Solaris、

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

AIX、HP-UX 等等操作系统，这时候当然要考虑移植性。如果编写公司内部的服务器上用的网络程序，那么大可只关注一个平台，比如 Linux。因为编写和维护可移植的网络程序的代价相当高，平台间的差异可能远比想象中大，即便是 POSIX 系统之间也有不小的差异（比如 Linux 没有 `SO_NOSIGPIPE` 选项，Linux 的 `pipe(2)` 是单向的，而 FreeBSD 是双向的），错误的返回码也大不一样。

我就不打算把 muduo 往 Windows 或其他操作系统移植。如果需要编写可移植的网络程序，我宁愿用 libevent、libuv、Java Netty 这样现成的库，把“脏活、累活”留给别人。

A.1.5 网络编程的各种任务角色

计算机网络是个 big topic，涉及很多人物和角色，既有开发人员，也有运维人员。比方说：公司内部两台机器之间 ping 不通，通常由网络运维人员解决，看看是布线有问题还是路由器设置不对；两台机器能 ping 通，但是程序连不上，经检查是本机防火墙设置有问题，通常由系统管理员解决；两台机器能连上，但是丢包很严重，发现是网卡或者交换机的网口故障，由硬件维修人员解决；两台机器的程序能连上，但是偶尔发过去的请求得不到响应，通常是程序 bug，应该由开发人员解决。

本文主要关心开发人员这一角色。下面简单列出一些我能想到的跟网络打交道的编程任务，其中前三项是面向网络本身，后面几项是在计算机网络之上构建信息系统。

1. 开发网络设备，编写防火墙、交换机、路由器的固件（firmware）。
2. 开发或移植网卡的驱动。
3. 移植或维护 TCP/IP 协议栈（特别是在嵌入式系统上）。
4. 开发或维护标准的网络协议程序，HTTP、FTP、DNS、SMTP、POP3、NFS。
5. 开发标准网络协议的“附加品”，比如 HAProxy、squid、varnish 等 Web load balancer。
6. 开发标准或非标准网络服务的客户端库，比如 ZooKeeper 客户端库、memcached 客户端库。
7. 开发与公司业务直接相关的网络服务程序，比如即时聊天软件的后台服务器、网游服务器、金融交易系统、互联网企业用的分布式海量存储、微博发帖的内部广播通知等等。
8. 客户端程序中涉及网络的部分，比如邮件客户端中与 POP3、SMTP 通信的部分，以及网游的客户端程序中与服务器通信的部分。

本文所指的“网络编程”专指第 7 项，即在 TCP/IP 协议之上开发业务软件。换句话说，不是用 Sockets API 开发 muduo 这样的网络库，而是用 libevent、muduo、Netty、gevent 这样现成的库开发业务软件，muduo 自带的十几个示例程序是业务软件的代表。

A.1.6 面向业务的网络编程的特点

与通用的网络服务器不同，面向公司业务的专用网络程序有其自身的特点。

业务逻辑比较复杂，而且时常变化 如果写一个 HTTP 服务器，在大致实现 HTTP 1.1 标准之后，程序的主体功能一般不会有太大的变化，程序员会把时间放在性能调优和 bug 修复上。而开发针对公司业务的专用程序时，功能说明书（spec）很可能不如 HTTP 1.1 标准那么细致明确。更重要的是，程序是快速演化的。以即时聊天工具的后台服务器为例，可能第一版只支持在线聊天；几个月之后发布第二版，支持离线消息；又过了几个月，第三版支持隐身聊天；随后，第四版支持上传头像；如此等等。这要求程序员能快速响应新的业务需求，公司才能保持竞争力。由于业务时常变化（假设每月一次版本升级），也会降低服务程序连续运行时间的要求。相反，我们要设计一套流程，通过轮流重启服务器来完成平滑升级（§9.2.2）。

不一定需要遵循公认的通信协议标准 比方说网游服务器就没什么协议标准，反正客户端和服务端都是本公司开发的，如果发现目前的协议设计有问题，两边一起改就行了。由于可以自己设计协议，因此我们可以绕开一些性能难点，简化程序结构。比方说，对于多线程的服务程序，如果用短连接 TCP 协议，为了优化性能通常要精心设计 accept 新连接的机制[2]，避免惊群并减少上下文切换。但是如果改用长连接，用最简单的单线程 accept 就行了。

