

Linux 多线程 服务端编程

使用 muduo C++ 网络库

陈硕
著

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



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
www.phei.com.cn

Linux 多线程服务端编程： 使用 muduo C++ 网络库

陈硕著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

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。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Linux 多线程服务端编程：使用 muduo C++网络库 / 陈硕著. —北京：电子工业出版社，2013.1
ISBN 978-7-121-19282-1

I. ①L… II. ①陈… III. ①Linux 操作系统—程序设计 IV. ①TP316.89

中国版本图书馆 CIP 数据核字（2012）第 304000 号

策划编辑：张春雨

责任编辑：李云静

印 刷：北京丰源印刷厂

装 订：三河市鹏成印业有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：38.5 字数：801 千字

印 次：2013 年 1 月第 1 次印刷

印 数：3000 册 定价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

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++”指的是不是 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周期管理、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。

```

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

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

第 1 部分	C++ 多线程系统编程	1
第 1 章	线程安全的对象生命期管理	3
第 2 章	线程同步精要	31
第 3 章	多线程服务器的适用场合与常用编程模型	59
第 4 章	C++ 多线程系统编程精要	83
第 5 章	高效的多线程日志	107
第 2 部分	muduo 网络库	123
第 6 章	muduo 网络库简介	125
第 7 章	muduo 编程示例	177
第 8 章	muduo 网络库设计与实现	277
第 3 部分	工程实践经验谈	337
第 9 章	分布式系统工程实践	339
第 10 章	C++ 编译链接模型精要	391
第 11 章	反思 C++ 面向对象与虚函数	429
第 12 章	C++ 经验谈	501
第 4 部分	附录	559
附录 A	谈一谈网络编程学习经验	561
附录 B	从《C++ Primer (第 4 版)》入手学习 C++	579
附录 C	关于 Boost 的看法	591
附录 D	关于 TCP 并发连接的几个思考题与试验	593
参考文献		599

目录

第 1 部分 C++ 多线程系统编程	1
第 1 章 线程安全的对象生命期管理	3
1.1 当析构函数遇到多线程	3
1.1.1 线程安全的定义	4
1.1.2 MutexLock 与 MutexLockGuard	4
1.1.3 一个线程安全的 Counter 示例	4
1.2 对象的创建很简单	5
1.3 销毁太难	7
1.3.1 mutex 不是办法	7
1.3.2 作为数据成员的 mutex 不能保护析构	8
1.4 线程安全的 Observer 有多难	8
1.5 原始指针有何不妥	11
1.6 神器 shared_ptr/weak_ptr	13
1.7 插曲：系统地避免各种指针错误	14
1.8 应用到 Observer 上	16
1.9 再论 shared_ptr 的线程安全	17
1.10 shared_ptr 技术与陷阱	19
1.11 对象池	21
1.11.1 enable_shared_from_this	23
1.11.2 弱回调	24
1.12 替代方案	26
1.13 心得与小结	26
1.14 Observer 之谬	28
第 2 章 线程同步精要	31
2.1 互斥器 (mutex)	32

2.1.1	只使用非递归的 mutex	33
2.1.2	死锁	35
2.2	条件变量 (condition variable)	40
2.3	不要用读写锁和信号量	43
2.4	封装 MutexLock、MutexLockGuard、Condition	44
2.5	线程安全的 Singleton 实现	48
2.6	sleep(3) 不是同步原语	50
2.7	归纳与总结	51
2.8	借 shared_ptr 实现 copy-on-write	52
第 3 章	多线程服务器的适用场合与常用编程模型	59
3.1	进程与线程	59
3.2	单线程服务器的常用编程模型	61
3.3	多线程服务器的常用编程模型	62
3.3.1	one loop per thread	62
3.3.2	线程池	63
3.3.3	推荐模式	64
3.4	进程间通信只用 TCP	65
3.5	多线程服务器的适用场合	67
3.5.1	必须用单线程的场合	69
3.5.2	单线程程序的优缺点	70
3.5.3	适用多线程程序的场景	71
3.6	“多线程服务器的适用场合”例释与答疑	74
第 4 章	C++ 多线程系统编程精要	83
4.1	基本线程原语的选用	84
4.2	C/C++ 系统库的线程安全性	85
4.3	Linux 上的线程标识	89
4.4	线程的创建与销毁的守则	91
4.4.1	pthread_cancel 与 C++	94
4.4.2	exit(3) 在 C++ 中不是线程安全的	94
4.5	善用 __thread 关键字	96
4.6	多线程与 IO	98

4.7	用 RAII 包装文件描述符	99
4.8	RAII 与 fork()	101
4.9	多线程与 fork()	102
4.10	多线程与 signal	103
4.11	Linux 新增系统调用的启示	105
第 5 章	高效的多线程日志	107
5.1	功能需求	109
5.2	性能需求	112
5.3	多线程异步日志	114
5.4	其他方案	120
第 2 部分	muduo 网络库	123
第 6 章	muduo 网络库简介	125
6.1	由来	125
6.2	安装	127
6.3	目录结构	129
6.3.1	代码结构	131
6.3.2	例子	134
6.3.3	线程模型	135
6.4	使用教程	136
6.4.1	TCP 网络编程本质论	136
6.4.2	echo 服务的实现	138
6.4.3	七步实现 finger 服务	140
6.5	性能评测	144
6.5.1	muduo 与 Boost.Asio、libevent2 的吞吐量对比	145
6.5.2	击鼓传花：对比 muduo 与 libevent2 的事件处理效率	148
6.5.3	muduo 与 Nginx 的吞吐量对比	153
6.5.4	muduo 与 ZeroMQ 的延迟对比	156
6.6	详解 muduo 多线程模型	157
6.6.1	数独求解服务器	157
6.6.2	常见的并发网络服务程序设计方案	160

第 7 章 muduo 编程示例	177
7.1 五个简单 TCP 示例	178
7.2 文件传输	185
7.3 Boost.Asio 的聊天服务器	194
7.3.1 TCP 分包	194
7.3.2 消息格式	195
7.3.3 编解码器 LengthHeaderCode	197
7.3.4 服务端的实现	198
7.3.5 客户端的实现	200
7.4 muduo Buffer 类的设计与使用	204
7.4.1 muduo 的 IO 模型	204
7.4.2 为什么 non-blocking 网络编程中应用层 buffer 是必需的	205
7.4.3 Buffer 的功能需求	207
7.4.4 Buffer 的数据结构	209
7.4.5 Buffer 的操作	211
7.4.6 其他设计方案	217
7.4.7 性能是不是问题	218
7.5 一种自动反射消息类型的 Google Protobuf 网络传输方案	220
7.5.1 网络编程中使用 Protobuf 的两个先决条件	220
7.5.2 根据 type name 反射自动创建 Message 对象	221
7.5.3 Protobuf 传输格式	226
7.6 在 muduo 中实现 Protobuf 编解码器与消息分发器	228
7.6.1 什么是编解码器 (codec)	229
7.6.2 实现 ProtobufCodec	232
7.6.3 消息分发器 (dispatcher) 有什么用	232
7.6.4 ProtobufCodec 与 ProtobufDispatcher 的综合运用	233
7.6.5 ProtobufDispatcher 的两种实现	234
7.6.6 ProtobufCodec 和 ProtobufDispatcher 有何意义	236
7.7 限制服务器的最大并发连接数	237
7.7.1 为什么要限制并发连接数	237
7.7.2 在 muduo 中限制并发连接数	238

7.8	定时器	240
7.8.1	程序中的时间	240
7.8.2	Linux 时间函数	241
7.8.3	muduo 的定时器接口	242
7.8.4	Boost.Asio Timer 示例	243
7.8.5	Java Netty 示例	245
7.9	测量两台机器的网络延迟和时间差	248
7.10	用 timing wheel 踢掉空闲连接	250
7.10.1	timing wheel 原理	251
7.10.2	代码实现与改进	254
7.11	简单的消息广播服务	257
7.12	“串并转换”连接服务器及其自动化测试	260
7.13	socks4a 代理服务器	264
7.13.1	TCP 中继器	264
7.13.2	socks4a 代理服务器	267
7.13.3	$N:1$ 与 $1:N$ 连接转发	267
7.14	短址服务	267
7.15	与其他库集成	268
7.15.1	UDNS	270
7.15.2	c-ares DNS	272
7.15.3	curl	273
7.15.4	更多	275
第 8 章	muduo 网络库设计与实现	277
8.0	什么都不做的 EventLoop	277
8.1	Reactor 的关键结构	280
8.1.1	Channel class	280
8.1.2	Poller class	283
8.1.3	EventLoop 的改动	287
8.2	TimerQueue 定时器	290
8.2.1	TimerQueue class	290
8.2.2	EventLoop 的改动	292

8.3	EventLoop::runInLoop() 函数	293
8.3.1	提高 TimerQueue 的线程安全性	296
8.3.2	EventLoopThread class	297
8.4	实现 TCP 网络库	299
8.5	TcpServer 接受新连接	303
8.5.1	TcpServer class	304
8.5.2	TcpConnection class	305
8.6	TcpConnection 断开连接	308
8.7	Buffer 读取数据	313
8.7.1	TcpConnection 使用 Buffer 作为输入缓冲	314
8.7.2	Buffer::readFd()	315
8.8	TcpConnection 发送数据	316
8.9	完善 TcpConnection	320
8.9.1	SIGPIPE	321
8.9.2	TCP No Delay 和 TCP keepalive	321
8.9.3	WriteCompleteCallback 和 HighWaterMarkCallback	322
8.10	多线程 TcpServer	324
8.11	Connector	327
8.12	TcpClient	332
8.13	epoll	333
8.14	测试程序一览	336
第 3 部分 工程实践经验谈		337
第 9 章 分布式系统工程实践		339
9.1	我们在技术浪潮中的位置	341
9.1.1	分布式系统的本质困难	343
9.1.2	分布式系统是个险恶的问题	344
9.2	分布式系统的可靠性浅说	349
9.2.1	分布式系统的软件不要求 7 × 24 可靠	352
9.2.2	“能随时重启进程”作为程序设计目标	354
9.3	分布式系统中心跳协议的设计	356

9.4 分布式系统中的进程标识	360
9.4.1 错误做法	361
9.4.2 正确做法	362
9.4.3 TCP 协议的启示	363
9.5 构建易于维护的分布式程序	364
9.6 为系统演化做准备	367
9.6.1 可扩展的消息格式	368
9.6.2 反面教材：ICE 的消息打包格式	369
9.7 分布式程序的自动化回归测试	370
9.7.1 单元测试的能与不能	370
9.7.2 分布式系统测试的要点	373
9.7.3 分布式系统的抽象观点	374
9.7.4 一种自动化的回归测试方案	375
9.7.5 其他用处	379
9.8 分布式系统部署、监控与进程管理的几重境界	380
9.8.1 境界 1：全手工操作	382
9.8.2 境界 2：使用零散的自动化脚本和第三方组件	383
9.8.3 境界 3：自制机群管理系统，集中化配置	386
9.8.4 境界 4：机群管理与 naming service 结合	389
第 10 章 C++ 编译链接模型精要	391
10.1 C 语言的编译模型及其成因	394
10.1.1 为什么 C 语言需要预处理	395
10.1.2 C 语言的编译模型	398
10.2 C++ 的编译模型	399
10.2.1 单遍编译	399
10.2.2 前向声明	402
10.3 C++ 链接（linking）	404
10.3.1 函数重载	406
10.3.2 inline 函数	407
10.3.3 模板	409
10.3.4 虚函数	414

10.4	工程项目中头文件的使用规则	415
10.4.1	头文件的害处	416
10.4.2	头文件的使用规则	417
10.5	工程项目中库文件的组织原则	418
10.5.1	动态库是有害的	423
10.5.2	静态库也好不到哪儿去	424
10.5.3	源码编译是王道	428
第 11 章	反思 C++ 面向对象与虚函数	429
11.1	朴实的 C++ 设计	429
11.2	程序库的二进制兼容性	431
11.2.1	什么是二进制兼容性	432
11.2.2	有哪些情况会破坏库的 ABI	433
11.2.3	哪些做法多半是安全的	435
11.2.4	反面教材: COM	435
11.2.5	解决办法	436
11.3	避免使用虚函数作为库的接口	436
11.3.1	C++ 程序库的作者的生存环境	437
11.3.2	虚函数作为库的接口的两大用途	438
11.3.3	虚函数作为接口的弊端	439
11.3.4	假如 Linux 系统调用以 COM 接口方式实现	442
11.3.5	Java 是如何应对的	443
11.4	动态库接口的推荐做法	443
11.5	以 boost::function 和 boost::bind 取代虚函数	447
11.5.1	基本用途	450
11.5.2	对程序库的影响	451
11.5.3	对面向对象程序设计的影响	453
11.6	iostream 的用途与局限	457
11.6.1	stdio 格式化输入输出的缺点	457
11.6.2	iostream 的设计初衷	461
11.6.3	iostream 与标准库其他组件的交互	463

11.6.4	iostream 在使用方面的缺点	464
11.6.5	iostream 在设计方面的缺点	468
11.6.6	一个 300 行的 memory buffer output stream	476
11.6.7	现实的 C++ 程序如何做文件 IO	480
11.7	值语义与数据抽象	482
11.7.1	什么是值语义	482
11.7.2	值语义与生命期	483
11.7.3	值语义与标准库	488
11.7.4	值语义与 C++ 语言	488
11.7.5	什么是数据抽象	490
11.7.6	数据抽象所需的语言设施	493
11.7.7	数据抽象的例子	495
第 12 章	C++ 经验谈	501
12.1	用异或来交换变量是错误的	501
12.1.1	编译器会分别生成什么代码	503
12.1.2	为什么短的代码不一定快	505
12.2	不要重载全局 ::operator new()	507
12.2.1	内存管理的基本要求	507
12.2.2	重载 ::operator new() 的理由	508
12.2.3	::operator new() 的两种重载方式	508
12.2.4	现实的开发环境	509
12.2.5	重载 ::operator new() 的困境	510
12.2.6	解决办法: 替换 malloc()	512
12.2.7	为单独的 class 重载 ::operator new() 有问题吗	513
12.2.8	有必要自行定制内存分配器吗	513
12.3	带符号整数的除法与余数	514
12.3.1	语言标准怎么说	515
12.3.2	C/C++ 编译器的表现	516
12.3.3	其他语言的规定	516
12.3.4	脚本语言解释器代码	517
12.3.5	硬件实现	521

12.4 在单元测试中 mock 系统调用	522
12.4.1 系统函数的依赖注入	522
12.4.2 链接期垫片 (link seam)	524
12.5 慎用匿名 namespace	526
12.5.1 C 语言的 static 关键字的两种用法	526
12.5.2 C++ 语言的 static 关键字的四种用法	526
12.5.3 匿名 namespace 的不利之处	527
12.5.4 替代办法	529
12.6 采用有利于版本管理的代码格式	529
12.6.1 对 diff 友好的代码格式	530
12.6.2 对 grep 友好的代码风格	537
12.6.3 一切为了效率	538
12.7 再探 std::string	539
12.7.1 直接拷贝 (eager copy)	540
12.7.2 写时复制 (copy-on-write)	542
12.7.3 短字符串优化 (SSO)	543
12.8 用 STL algorithm 轻松解决几道算法面试题	546
12.8.1 用 next_permutation() 生成排列与组合	546
12.8.2 用 unique() 去除连续重复空白	548
12.8.3 用 {make, push, pop}_heap() 实现多路归并	549
12.8.4 用 partition() 实现“重排数组, 让奇数位于偶数前面”	553
12.8.5 用 lower_bound() 查找 IP 地址所属的城市	554
 第 4 部分 附录	 559
附录 A 谈一谈网络编程学习经验	561
附录 B 从《C++ Primer (第 4 版)》入手学习 C++	579
附录 C 关于 Boost 的看法	591
附录 D 关于 TCP 并发连接的几个思考题与试验	593
参考文献	599

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_ptr¹ 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 L24，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；
- 也不要再在构造函数中把 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。）

<pre>Foo::~~Foo() { MutexLockGuard lock(mutex_); // free internal state (1) }</pre>	<pre>void Foo::update() { MutexLockGuard lock(mutex_); // (2) // make use of internal state }</pre>

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()`。

<pre>extern Foo* x; // visible by all threads</pre>	
<pre>// thread A delete x; x = NULL; // helpless</pre>	<pre>// thread B if (x) { x->update(); }</pre>

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（引用也一样看得出来）。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）。