程序结构没有定论 对于高并发大吞吐的标准网络服务，一般采用单线程事件驱动的方式开发，比如 HAProxy、lighttpd 等都是这个模式。但是对于专用的业务系统，其业务逻辑比较复杂，占用较多的 CPU 资源，这种单线程事件驱动方式不见得能发挥现在多核处理器的优势。这留给程序员比较大的自由发挥空间，做好了“横扫千军”，做烂了一败涂地。我认为目前 one loop per thread 是通用性较高的一种程序结构，能发挥多核的优势，见 §3.3 和 §6.6。

性能评判的标准不同 如果开发 httpd 这样的通用服务，必然会和开源的 Nginx、lighttpd 等高性能服务器比较，程序员要投入相当的精力去优化程序，才能在市场上

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

占有一席之地。而面向业务的专用网络程序不一定是 IO bound，也不一定有开源的实现以供对比性能，优化方向也可能不同。程序员通常更加注重功能的稳定性与开发的便捷性。性能只要一代比一代强即可。

网络编程起到支撑作用，但不处于主导地位 程序员的主要工作是实现业务逻辑，而不只是实现网络通信协议。这要求程序员深入理解业务。程序的性能瓶颈不一定在网络上，瓶颈有可能是 CPU、Disk IO、数据库等，这时优化网络方面的代码并不能提高整体性能。只有对所在的领域有深入的了解，明白各种因素的权衡（trade-off），才能做出一些有针对性的优化。现在的机器上，简单的并发长连接 echo 服务程序不用特别优化就做到十多万 qps，但是如果每个业务请求需要 1ms 密集计算，在 8 核机器上充其量能达到 8 000 qps，优化 IO 不如去优化业务计算（如果投入产出合算的话）。

A.1.7 几个术语

互联网上的很多“口水战”是由对同一术语的不同理解引起的，比如我写的《多线程服务器的适用场合》[3]，就曾经被人说是“挂羊头卖狗肉”，因为这篇文章中举的 master 例子“根本就算不上是个网络服务器。因为它的瓶颈根本就跟网络无关。”

网络服务器 “网络服务器”这个术语确实含义模糊，到底指硬件还是软件？到底是服务于网络本身的机器（交换机、路由器、防火墙、NAT），还是利用网络为其他人或程序提供服务的机器（打印服务器、文件服务器、邮件服务器）？每个人根据自己熟悉的领域，可能会有不同的解读。比方说，或许有人认为只有支持高并发、高吞吐量的才算是网络服务器。

为了避免无谓的争执，我只用“网络服务程序”或者“网络应用程序”这种含义明确的术语。“开发网络服务程序”通常不会造成误解。

客户端？服务端？ 在 TCP 网络编程中，客户端和服务端很容易区分，主动发起连接的是客户端，被动接受连接的是服务端。当然，这个“客户端”本身也可能是个后台服务程序，HTTP proxy 对 HTTP server 来说就是个客户端。

客户端编程？服务端编程？ 但是“服务端编程”和“客户端编程”就不那么好区分了。比如 Web crawler，它会主动发起大量连接，扮演的是 HTTP 客户端的角色，但似乎应该归入“服务端编程”。又比如写一个 HTTP proxy，它既会扮演服务端——

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

被动接受 Web browser 发起的连接，也会扮演客户端——主动向 HTTP server 发起连接，它究竟算服务端还是客户端？我猜大多数人会把它归入服务端编程。

那么究竟如何定义“服务端编程”？

服务端编程需要处理大量并发连接？也许是，也许不是。比如云风在一篇介绍网游服务器的博客[4]中就谈到，网游中用到的“连接服务器”需要处理大量连接，而“逻辑服务器”只有一个外部连接。那么开发这种网游“逻辑服务器”算服务端编程还是客户端编程呢？又比如机房的服务进程监控软件，并发数跟机器数成正比，至多也就是两三千的并发连接。(再大规模就超出本书的范围了。)

我认为，“服务端网络编程”指的是编写没有用户界面的长期运行的网络程序，程序默默地运行在一台服务器上，通过网络与其他程序打交道，而不必和人打交道。与之对应的是客户端网络程序，要么是短时间运行，比如 wget；要么是有用户界面(无论是字符界面还是图形界面)。本文主要谈服务端网络编程。

A.1.8 7×24 重要吗，内存碎片可怕吗

一谈到服务端网络编程，有人立刻会提出 7×24 运行的要求。对于某些网络设备而言，这是合理的需求，比如交换机、路由器。对于开发商业系统，我认为要求程序 7×24 运行通常是系统设计上考虑不周。具体见本书 §9.2 “分布式系统的可靠性浅说”。重要的不是 7×24，而是在程序不必做到 7×24 的情况下也能达到足够高的可用性。一个考虑周到的系统应该允许每个进程都能随时重启，这样才能在廉价的服务器硬件上做到高可用性。