```

1 class Observer // : boost::noncopyable
2 {
3     public:
4         virtual ~Observer();
5         virtual void update() = 0;
6         // ...
7 };
8
9 class Observable // : boost::noncopyable
10 {
11     public:
12         void register_(Observer* x);
13         void unregister(Observer* x);
14
15         void notifyObservers() {
16             for (Observer* x : observers_) { // 这行是 C++11
17                 x->update(); // (3)
18             }
19         }
20     private:
21         std::vector<Observer*> observers_;
22 };

```

当 Observable 通知每一个 Observer 时 (L17)，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。

```

23 class Observer
24 {
25     // 同前
26     void observe(Observable* s) {
27         s->register_(this);
28         subject_ = s;
29     }
30
31     virtual ~Observer() {
32         subject_->unregister(this);
33     }
34
35     Observable* subject_;
36 };

```

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：L32 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 L32 之前，还没有来得及 unregister 本对象。
2. 线程 B 执行到 L17，x 正好指向是 L32 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 `L32` 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。

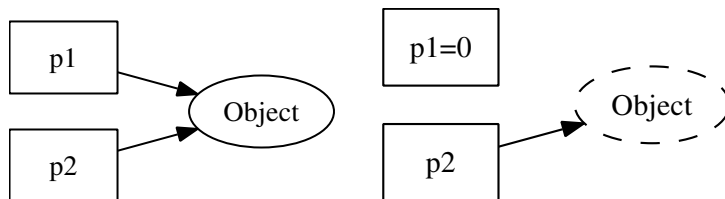


图 1-1

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。）

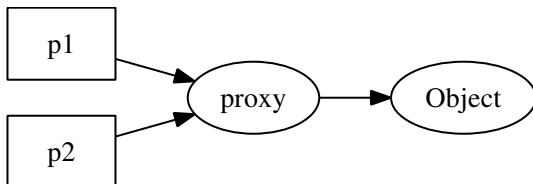


图 1-2

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。

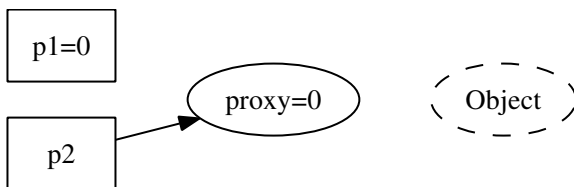


图 1-3

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）。

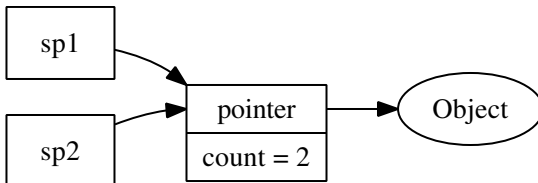


图 1-4

2. sp1 析构了，引用计数的值减为 1（见图 1-5）。

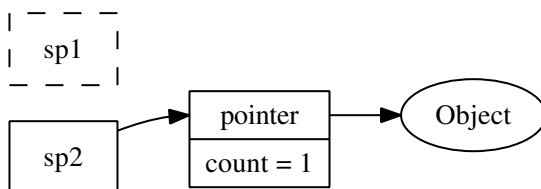


图 1-5

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）。

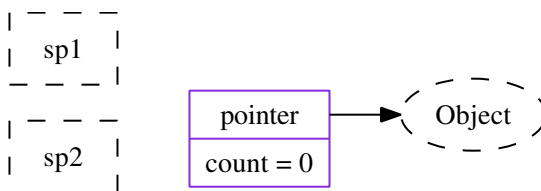


图 1-6

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：

```

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  }

```

就这么简单。前文代码 (3) 处 (p. 10 的 L17) 的竞态条件已经弥补了。思考：如果把 L48 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 `L62` 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 L48 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 StockFactory¹³。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，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` 是值语义，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 章

线程同步精要

并发编程有两种基本模型，一种是 message passing，另一种是 shared memory。在分布式系统中，运行在多台机器上的多个进程的并行编程只有一种实用模型：message passing¹。在单机上，我们也可以照搬 message passing 作为多个进程的并发模型。这样整个分布式系统的架构的一致性很强，扩容（scale out）起来也较容易。在多线程编程中，message passing 更容易保证程序的正确性，有的语言只提供这一种模型。不过在用 C/C++ 编写多线程程序时，我们仍然需要了解底层的 shared memory 模型下的同步原语，以备不时之需。本章不是多线程教程²，而是个人经验总结，分享一些 C++ 多线程编程的经验。本章多次引用《Real-World Concurrency》一文³的观点，这篇文章的地址是 <http://queue.acm.org/detail.cfm?id=1454462>，后文简称 [RWC]。

线程同步的四项原则，按重要性排列：

1. 首要原则是尽量最低限度地共享对象，减少需要同步的场合。一个对象能不暴露给别的线程就不要暴露；如果要暴露，优先考虑 immutable 对象；实在不行才暴露可修改的对象，并用同步措施来充分保护它。
2. 其次是使用高级的并发编程构件，如 TaskQueue、Producer-Consumer Queue、CountDownLatch 等等。
3. 最后不得已必须使用底层同步原语（primitives）时，只用非递归的互斥器和条件变量，慎用读写锁，不要用信号量。
4. 除了使用 atomic 整数之外，不自己编写 lock-free 代码⁴，也不要“内核级”同步原语^{4 5}。不凭空猜测“哪种做法性能会更好”，比如 spin lock vs. mutex。

前面两条很容易理解，这里着重讲一下第 3 条：底层同步原语的使用。

¹ Parallel Virtual Machine 似乎已经退出主流 HPC 了。

² 教程可参考：<https://computing.lln.gov/tutorials/pthreads>。

³ [RWC]: Use wait- and lock-free structures only if you absolutely must.

⁴ <http://www.thinkingparallel.com/2007/02/19/please-dont-rely-on-memory-barriers-for-synchronization/>

⁵ <http://zaitcev.livejournal.com/144041.html> <http://www.kernel.org/doc/Documentation/volatile-considered-harmful.txt>

2.1 互斥器 (mutex)

互斥器 (mutex)⁶ 恐怕是使用得最多的同步原语, 粗略地说, 它保护了临界区, 任何一个时刻最多只能有一个线程在此 mutex 划出的临界区内活动。单独使用 mutex 时, 我们主要为了保护共享数据。我个人的原则是:

- 用 RAII 手法封装 mutex 的创建、销毁、加锁、解锁这四个操作。用 RAII 封装这几个操作是通行的做法, 这几乎是 C++ 的标准实践, 后面我会给出具体的代码示例, 相信大家都已经写过或用过类似的代码了。Java 里的 synchronized 语句和 C# 的 using 语句也有类似的效果, 即保证锁的生效期间等于一个作用域 (scope), 不会因异常而忘记解锁。
- 只用非递归的 mutex (即不可重入的 mutex)。
- 不手工调用 lock() 和 unlock() 函数, 一切交给栈上的 Guard 对象的构造和析构函数负责。Guard 对象的生命期正好等于临界区 (分析对象在什么时候析构是 C++ 程序员的基本功)。这样我们保证始终在同一个函数同一个 scope 里对某个 mutex 加锁和解锁。避免在 foo() 里加锁, 然后跑到 bar() 里解锁; 也避免在不同的语句分支中分别加锁、解锁。这种做法被称为 Scoped Locking⁷。
- 在每次构造 Guard 对象的时候, 思考一路上 (调用栈上) 已经持有的锁, 防止因加锁顺序不同而导致死锁 (deadlock)。由于 Guard 对象是栈上对象, 看函数调用栈就能分析用锁的情况, 非常便利。

次要原则有:

- 不使用跨进程的 mutex, 进程间通信只用 TCP sockets。
- 加锁、解锁在同一个线程, 线程 a 不能去 unlock 线程 b 已经锁住的 mutex (RAII 自动保证)。
- 别忘了解锁 (RAII 自动保证)。
- 不重复解锁 (RAII 自动保证)。
- 必要的时候可以考虑用 PTHREAD_MUTEX_ERRORCHECK 来排错。

mutex 恐怕是最简单的同步原语, 按照上面的几条原则, 几乎不可能用错。我自己从来没有违背过这些原则, 编码时出现问题都很快能定位并修复。

⁶ 请注意, 本书谈的是 Pthreads 里的 mutex, 不是 Windows 里的重量级跨进程 Mutex 内核对象。

⁷ 见 Douglas Schmidt 的论文: <http://www.cs.wustl.edu/~schmidt/PDF/locking-patterns.pdf>。

2.1.1 只使用非递归的 mutex

谈谈我坚持使用非递归的互斥器的个人想法。

mutex 分为递归 (recursive) 和非递归 (non-recursive) 两种, 这是 POSIX 的叫法, 另外的名字是可重入 (reentrant) 与非可重入。这两种 mutex 作为线程间 (inter-thread) 的同步工具时没有区别, 它们的唯一区别在于: 同一个线程可以重复对 recursive mutex 加锁, 但是不能重复对 non-recursive mutex 加锁。

首选非递归 mutex, 绝对不是为了性能, 而是为了体现设计意图。non-recursive 和 recursive 的性能差别其实不大, 因为少用一个计数器, 前者略快一点点而已。在同一个线程里多次对 non-recursive mutex 加锁会立刻导致死锁, 我认为这是它的优点, 能帮助我们思考代码对锁的期求, 并且及早 (在编码阶段) 发现问题。

毫无疑问 recursive mutex 使用起来要方便一些, 因为不用考虑一个线程会自己把自己给锁死了, 我猜这也是 Java 和 Windows 默认提供 recursive mutex 的原因。(Java 语言自带的 intrinsic lock 是可重入的, 它的 util.concurrent 库里提供 ReentrantLock, Windows 的 CRITICAL_SECTION 也是可重入的。似乎它们都不提供轻量级的 non-recursive mutex。)

正因为它方便, recursive mutex 可能会隐藏代码里的一些问题。典型情况是你以为拿到一个锁就能修改对象了, 没想到外层代码已经拿到了锁, 正在修改 (或读取) 同一个对象呢。来看一个具体的例子 (recipes/thread/test/NonRecursiveMutex_test.cc):