既然不要求 7×24，那么也不必害怕内存碎片[5][6]，理由如下：

- 64-bit 系统的地址空间足够大，不会出现没有足够的连续空间这种情况。有没有谁能够故意制造内存碎片（不是内存泄漏）使得服务程序失去响应？
- 现在的内存分配器（`malloc` 及其第三方实现）今非昔比，除了 memcached 这种纯以内存为卖点的程序需要自己设计分配器之外，其他网络程序大可使用系统自带的 `malloc` 或者某个第三方实现。重新发明 memory pool 似乎已经不流行了（§12.2.8）。

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

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

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

- Linux Kernel 也大量用到了动态内存分配。既然操作系统内核都不怕动态分配内存造成碎片，应用程序为什么要害怕？应用程序的可靠性只要不低于硬件和操作系统的可靠性就行。普通 PC 服务器的年故障率约为 3% ~ 5%，算一算你的服务程序一年要被意外重启多少次。
- 内存碎片如何度量？有没有什么工具能为当前进程的内存碎片状况评个分？如果不能比较两种方案的内存碎片程度，谈何优化？

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

A.1.9 协议设计是网络编程的核心

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

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

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

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

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

比连接的建立与断开更重要的是设计消息协议。消息格式很好办，XML、JSON、Protobuf 都是很好的选择；难的是消息内容。一个消息应该包含哪些内容？多个程序

相互通信如何避免 race condition？（见 p. 348 举的例子）外部事件发生时，网络消息应该发 snapshot 还是 delta？新增功能时，各个组件如何平滑升级？

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

A.1.10 网络编程的三个层次

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

我认为网络编程也可以分为三个层次：

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

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

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

1. 有可能出现 TCP 自连接（self-connection）[8]，程序应该有所准备。
2. Linux 的内核会有 bug，比如某种 TCP 拥塞控制算法曾经出现 TCP window clamping（窗口箝位）bug，导致吞吐量暴跌，可以选用其他拥塞控制算法来绕开（work around）这个问题。

这些“阴暗角落”在 manpage 里没有描述，要通过其他渠道了解。

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

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

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

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

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

如果有时间，我打算自己写一个 Mini/Tiny/Toy/Trivial/Yet-Another TCP/IP。我准备换一个思路，用 TUN/TAP 设备在用户态实现一个能与本机点对点通信的 TCP/IP 协议栈（见本书附录 D），这样那三个接口函数就表现为我最熟悉的文件读写。在用户态实现的好处是便于调试，协议栈做成静态库，与应用程序链接到一起(库的接口不必是标准的 Sockets API)。写完这一版协议栈，还可以继续发挥，用 FTDI 的 USB-SPI 接口芯片连接 ENC28J60 适配器，做一个真正独立于操作系统的 TCP/IP stack。如果只实现最基本的 IP、ICMP Echo、TCP，代码应能控制在 3000 行以内；也可以实现 UDP，如果应用程序需要用到 DNS 的话。

A.1.11 最主要的三个例子

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

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

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

proxy 的作用：连接的管理更加复杂：既要被动接受连接，也要主动发起连接；既要主动关闭连接，也要被动关闭连接。还要考虑两边速度不匹配（§7.13）。

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

A.1.12 学习 Sockets API 的利器：IPython

我在编写 muduo 网络库的时候，写了一个命令行交互式的调试工具[9]，方便试验各个 Sockets API 的返回时机和返回值。后来发现其实可以用 IPython 达到相同的效果，不必自己编程。用交互式工具很快就能摸清各种 IO 事件的发生条件，比反复编译 C 代码高效得多。比方说想简单试验一下 TCP 服务器和 epoll，可以这么写：

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

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

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

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

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

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

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

In [15]: client.recv(1024)      # 再去读数据，立刻返回结果
Out[15]: 'Bye!\n'

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

同时在另一个命令行窗口用 nc 发送数据：

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

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

在编写 muduo 的时候，我一般会开四个命令行窗口，其一看 log，其二看 strace，其三用 netcat/tempest/ipython 充作通信对方，其四看 tcpdump。各个工具的输出相互验证，很快就摸清了门道。muduo 是一个基于 Reactor 模式的 Linux C++ 网络库，采用非阻塞 IO，支持高并发和多线程，核心代码量不大（4000 多行），示例丰富，可供网络编程的学习者参考。