```
MutexLock mutex;
std::vector<Foo> foos;

void post(const Foo& f)
{
    MutexLockGuard lock(mutex);
    foos.push_back(f);
}

void traverse()
{
    MutexLockGuard lock(mutex);
    for (std::vector<Foo>::const_iterator it = foos.begin();
         it != foos.end(); ++it)
    {
        it->doit();
    }
}
```

post() 加锁, 然后修改 foos 对象; traverse() 加锁, 然后遍历 foos 向量。这些都是正确的。

将来有一天, `Foo::doit()` 间接调用了 `post()`, 那么会很有戏剧性的结果:

1. `mutex` 是非递归的, 于是死锁了。
2. `mutex` 是递归的, 由于 `push_back()` 可能 (但不总是) 导致 `vector` 迭代器失效, 程序偶尔会 `crash`。

这时候就能体现 `non-recursive` 的优越性: 把程序的逻辑错误暴露出来。死锁比较容易 `debug`, 把各个线程的调用栈打出来⁸, 只要每个函数不是特别长, 很容易看出来是怎么死的, 见 §2.1.2 的例子⁹。或者可以用 `PTHREAD_MUTEX_ERRORCHECK` 一下子就能找到错误 (前提是 `MutexLock` 带 `debug` 选项)。程序反正要死, 不如死得有意义一点, 留个“全尸”, 让验尸 (`post-mortem`) 更容易些。

如果确实需要在遍历的时候修改 `vector`, 有两种做法, 一是把修改推后, 记住循环中试图添加或删除哪些元素, 等循环结束了再依记录修改 `foos`; 二是用 `copy-on-write`, 见 §2.8 的例子。

如果一个函数既可能在已加锁的情况下调用, 又可能在未加锁的情况下调用, 那么就拆成两个函数:

1. 跟原来的函数同名, 函数加锁, 转而调用第 2 个函数。
2. 给函数名加上后缀 `WithLockHold`, 不加锁, 把原来的函数体搬过来。

就像这样:

```
void post(const Foo& f)
{
    MutexLockGuard lock(mutex);
    postWithLockHold(f); // 不用担心开销, 编译器会自动内联的
}

// 引入这个函数是为了体现代码作者的意图, 尽管 push_back 通常可以手动内联
void postWithLockHold(const Foo& f)
{
    foos.push_back(f);
}
```

这有可能出现两个问题 (感谢水木网友 `ilovecpp` 提出):

- (a) 误用了加锁版本, 死锁了。
- (b) 误用了不加锁版本, 数据损坏了。

⁸ `gdb` 中使用 `thread apply all bt` 命令。

⁹ 另一方面支持了“函数不要写得过长”这一观点。

对于 (a)，仿造 §2.1.2 的办法能比较容易地排错。对于 (b)，如果 Pthreads 提供 `isLockedByThisThread()` 就好办，可以写成：

```
void postWithLockHold(const Foo& f)
{
    assert(mutex.isLockedByThisThread()); // muduo::MutexLock 提供了这个成员函数
    // ...
}
```

另外，`WithLockHold` 这个显眼的后缀也让程序中的误用容易暴露出来。

C++ 没有 annotation，不能像 Java 那样给 method 或 field 标上 `@GuardedBy` 注解，需要程序员自己小心在意。虽然这里的办法不能一劳永逸地解决全部多线程错误，但能帮上一点是一点了。

我还没有遇到过需要使用 recursive mutex 的情况，我想将来遇到了都可以借助 wrapper 改用 non-recursive mutex，代码只会更清晰。

Pthreads 的权威专家，《Programming with POSIX Threads》的作者 David Butenhof 也排斥使用 recursive mutex。他说：¹⁰

First, implementation of efficient and reliable threaded code revolves around one simple and basic principle: follow your design. That implies, of course, that you have a design, and that you understand it.

A correct and well understood design does not require recursive mutexes.
(后略)

回到正题。本文这里只谈了 mutex 本身的正确使用，在 C++ 里多线程编程还会遇到其他一些 race condition，请参看第 1 章。

性能注脚：Linux 的 Pthreads mutex 采用 `futex(2)` 实现¹¹，不必每次加锁、解锁都陷入系统调用，效率不错。Windows 的 `CRITICAL_SECTION` 也是类似的，不过它可以嵌入一小段 spin lock。在多 CPU 系统上，如果不能立刻拿到锁，它会先 spin 一小段时间，如果还不能拿到锁，才挂起当前线程¹²。

2.1.2 死锁

前面说过，如果坚持只使用 Scoped Locking，那么在出现死锁的时候很容易定位。考虑下面这个线程自己与自己死锁的例子（`recipes/thread/test/SelfDeadLock.cc`）。

¹⁰ <http://zaval.org/resources/library/butenhof1.html>

¹¹ <http://www.akkadia.org/drepper/futex.pdf>

¹² [http://msdn.microsoft.com/en-us/library/windows/desktop/ms682530\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms682530(v=vs.85).aspx)

```

1 class Request
2 {
3 public:
4     void process() // __attribute__((noinline))
5     {
6         muduo::MutexLockGuard lock(mutex_);
7         // ...
8         print(); // 原本没有这行, 某人为了调试程序不小心添加了。
9     }
10
11     void print() const // __attribute__((noinline))
12     {
13         muduo::MutexLockGuard lock(mutex_);
14         // ...
15     }
16
17 private:
18     mutable muduo::MutexLock mutex_;
19 };
20
21 int main()
22 {
23     Request req;
24     req.process();
25 }

```

在上面这个例子中, 原本没有 L8, 在添加它之后, 程序立刻出现了死锁。要调试定位这种死锁很容易, 只要把函数调用栈打印出来, 结合源码一看, 我们立刻就会发现第 6 帧 `Request::process()` 和第 5 帧 `Request::print()` 先后对同一个 `mutex` 上锁, 引发了死锁。(必要的时候可以加上 `__attribute__` 来防止函数 `inline` 展开。)

```

$ gdb ./self_deadlock core
(gdb) bt
#0  __lll_lock_wait () at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:136
#1  _L_lock_953 () from /lib/libpthread.so.0
#2  __pthread_mutex_lock (mutex=0x7ffffecf57bf0) at pthread_mutex_lock.c:61
#3  muduo::MutexLock::lock () at test/./Mutex.h:49
#4  MutexLockGuard () at test/./Mutex.h:75
#5  Request::print () at test/SelfDeadLock.cc:14
#6  Request::process () at test/SelfDeadLock.cc:9
#7  main () at test/SelfDeadLock.cc:24

```

要修复这个错误也很容易, 按前面的办法, 从 `Request::print()` 抽取出¹³ `Request::printWithLockHold()`, 并让 `Request::print()` 和 `Request::process()` 都调用它即可。

再来看一个更真实的两个线程死锁的例子 (`recipes/thread/test/MutualDeadLock.cc`)。

¹³ 即 `extract method` 重构手法。

有一个 Inventory (清单) class, 记录当前的 Request 对象。¹⁴ 容易看出, 下面这个 Inventory class 的 add() 和 remove() 成员函数都是线程安全的, 它使用了 mutex 来保护共享数据 requests_。

```
class Inventory
{
public:
    void add(Request* req)
    {
        muduo::MutexLockGuard lock(mutex_);
        requests_.insert(req);
    }

    void remove(Request* req) // __attribute__((noinline))
    {
        muduo::MutexLockGuard lock(mutex_);
        requests_.erase(req);
    }

    void printAll() const;

private:
    mutable muduo::MutexLock mutex_;
    std::set<Request*> requests_;
};
```

Inventory g_inventory; // 为了简单起见, 这里使用了全局对象。

Request class 与 Inventory class 的交互逻辑很简单, 在处理 (process) 请求的时候, 往 g_inventory 中添加自己。在析构的时候, 从 g_inventory 中移除自己。目前看来, 整个程序还是线程安全的。

```
1 class Request
2 {
3     public:
4         void process() // __attribute__((noinline))
5         {
6             muduo::MutexLockGuard lock(mutex_);
7             g_inventory.add(this);
8             // ...
9         }
10
11     ~Request() __attribute__((noinline))
12     {
13         muduo::MutexLockGuard lock(mutex_);
14         sleep(1); // 为了容易复现死锁, 这里用了延时
15         g_inventory.remove(this);
16     }
```

¹⁴ 为了简单起见, 这里没有使用第 1 章介绍的 shared_ptr/weak_ptr 来管理 Request。

```

17
18     void print() const __attribute__((noinline))
19     {
20         muduo::MutexLockGuard lock(mutex_);
21         // ...
22     }
23
24 private:
25     mutable muduo::MutexLock mutex_;
26 };

```

Inventory class 还有一个功能是打印全部已知的 Request 对象。Inventory::printAll() 里的逻辑单独看是没问题的，但是它有可能引发死锁。

```

void Inventory::printAll() const
{
    muduo::MutexLockGuard lock(mutex_);
    sleep(1); // 为了容易复现死锁，这里用了延时
    for (std::set<Request*>::const_iterator it = requests_.begin();
         it != requests_.end();
         ++it)
    {
        (*it)->print();
    }
    printf("Inventory::printAll() unlocked\n");
}

```

下面这个程序运行起来发生了死锁：

```

void threadFunc()
{
    Request* req = new Request;
    req->process();
    delete req;
}

int main()
{
    muduo::Thread thread(threadFunc);
    thread.start();
    usleep(500 * 1000); // 为了让另一个线程等在前面第 14 行的 sleep() 上。
    g_inventory.printAll();
    thread.join();
}

```

通过 gdb 查看两个线程的函数调用栈，我们发现两个线程都等在 mutex 上 (__lll_lock_wait)，估计是发生了死锁。因为一个程序中的线程一般只会等在 condition variable 上，或者等在 epoll_wait 上 (p. 73)。

```
$ gdb ./mutual_deadlock core
```

```
(gdb) thread apply all bt
```

```
Thread 1 (Thread 31229): # 这是 main() 线程
#0 __lll_lock_wait () at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:136
#1 __L_lock_953 () from /lib/libpthread.so.0
#2 __pthread_mutex_lock (mutex=0xecd150) at pthread_mutex_lock.c:61
#3 muduo::MutexLock::lock (this=0xecd150) at test/./Mutex.h:49
#4 MutexLockGuard (this=0xecd150) at test/./Mutex.h:75
#5 Request::print (this=0xecd150) at test/MutualDeadLock.cc:51
#6 Inventory::printAll (this=0x605aa0) at test/MutualDeadLock.cc:67
#7 0x00000000403368 in main () at test/MutualDeadLock.cc:84

Thread 2 (Thread 31230): # 这是 threadFunc() 线程
#0 __lll_lock_wait () at ../nptl/sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:136
#1 __L_lock_953 () from /lib/libpthread.so.0
#2 __pthread_mutex_lock (mutex=0x605aa0) at pthread_mutex_lock.c:61
#3 muduo::MutexLock::lock (this=0x605aa0, req=0x80) at test/./Mutex.h:49
#4 MutexLockGuard (this=0x605aa0, req=0x80) at test/./Mutex.h:75
#5 Inventory::remove (this=0x605aa0, req=0x80) at test/MutualDeadLock.cc:19
#6 ~Request (this=0xecd150, ...) at test/MutualDeadLock.cc:46
#7 threadFunc () at test/MutualDeadLock.cc:76
#8 boost::function0<void>::operator() (this=0x7fff21c10310)
  at /usr/include/boost/function/function_template.hpp:1013
#9 muduo::Thread::runInThread (this=0x7fff21c10310) at Thread.cc:113
#10 muduo::Thread::startThread (obj=0x605aa0) at Thread.cc:105
#11 start_thread (arg=<value optimized out>) at pthread_create.c:300
#12 clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:112
```

注意到 main() 线程是先调用 Inventory::printAll(#6) 再调用 Request::print(#5)，而 threadFunc() 线程是先调用 Request::~~Request(#6) 再调用 Inventory::remove(#5)。这两个调用序列对两个 mutex 的加锁顺序正好相反，于是造成了经典的死锁。见图 2-1，Inventory class 的 mutex 的临界区由灰底表示，Request class 的 mutex 的临界区由斜纹表示。一旦 main() 线程中的 printAll() 在另一个线程的 ~Request() 和 remove() 之间开始执行，死锁已不可避免。

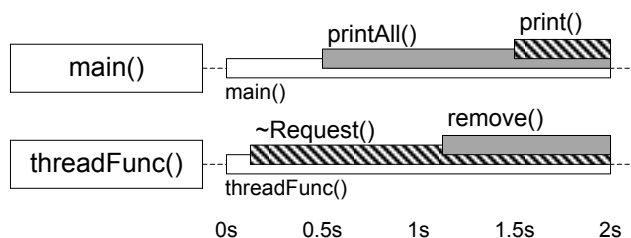


图 2-1

思考：如果 printAll() 晚于 remove() 执行，还会出现死锁吗？

练习：修改程序，让 ~Request() 在 printAll() 和 print() 之间开始执行，复现另一种可能的死锁时序。

这里也出现了第1章所说的对象析构的 *race condition*，即一个线程正在析构对象，另一个线程却在调用它的成员函数。

解决死锁的办法很简单，要么把 `print()` 移出 `printAll()` 的临界区，这可以用 §2.8 介绍的办法；要么把 `remove()` 移出 `~Request()` 的临界区，比如交换 p. 37 中 L13 和 L15 两行代码的位置。当然这没有解决对象析构的 *race condition*，留给读者当做练习吧。

思考：`Inventory::printAll` → `Request::print` 有没有可能与 `Request::process` → `Inventory::add` 发生死锁？

死锁会让程序行为失常，其他一些锁使用不当则会影响性能，例如潘爱民老师写的《*Lock Convoys Explained*》¹⁵ 详细解释了一种性能衰退的现象。除此之外，编写高性能多线程程序至少还要知道 *false sharing* 和 CPU cache 效应，可看脚注中的这几篇文章^{16 17 18}。

2.2 条件变量（*condition variable*）

互斥器（*mutex*）是加锁原语，用来排他性地访问共享数据，它不是等待原语。在使用 *mutex* 的时候，我们一般都会期望加锁不要阻塞，总是能立刻拿到锁。然后尽快访问数据，用完之后尽快解锁，这样才能不影响并发性和性能。

如果需要等待某个条件成立，我们应该使用条件变量（*condition variable*）。条件变量顾名思义是一个或多个线程等待某个布尔表达式为真，即等待别的线程“唤醒”它。条件变量的学名叫管程（*monitor*）。Java Object 内置的 `wait()`、`notify()`、`notifyAll()` 是条件变量¹⁹。

条件变量只有一种正确使用的方式，几乎不可能用错。对于 *wait* 端：

1. 必须与 *mutex* 一起使用，该布尔表达式的读写需受此 *mutex* 保护。
2. 在 *mutex* 已上锁的时候才能调用 `wait()`。
3. 把判断布尔条件和 `wait()` 放到 *while* 循环中。

¹⁵ <http://blog.csdn.net/panaimin/article/details/5981766>

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

¹⁷ http://www.aristeia.com/TalkNotes/ACCU2011_CPU_Caches.pdf <http://www.akkadia.org/drepper/cpumemory.pdf>

¹⁸ <http://igoro.com/archive/gallery-of-processor-cache-effects/> <http://simplygenius.net/Article/FalseSharing>

¹⁹ Java 的这三个函数以容易用错著称，一般建议用 `java.util.concurrent` 中的同步原语。

写成代码是：

```
muduo::MutexLock mutex;
muduo::Condition cond(mutex);
std::deque<int> queue;

int dequeue()
{
    MutexLockGuard lock(mutex);
    while (queue.empty()) // 必须用循环；必须在判断之后再 wait()
    {
        cond.wait(); // 这一步会原子地 unlock mutex 并进入等待，不会与 enqueue 死锁
        // wait() 执行完毕时会自动重新加锁
    }
    assert(!queue.empty());
    int top = queue.front();
    queue.pop_front();
    return top;
}
```

上面的代码中必须用 `while` 循环来等待条件变量，而不能用 `if` 语句，原因是 `spurious wakeup`²⁰。这也是面试多线程编程的常见考点。

对于 `signal/broadcast` 端：

1. 不一定要在 `mutex` 已上锁的情况下调用 `signal`（理论上）。
2. 在 `signal` 之前一般要修改布尔表达式。
3. 修改布尔表达式通常要用 `mutex` 保护（至少用作 `full memory barrier`）。
4. 注意区分 `signal` 与 `broadcast`：“`broadcast` 通常用于表明状态变化，`signal` 通常用于表示资源可用。（`broadcast should generally be used to indicate state change rather than resource availability.`）²¹”

写成代码是²²：

```
void enqueue(int x)
{
    MutexLockGuard lock(mutex);
    queue.push_back(x);
    cond.notify(); // 可以移出临界区之外
}
```

上面的 `dequeue()/enqueue()` 实际上实现了一个简单的容量无限的（`unbounded`）`BlockingQueue`²³。

²⁰ http://en.wikipedia.org/wiki/Spurious_wakeup

²¹ [RWC] “Know when to broadcast—and when to signal.”

²² `muduo::Condition` 采用了 `notify()` 和 `notifyAll()` 为函数名，避免重载 `signal` 这个术语。

²³ 实际使用时一般会做成类模板，如 `muduo/base/BlockingQueue.h`。

思考：enqueue() 中每次添加元素都会调用 Condition::notify(), 如果改成只在 queue.size() 从 0 变 1 的时候才调用 Condition::notify(), 会出现什么后果？²⁴

条件变量是非常底层的同步原语，很少直接使用，一般都是用它来实现高层的同步措施，如 BlockingQueue<T> 或 CountdownLatch。

倒计时 (CountDownLatch) ²⁵ 是一种常用且易用的同步手段。它主要有两种用途：

- 主线程发起多个子线程，等这些子线程各自都完成一定的任务之后，主线程才继续执行。通常用于主线程等待多个子线程完成初始化。
- 主线程发起多个子线程，子线程都等待主线程，主线程完成其他一些任务之后通知所有子线程开始执行。通常用于多个子线程等待主线程发出“起跑”命令。

当然我们可以直接用条件变量来实现以上两种同步。不过如果用 CountdownLatch 的话，程序的逻辑更清晰。CountDownLatch 的接口很简单：

```
class CountdownLatch : boost::noncopyable
{
public:
    explicit CountdownLatch(int count); // 倒数几次
    void wait();                       // 等待计数值变为 0
    void countDown();                 // 计数减一

private:
    mutable MutexLock mutex_;
    Condition condition_;
    int count_;
};
```

CountDownLatch 的实现同样简单，几乎就是条件变量的教科书式应用：

// 构造函数见第 48 页