A.1.13 TCP 的可靠性有多高

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

我在 §7.5 “一种自动反射消息类型的 Google Protobuf 网络传输方案”中设计了带 check sum 的消息格式，很多人表示不理解，认为是多余的。IP header 中有 check sum，TCP header 也有 check sum，链路层以太网还有 CRC32 校验，那么为什么还需要在应用层做校验？什么情况下 TCP 传送的数据会出错？

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

图 A-1 中 client 向 server 发了一个 TCP segment，这个 segment 先被封装成一个 IP packet，再被封装成 ethernet frame，发送到路由器（图 A-1 中的消息 a）。router 收到 ethernet frame b，转发到另一个网段（消息 c），最后 server 收到 d，通知应用程序。以太网 CRC 能保证 a 和 b 相同，c 和 d 相同；TCP header checksum 的强度不足以保证收发 payload 的内容一样。另外，如果把 router 换成 NAT，那么 NAT 自己会构造消息 c（替换掉源地址），这时候 a 和 d 的 payload 不能用 TCP header checksum 校验。

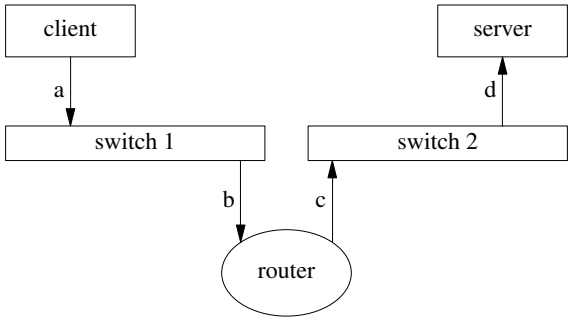


图 A-1

路由器可能出现硬件故障，比方说它的内存故障（或偶然错误）导致收发 IP 报文出现多 bit 的反转或双字节交换，这个反转如果发生在 payload 区，那么无法用链路层、网络层、传输层的 check sum 查出来，只能通过应用层的 check sum 来检测。这个现象在开发的时候不会遇到，因为开发用的几台机器很可能都连到同一个交换机，ethernet CRC 能防止错误。开发和测试的时候数据量不大，错误很难发生。之后大规模部署到生产环境，网络环境复杂，这时候出个错就让人措手不及。有一篇论文《When the CRC and TCP checksum disagree》分析了这个问题。另外《The Limitations of the Ethernet CRC and TCP/IP checksums for error detection》[10] 也值得一读。

这个情况真的会发生吗？会的，Amazon S3 在 2008 年 7 月就遇到过 [11]，单 bit 反转导致了一次严重线上事故，所以他们吸取教训加了 check sum。另外见 Google 工程师的经验分享 [12]。

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

A.2 三本必看的书

谈到 Unix 编程和网络编程，W. Richard Stevens 是个绕不开的人物，他生前写了 6 本书，即 [APUE]、两卷《UNIX 网络编程》、三卷《TCP/IP 详解》。其中四本与

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

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

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

网络编程直接相关。[UNPv2] 其实跟网络编程关系不大，是 [APUE] 在多线程和进程间通信（IPC）方面的补充。很多人把《TCP/IP 详解》一二三卷作为整体推荐，其实这三本书的用处不同，应该区别对待。

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

第一本：《TCP/IP Illustrated, Vol. 1: The Protocols》（中文名《TCP/IP 详解》），以下简称 TCPv1。

TCPv1 是一本奇书。这本书迄今至少被三百多篇学术论文引用过 [13]。一本学术专著被论文引用算不上出奇，难得的是一本写给程序员看的技术书能被学术论文引用几百次，我不知道还有哪本技术书能做到这一点。

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

TCP 作为一个可靠的传输层协议，其核心有三点：

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

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

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

TCPv1 唯一的不足是它出版得太早了，1993 年至今网络技术发展了几代。链路层方面，当年主流的 10Mbit 网卡和集线器早已经被淘汰；100Mbit 以太网也没什么企业在用了，交换机（switch）也已经全面取代了集线器（hub）；服务器机房以 1Gbit