```
void CountdownLatch::wait()
{
    MutexLockGuard lock(mutex_);
    while (count_ > 0)
        condition_.wait();
}

void CountdownLatch::countDown()
{
    MutexLockGuard lock(mutex_);
    --count_;
    if (count_ == 0)
        condition_.notifyAll();
}
```

²⁴ <http://blog.csdn.net/Solstice/article/details/5829421#comments>

²⁵ muduo/base/CountDownLatch.{h,cc}

注意到 `CountDownLatch::countDown()` 使用的是 `Condition::notifyAll()`，而前面 p. 41 的 `enqueue()` 使用的是 `Condition::notify()`，这都是有意为之。请读者思考，如果交换两种用法会出现什么情况？

互斥器和条件变量构成了多线程编程的全部必备同步原语，用它们即可完成任何多线程同步任务，二者不能相互替代。²⁶ 我认为应该精通这两个同步原语的用法，先学会编写正确的、安全的多线程程序，再在必要的时候考虑用其他“高技术”手段提高性能，如果确实能提高性能的话。千万不要连 `mutex` 都还没学会、用好，一上来就考虑 `lock-free` 设计²⁷。

2.3 不要用读写锁和信号量

读写锁（Readers-Writer lock，简称为 `rwlock`）是个看上去很美的抽象，它明确区分了 `read` 和 `write` 两种行为。

初学者常干的一件事情是，一见到某个共享数据结构频繁读而很少写，就把 `mutex` 替换为 `rwlock`。甚至首选 `rwlock` 来保护共享状态，这不见得是正确的。²⁸

- 从正确性方面来说，一种典型的易犯错误是在持有 `read lock` 的时候修改了共享数据。这通常发生在程序的维护阶段，为了新增功能，程序员不小心在原来 `read lock` 保护的函数中调用了会修改状态的函数。这种错误的后果跟无保护并发读写共享数据是一样的。
- 从性能方面来说，读写锁不见得比普通 `mutex` 更高效。无论如何 `reader lock` 加锁的开销不会比 `mutex lock` 小，因为它要更新当前 `reader` 的数目。如果临界区很小²⁹，锁竞争不激烈，那么 `mutex` 往往会更快。见 §1.9 的例子。
- `reader lock` 可能允许提升（`upgrade`）为 `writer lock`，也可能不允许提升³⁰。考虑 §2.1.1 的 `post()` 和 `traverse()` 示例，如果用读写锁来保护 `foos` 对象，那么 `post()` 应该持有写锁，而 `traverse()` 应该持有读锁。如果允许把读锁提升为写锁，后果跟使用 `recursive mutex` 一样，会造成迭代器失效，程序崩溃。如果不允许提升，后果跟使用 `non-recursive mutex` 一样，会造成死锁。我宁愿程序死锁，留个“全尸”好查验。

²⁶ 就像与非门和 D 触发器构成了数字电路设计所需的全部基础元件一样，用它们可以完成任何组合和同步时序逻辑电路设计。

²⁷ <http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279>

²⁸ [RWC] “Be wary of readers/writer locks.”

²⁹ 在多线程编程中，我们总是设法缩短临界区，不是吗？

³⁰ Pthreads `rwlock` 不允许提升。

- 通常 reader lock 是可重入的，writer lock 是不可重入的。但是为了防止 writer 饥饿，writer lock 通常会阻塞后来的 reader lock，因此 reader lock 在重入的时候可能死锁。另外，在追求低延迟读取的场合也不适用读写锁，见 p. 55。

muduo 线程库有意不提供读写锁的封装，因为我还没有在工作中遇到过用 rwlock 替换普通 mutex 会显著提高性能的例子。相反，我们一般建议首选 mutex。

遇到并发读写，如果条件合适，我通常会用 §2.8 的办法，而不用读写锁，同时避免 reader 被 writer 阻塞。如果确实对并发读写有极高的性能要求，可以考虑 read-copy-update³¹。

信号量 (Semaphore)：我没有遇到过需要使用信号量的情况，无从谈及个人经验。我认为信号量不是必备的同步原语，因为条件变量配合互斥器可以完全替代其功能，而且更不易用错。除了 [RWC] 指出的“semaphore has no notion of ownership”之外，信号量的另一个问题在于它有自己的计数值，而通常我们自己的数据结构也有长度值，这就造成了同样的信息存了两份，需要时刻保持一致，这增加了程序员的负担和出错的可能。如果要控制并发度，可以考虑用 `muduo::ThreadPool`。

说一句不知天高地厚的话，如果程序里需要解决如“哲学家就餐”之类的复杂 IPC 问题，我认为应该首先检讨这个设计：为什么线程之间会有如此复杂的资源争抢（一个线程要同时抢到两个资源，一个资源可以被两个线程争夺）？如果在工作中遇到，我会把“想吃饭”这个事情专门交给一个为各位哲学家分派餐具的线程来做，然后每个哲学家等在一个简单的 condition variable 上，到时间了有人通知他去吃饭。从哲学上说，教科书上的解决方案是平权，每个哲学家有自己的线程，自己去拿筷子；我宁愿用集权的方式，用一个线程专门管餐具的分配，让其他哲学家线程拿个号等在食堂门口好了。这样不损失多少效率，却让程序简单很多。虽然 Windows 的 `WaitForMultipleObjects` 让这个问题 trivial 化，但在 Linux 下正确模拟 `WaitForMultipleObjects` 不是普通程序员该干的。

Pthreads 还提供了 barrier 这个同步原语，我认为不如 `CountDownLatch` 实用。

2.4 封装 MutexLock、MutexLockGuard、Condition

本节把前面用到的 `MutexLock`、`MutexLockGuard`、`Condition` 等 class 的代码列出来，前面两个 class 没多大难度，后面那个有点意思。这几个 class 都不允许拷贝构造

³¹ <http://en.wikipedia.org/wiki/Read-copy-update>

和赋值。完整代码可以在 muduo/base 找到。

MutexLock 和 MutexLockGuard 这两个 class 应该能在纸上默写出来，没有太多需要解释的。MutexLock 的附加值在于提供了 isLockedByThisThread() 函数，用于程序断言。它用到的 CurrentThread::tid() 函数将在 §4.3 介绍。

```
class MutexLock : boost::noncopyable
{
public: // 为了节省版面，单行函数都没有正确缩进
    MutexLock()
        : holder_(0)
    { pthread_mutex_init(&mutex_, NULL); }

    ~MutexLock()
    {
        assert(holder_ == 0);
        pthread_mutex_destroy(&mutex_);
    }

    bool isLockedByThisThread()
    { return holder_ == CurrentThread::tid(); }

    void assertLocked()
    { assert(isLockedByThisThread()); }

    void lock() // 仅供 MutexLockGuard 调用，严禁用户代码调用
    {
        pthread_mutex_lock(&mutex_); // 这两行顺序不能反
        holder_ = CurrentThread::tid();
    }

    void unlock() // 仅供 MutexLockGuard 调用，严禁用户代码调用
    {
        holder_ = 0; // 这两行顺序不能反
        pthread_mutex_unlock(&mutex_);
    }

    pthread_mutex_t* getPthreadMutex() // 仅供 Condition 调用，严禁用户代码调用
    { return &mutex_; }

private:
    pthread_mutex_t mutex_;
    pid_t holder_;
};

class MutexLockGuard : boost::noncopyable
{
public:
    explicit MutexLockGuard(MutexLock& mutex)
        : mutex_(mutex)
    { mutex_.lock(); }
```

```

~MutexLockGuard()
{ mutex_.unlock(); }

private:
    MutexLock& mutex_;
};

#define MutexLockGuard(x) static_assert(false, "missing mutex guard var name")

```

注意上面代码的最后一行定义了一个宏，这个宏的作用是防止程序里出现如下错误：

```

void doit()
{
    MutexLockGuard(mutex); // 遗漏变量名，产生一个临时对象又马上销毁了，
                          // 结果没有锁住临界区。
    // 正确写法是 MutexLockGuard lock(mutex);

    // 临界区
}

```

我见过有人把 `MutexLockGuard` 写成 `template`，我没有这么做是因为它的模板类型参数只有 `MutexLock` 一种可能，没有必要随意增加灵活性，于是我手工把模板具现化（`instantiate`）了。此外一种更激进的写法是，把 `lock/unlock` 放到 `private` 区，然后把 `MutexLockGuard` 设为 `MutexLock` 的 `friend`。我认为在注释里告知程序员即可，另外 `check-in` 之前的 `code review` 也很容易发现误用的情况（`grep getPthreadMutex`）。

这段代码没有达到工业强度：

- `mutex` 创建为 `PTHREAD_MUTEX_DEFAULT` 类型，而不是我们预想的 `PTHREAD_MUTEX_NORMAL` 类型（实际上这二者很可能是等同的），严格的做法是用 `mutexattr` 来显示指定 `mutex` 的类型。
- 没有检查返回值。这里不能用 `assert()` 检查返回值，因为 `assert()` 在 `release build` 里是空语句。我们检查返回值的意义在于防止 `ENOMEM` 之类的资源不足情况，这一般只可能在负载很重的产品程序中出现。一旦出现这种错误，程序必须立刻清理现场并主动退出，否则会莫名其妙地崩溃，给事后调查造成困难。这里我们需要 `non-debug` 的 `assert`，或许 `google-glog` 的 `CHECK()` 宏是个不错的思路。

以上两点改进留作练习。

`muduo` 库的一个特点是只提供最常用、最基本的功能，特别有意避免提供多种功能近似的选择。`muduo` 不是“杂货铺”，不会不分青红皂白地把各种有用的、没用的功能全铺开摆出来。`muduo` 删繁就简，举重若轻；减少选择余地，生活更简单。

MutexLock 没有提供 trylock() 函数，因为我没有在生成代码中用过它。我想不出什么时候程序需要“试着去锁一锁”，或许我写过的代码太简单了³²。

Condition class 的实现有点意思。Pthreads condition variable 允许在 wait() 的时候指定 mutex，但是我想不出有什么理由一个 condition variable 会和不同的 mutex 配合使用。Java 的 intrinsic condition 和 Condition class 都不支持这么做，因此我觉得可以放弃这一灵活性，老老实实地一对一好了。

相反，boost::thread 的 condition_variable 是在 wait 的时候指定 mutex，请参观其同步原语的庞杂设计：

- Concept 有四种 Lockable、TimedLockable、SharedLockable、UpgradeLockable。
- Lock 有六种：lock_guard、unique_lock、shared_lock、upgrade_lock、upgrade_to_unique_lock、scoped_try_lock。
- Mutex 有七种：mutex、try_mutex、timed_mutex、recursive_mutex、recursive_try_mutex、recursive_timed_mutex、shared_mutex。

恕我愚钝，见到 boost::thread 这样如 Rube Goldberg Machine 一样让人眼花缭乱的库，我只得三揖绕道而行。很不幸 C++11 的线程库也采纳了这套方案。这些 class 名字也很无厘头，为什么不老老实实用 readers_writer_lock 这样的通俗名字呢？非得增加精神负担，自己发明新名字。我不愿为这样的灵活性付出代价，宁愿自己做几个简简单单的一看就明白的 class 来用，这种简单的几行代码的“轮子”造造也无妨。提供灵活性固然是本事，然而在不需要灵活性的地方把代码写死，更需要大智慧。

下面这个 muduo::Condition class 简单地封装了 Pthreads condition variable，用起来也容易，见本节前面的例子。这里我用 notify/notifyAll 作为函数名，因为 signal 有别的含义，C++ 里的 signal/slot、C 里的 signal handler 等等。就别 overload 这个术语了。

```
class Condition : boost::noncopyable
{
public: // 为了节省版面，单行函数没有正确缩进
    explicit Condition(MutexLock& mutex)
        : mutex_(mutex)
    { pthread_cond_init(&pcond_, NULL); }

    ~Condition() { pthread_cond_destroy(&pcond_); }
```

³² trylock 的一个用途是用来观察 lock contention，见 [RWC] “Consider using nonblocking synchronization routines to monitor contention.”


```

void wait() { pthread_cond_wait(&pcond_, mutex_.getPthreadMutex()); }
void notify() { pthread_cond_signal(&pcond_); }
void notifyAll() { pthread_cond_broadcast(&pcond_); }

private:
    MutexLock& mutex_;
    pthread_cond_t pcond_;
};

```

如果一个 class 要包含 `MutexLock` 和 `Condition`，请注意它们的声明顺序和初始化顺序，`mutex_` 应先于 `condition_` 构造，并作为后者的构造参数：

```

class CountdownLatch
{
public:
    CountdownLatch(int count)
        : mutex_(),
          condition_(mutex_), // 初始化顺序要与成员声明保持一致
          count_(count)
    { }

private:
    mutable MutexLock mutex_; // 顺序很重要，先 mutex 后 condition
    Condition condition_;
    int count_;
};

```

请允许我再次强调，虽然本章花了大量篇幅介绍如何正确使用 `mutex` 和 `condition variable`，但并不代表我鼓励到处使用它们。这两者都是非常底层的同步原语，主要用来实现更高级的并发编程工具。一个多线程程序里如果大量使用 `mutex` 和 `condition variable` 来同步，基本跟用铅笔刀锯大树（孟岩语）没啥区别。

在程序里使用 Pthreads 库有一个额外的好处：分析工具认得它们，懂得其语意。线程分析工具如 Intel Thread Checker 和 Valgrind-Helgrind³³ 等能识别 Pthreads 调用，并依据 happens-before 关系³⁴ 分析程序有无 data race。

2.5 线程安全的 Singleton 实现

研究 Singleton 的线程安全实现的历史会发现很多有意思的事情，人们一度认为 double checked locking（缩写为 DCL）是王道³⁵，兼顾了效率与正确性。后来

³³ <http://valgrind.org/docs/manual/hg-manual.html#hg-manual.data-races.algorithm>

³⁴ <http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf>

³⁵ <http://www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf>

有“神牛”指出由于乱序执行的影响，DCL 是靠不住的^{36 37 38}。Java 开发者还算幸运，可以借助内部静态类的装载来实现。C++ 就比较惨，要么次次锁，要么 eager initialize，或者动用 memory barrier 这样的“大杀器”³⁹。接下来 Java 5 修订了内存模型，并给 volatile 赋予了 acquire/release 语义，这下 DCL（with volatile）又是安全的了。然而 C++ 的内存模型还在修订中⁴⁰，C++ 的 volatile 目前还不能（将来也难说）保证 DCL 的正确性（只在 Visual C++ 2005 及以上版本有效）。

其实没那么麻烦，在实践中用 pthread_once 就行：

```

template<typename T>
class Singleton : boost::noncopyable
{
public:
    static T& instance()
    {
        pthread_once(&ponce_, &Singleton::init);
        return *value_;
    }

private:
    Singleton();
    ~Singleton();

    static void init()
    {
        value_ = new T();
    }

private:
    static pthread_once_t ponce_;
    static T* value_;
};

// 必须在头文件中定义 static 变量
template<typename T>
pthread_once_t Singleton<T>::ponce_ = PTHREAD_ONCE_INIT;

template<typename T>
T* Singleton<T>::value_ = NULL;

```

muduo/base/Singleton.h

³⁶ <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

³⁷ <http://www.javaworld.com/jw-02-2001/jw-0209-double.html>

³⁸ 这个又让我想起了 SQL 注入，10 年前用字符串拼接出 SQL 语句是 Web 开发的通行做法，直到有一天有人利用这个漏洞越权获得并修改网站数据，人们才幡然醒悟，赶紧修补。

³⁹ http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

⁴⁰ C++11 已经有了全新定义的内存模型，见 <http://scottmeyers.blogspot.com/2012/04/information-on-c11-memory-model.html>。

上面这个 Singleton 没有任何花哨的技巧，它用 `pthread_once_t` 来保证 lazy-initialization 的线程安全。线程安全性由 Pthreads 库保证，如果系统的 Pthreads 库有 bug，那就认命吧，多线程程序反正也不可能正确执行了。

使用方法也很简单：

```
Foo& foo = Singleton<Foo>::instance();
```

这个 Singleton 没有考虑对象的销毁。在长时间运行的服务器程序里，这不是一个问题，反正进程也不打算正常退出 (§9.2.2)。在短期运行的程序中，程序退出的时候自然就释放所有资源了（前提是程序里不使用不能由操作系统自动关闭的资源，比如跨进程的 mutex）。在实际的 `muduo::Singleton` class 中，通过 `atexit(3)` 提供了销毁功能⁴¹，聊胜于无罢了。

另外，这个 Singleton 只能调用默认构造函数，如果用户想要指定 T 的构造方式，我们可以用模板特化（template specialization）技术来提供一个定制点，这需要引入另一层间接（another level of indirection）。

2.6 sleep(3) 不是同步原语

我认为 `sleep()`/`usleep()`/`nanosleep()` 只能出现在测试代码中，比如写单元测试的时候⁴²；或者用于有意延长临界区，加速复现死锁的情况，就像 §2.1.2 示范的那样。`sleep` 不具备 memory barrier 语义，它不能保证内存的可见性，见 p. 84 的例子。

生产代码中线程的等待可分为两种：一种是等待资源可用（要么等在 `select/poll/epoll_wait` 上，要么等在条件变量上⁴³）；一种是等着进入临界区（等在 mutex 上）以便读写共享数据。后一种等待通常极短，否则程序性能和伸缩性就会有问题。

在程序的正常执行中，如果需要等待一段已知的时间，应该往 event loop 里注册一个 timer，然后在 timer 的回调函数里接着干活，因为线程是个珍贵的共享资源，不能轻易浪费（阻塞也是浪费）。如果等待某个事件发生，那么应该采用条件变量或 IO 事件回调，不能用 `sleep` 来轮询。不要使用下面这种业余做法：

⁴¹ Linux 上 `sysconf(_SC_ATEXIT_MAX)`；返回一个足够大的数，不必担心 `ATEXIT_MAX(32)` 的限制。

⁴² 涉及时间的单元测试不那么好写，短的如一两秒，可以用 `sleep()`；长的如一小时、一天，则得想其他办法，比如把算法提取出来并把时间注入进去。

⁴³ 等待 `BlockingQueue/CountDownLatch` 亦可归入此类。