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

网络为主，有些场合甚至用上了 10Gbit 以太网。另外，无线网的普及也让 TCP flow control 面临新挑战；原来设计 TCP 的时候，人们认为丢包通常是拥塞造成的，这时应该放慢发送速度，减轻拥塞；而在无线网中，丢包可能是信号太弱造成的，这时反而应该快速重试，以保证性能。网络层方面变化不大，IPv6 “雷声大、雨点小”。传输层方面，由于链路层带宽大增，TCP window scale option 被普遍使用，另外 TCP timestamps option 和 TCP selective ack option 也很常用。由于这些因素，在现在的 Linux 机器上运行 tcpdump 观察 TCP 协议，程序输出会与原书有些不同。

一个好消息：TCPv1 已于 2011 年 10 月推出第 2 版，经典能否重现？

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

UNP 是 Sockets API 的权威指南，但是网络编程远不是使用那十几个 Sockets API 那么简单，作者 W. Richard Stevens 深刻地认识到了这一点，他在 UNP 第 2 版的前言中写道：[14]

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

搞网络编程，一定要熟悉 TCP/IP 协议及其外在表现（比如打开和关闭 Nagle 算法对收发包延时的影响），不然出点意料之外的情况就摸不着头脑了。我不知道为什么 UNP 第 3 版在前言中去掉了这段至关重要的话。

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

UNP 很详细，面面俱到，UDP、TCP、IPv4、IPv6 都讲到了。要说有什么缺点的话，就是太详细了，重点不够突出。我十分赞同孟岩说的：[15]

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

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

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

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

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

另外，UNP 举的两个例子（菜谱）太简单，daytime 和 echo 一个是短连接协议，一个是长连接无格式协议，不足以覆盖基本的网络开发场景（比如 TCP 封包与拆包、

多连接之间交换数据)。我估计 W. Richard Stevens 原打算在 UNP 第三卷中讲解一些实际的例子，只可惜他英年早逝，我等无福阅读。

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

TCPv1 和 UNP 应该先看哪本？见仁见智吧。我自己是先看的 TCPv1，花了大约两个月时间，然后再读 UNP 和 APUE。

第三本：《Effective TCP/IP Programming》

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

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

其他值得一看的书

以下两本都不易读，需要相当的基础。

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

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

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

对于主要使用 TCP 的程序员，我认为 TCPv2 的一大半内容可以跳过不看，比如路由表、IGMP 等等（开发网络设备的人可能更关心这些内容)。在工作中大可以把 IP 视为 host-to-host 的协议，把“IP packet 如何送达对方机器”的细节视为黑盒子，

这不会影响对 TCP 的理解和运用，因为网络协议是分层的。这样精简下来，需要看的只有三四百页，四五千行代码，大大减轻了阅读的负担。

这本书直接呈现高质量的工业级操作系统源码，读起来有难度，读懂它甚至要有“不求甚解的能力”。其一，代码只能看，不能上机运行，也不能改动试验。其二，与操作系统的其他部分紧密关联。比如 TCP/IP stack 下接网卡驱动、软中断；上承 inode 转发来的系统调用操作；中间还要与平级的进程文件描述符管理子系统打交道。如果要把每一部分都弄清楚，把持不住就会迷失主题。其三，一些历史包袱让代码变得复杂晦涩。比如 BSD 在 20 世纪 80 年代初需要在只有 4MiB 内存的 VAX 小型机上实现 TCP/IP，内存方面捉襟见肘，这才发明了 `mbuf` 结构，代码也增加了不少偶发复杂度（buffer 不连续的处理）。

读这套 TCP/IP 书切忌胶柱鼓瑟，这套书以 4.4BSD 为讲解对象，其描述的行为（特别是与 timer 相关的行为）与现在的 Linux TCP/IP 有不小的出入，用书本上的知识直接套用到生产环境的 Linux 系统可能会造成不小的误解和困扰。（《TCP/IP 详解（第 3 卷）》不重要，可以成套买来收藏，不读亦可。）

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

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

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

附录 B
从《C++ Primer（第 4 版）》入手学习 C++

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

B.1 为什么要学习 C++

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B.2 学习 C++ 只需要读一本大部头

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

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

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

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

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

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

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

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

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

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

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

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

《C++ Primer （第 4 版）》讲什么？适合谁读？

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

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

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

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

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

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

[23] 侯捷《C++ Primer 4/e 译序》。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

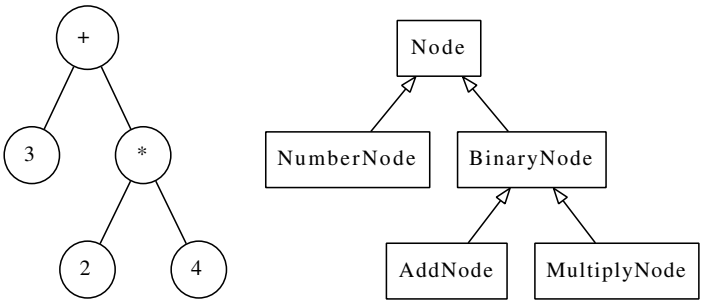


图 B-1

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

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

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

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

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

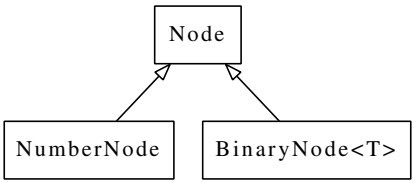


图 B-2

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

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

B.3 继续前进

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

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

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

[38] 从代码风格上往往能判断项目成型的时代。

[39] Scott Meyers 著，侯捷译，电子工业出版社出版。

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

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

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

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

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

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

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

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

[42] “分配内存”包括在堆（heap）上创建对象。

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

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

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

绍 TR1 的这几个主要部件[46]。

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

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

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

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

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

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

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

[47] 最好再学一点基础的离散数学。

[48] Matthew Austern 著，侯捷译，中国电力出版社。

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

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

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

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

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

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

有的条款则需要相当的设计与编码经验才能解其中三昧：

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

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

B.4 评注版使用说明

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

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

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

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

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

网上资源

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

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

陈硕
2012 年 5 月
中国 · 香港

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

附录 C
关于 Boost 的看法

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

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

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

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

[1] 这篇文章写于 2010 年 8 月。

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

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

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

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

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

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

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

（后略）

附录 D
关于 TCP 并发连接的几个思考题与试验

前几天我在新浪微博上出了两道有关 TCP 的思考题，引发了一场讨论[1]。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

faketcp 的程序实现

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

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

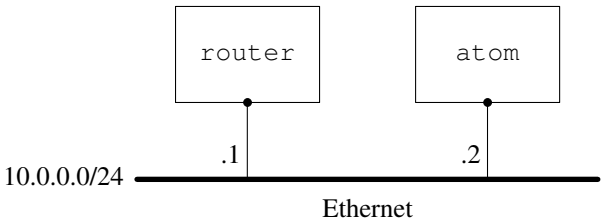


图 D-1

我在附录 A 中曾提到“可以用 TUN/TAP 设备在用户态实现一个能与本机点对点通信的 TCP/IP 协议栈”，这次的试验正好可以用上这个办法。试验的网络配置如图 D-2 所示。

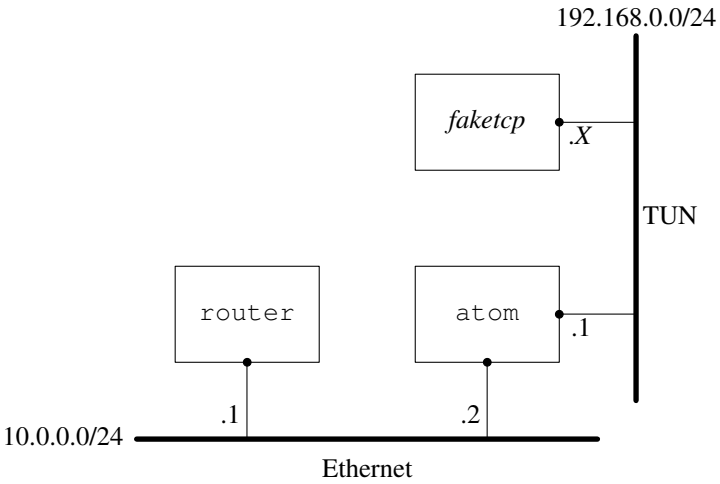


图 D-2

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

程序分成几步来实现。

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

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

运行方法，打开 3 个命令行窗口：

1. 在第 1 个窗口运行 `sudo ./icmpecho`，程序显示

```
allocted tunnel interface tun0
```

2. 在第 2 个窗口运行

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

3. 在第 3 个窗口运行

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

注意到每个 192.168.0.X 的 IP 都能 ping 通。

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

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

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

注意到向其中任意一个 IP 发起的 TCP 连接都被拒接了。

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

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

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

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

这一步已经解决了前面的问题 2，扮演任意 TCP 服务端。

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

这一步的运行方法与前面不同，打开 4 个命令行窗口：

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

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

第一道题的答案：

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

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

即便客户端使用 NAT，也不影响这个理论上限。(为什么?)

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

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

参考文献

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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