```
while (true) {  
    if (!dataAvailable)  
        sleep(some_time);  
    else  
        consumeData();  
}
```

如果多线程的安全性和效率要靠代码主动调用 `sleep` 来保证，这显然是设计出了问题。等待某个事件发生，正确的做法是用 `select()` 等价物或 `Condition`，抑或（更理想地）高层同步工具；在用户态做轮询（`polling`）是低效的。

2.7 归纳与总结

前面几节内容归纳如下：

- 线程同步的四项原则，尽量用高层同步设施（线程池、队列、倒计时）；
- 使用普通互斥器和条件变量完成剩余的同步任务，采用 RAII 惯用手法（idiom）和 Scoped Locking。

用好这几样东西，基本上就能应付多线程服务端开发的各种场合。或许有人会觉得性能没有发挥到极致。我认为，应该先把程序写正确（并尽量保持清晰和简单），然后再考虑性能优化，如果确实还有必要优化的话。这在多线程下仍然成立。让一个正确的程序变快，远比“让一个快的程序变正确”容易得多。

在现代的多核计算背景下，多线程是不可避免的。尽管在一定程度上可以通过 `framework` 来屏蔽，让你感觉像是在写单线程程序，比如 `Java Servlet`。了解 `under the hood` 发生了什么对于编写这种程序也会有帮助。

多线程编程是一项重要的个人技能，不能因为它难就本能地排斥，现在的软件开发比起 10 年、20 年前已经难了不知道多少倍。掌握多线程编程，才能更理智地选择用还是不用多线程，因为你能预估多线程实现的难度与收益，在一开始做出正确的选择。要知道把一个单线程程序改成多线程的，往往比从头实现一个多线程的程序更困难。要明白多线程编程中哪些是能做的，哪里是一般程序员应该避开的雷区。

掌握同步原语和它们的适用场合是多线程编程的基本功。以我的经验，熟练使用文中提到的同步原语，就能比较容易地编写线程安全的程序。本文没有考虑 `signal` 对多线程编程的影响（§4.10），Unix 的 `signal` 在多线程下的行为比较复杂，一般要靠底层的网络库（如 `Reactor`）加以屏蔽，避免干扰上层应用程序的开发。

通篇来看,“效率”并不是我的主要考虑点,我提倡正确加锁而不是自己编写 lock-free 算法(使用原子整数除外),更不要想当然地自己发明同步设施⁴⁴。在没有实测数据支持的情况下,妄谈哪种做法效率更高是靠不住的⁴⁵,不能听信传言或凭感觉“优化”。很多人误认为用锁会让程序变慢,其实真正影响性能的不是锁,而是锁争用(lock contention)⁴⁶。在程序的复杂度和性能之前取得平衡,并考虑未来两三年扩容的可能(无论是CPU变快、核数变多,还是机器数量增加、网络升级)。我认为在分布式系统中,多机伸缩性(scale out)比单机的性能优化更值得投入精力。

本章内容记录了我目前对多线程编程的理解,用文中介绍的手法,我能化繁为简,编写容易验证其正确性的多线程程序,解决自己面临的全部多线程编程任务。如果本章的观点与你的经验不合,比如你使用了我没有推荐使用的技术或手法(共享内存、信号量等等),只要你理由充分,但行无妨。

2.8 借 shared_ptr 实现 copy-on-write

本节解决 §2.1 的几个未决问题:

- §2.1.1 post() 和 traverse() 死锁。
- §2.1.2 把 Request::print() 移出 Inventory::printAll() 临界区。
- §2.1.2 解决 Request 对象析构的 race condition。

然后再示范用普通 mutex 替换读写锁。解决办法都基于同一个思路,那就是用 shared_ptr 来管理共享数据。原理如下:

- shared_ptr 是引用计数型智能指针,如果当前只有一个观察者,那么引用计数的值为 1⁴⁷。
- 对于 write 端,如果发现引用计数为 1,这时可以安全地修改共享对象,不必担心有人正在读它。
- 对于 read 端,在读之前把引用计数加 1,读完之后减 1,这样保证在读的期间其引用计数大于 1,可以阻止并发写。
- 比较难的是,对于 write 端,如果发现引用计数大于 1,该如何处理? sleep() 一小段时间肯定是错的。

⁴⁴ 《Ad Hoc Synchronization Considered Harmful》https://www.usenix.org/events/osdi10/tech/full_papers/Xiong.pdf。

⁴⁵ <http://pdos.csail.mit.edu/papers/linux/osdi10.pdf>

⁴⁶ <http://preshing.com/20111118/locks-arent-slow-lock-contention-is>

⁴⁷ 在实际代码中判断 shared_ptr::unique() 是否为 true。

先来看一个简单的例子，解决 §2.1.1 中的 `post()` 和 `traverse()` 死锁。⁴⁸ 数据结构改成：

```
typedef std::vector<Foo> FooList;
typedef boost::shared_ptr<FooList> FooListPtr;
MutexLock mutex;
FooListPtr g_foos;
```

在 read 端，用一个栈上局部 `FooListPtr` 变量当做“观察者”，它使得 `g_foos` 的引用计数增加 (L6)。`traverse()` 函数的临界区是 L4~L8，临界区内只读了一次共享变量 `g_foos`（这里多线程并发读写 `shared_ptr`，因此必须用 `mutex` 保护），比原来的写法大为缩短。而且多个线程同时调用 `traverse()` 也不会相互阻塞。

```
1 void traverse()
2 {
3     FooListPtr foos;
4     {
5         MutexLockGuard lock(mutex);
6         foos = g_foos;
7         assert(!g_foos.unique());
8     }
9
10    // assert(!foos.unique()); 这个断言不成立
11    for (std::vector<Foo>::const_iterator it = foos->begin();
12         it != foos->end(); ++it)
13    {
14        it->doit();
15    }
16 }
```

关键看 write 端的 `post()` 该如何写。按照前面的描述，如果 `g_foos.unique()` 为 `true`，我们可以放心地在原地 (in-place) 修改 `FooList`。如果 `g_foos.unique()` 为 `false`，说明这时别的线程正在读取 `FooList`，我们不能原地修改，而是复制一份 (L23)，在副本上修改 (L27)。这样就避免了死锁。

```
17 void post(const Foo& f)
18 {
19     printf("post\n");
20     MutexLockGuard lock(mutex);
21     if (!g_foos.unique())
22     {
23         g_foos.reset(new FooList(*g_foos));
24         printf("copy the whole list\n"); // 练习：将这句话移出临界区
25     }
26     assert(g_foos.unique());
27     g_foos->push_back(f);
28 }
```

⁴⁸ `recipes/thread/test/CopyOnWrite_test.cc`

注意这里临界区包括整个函数（L20~L27），其他写法都是错的。读者可以试着运行这个程序，看看什么时候会打印 L24 的消息。练习：找出以下几种写法的错误。

```
// 错误一：直接修改 g_foos 所指的 FooList
void post(const Foo& f)
{
    MutexLockGuard lock(mutex);
    g_foos->push_back(f);
}

// 错误二：试图缩小临界区，把 copying 移出临界区
void post(const Foo& f)
{
    FooListPtr newFoos(new FooList(*g_foos));
    newFoos->push_back(f);
    MutexLockGuard lock(mutex);
    g_foos = newFoos; // 或者 g_foos.swap(newFoos);
}

// 错误三：把临界区拆成两个小的，把 copying 放到临界区之外
void post(const Foo& f)
{
    FooListPtr oldFoos;
    {
        MutexLockGuard lock(mutex);
        oldFoos = g_foos;
    }
    FooListPtr newFoos(new FooList(*oldFoos));
    newFoos->push_back(f);
    MutexLockGuard lock(mutex);
    g_foos = newFoos; // 或者 g_foos.swap(newFoos);
}
```

希望读者先吃透上面举的这个例子，再来看如何用相同的思路解决剩下的问题。

解决 §2.1.2 把 Request::print() 移出 Inventory::printAll() 临界区有两个做法。其一很简单，把 requests_ 复制一份，在临界区之外遍历这个副本。

```
void Inventory::printAll() const
{
    std::set<Request*> requests
    {
        muduo::MutexLockGuard lock(mutex_);
        requests = requests_;
    }

    // 遍历局部变量 requests，调用 Request::print()
}
```

这么做有一个明显的缺点，它复制了整个 std::set 中的每个元素，开销可能会比较大。如果遍历期间没有其他人修改 requests_，那么我们可以减小开销，这就引出了第二种做法。

第二种做法的要点是用 shared_ptr 管理 std::set，在遍历的时候先增加引用计数，阻止并发修改。当然 Inventory::add() 和 Inventory::remove() 也要相应修改，采用本节前面 post() 和 traverse() 的方案。完整的代码见 recipes/thread/test/RequestInventory_test.cc。

注意目前的方案仍然没有解决 Request 对象析构的 race condition，这点还是留作练习吧。一种可能的答案见 recipes/thread/test/RequestInventory_test2.c。

用普通 mutex 替换读写锁的一个例子

场景：一个多线程的 C++ 程序，24h x 5.5d 运行。有几个工作线程 ThreadWorker{0, 1, 2, 3}，处理客户发过来的交易请求；另外有一个背景线程 ThreadBackground，不定期更新程序内部的参考数据。这些线程都跟一个 hash 表打交道，工作线程只读，背景线程读写，必然要用到一些同步机制，防止数据损坏。这里的示例代码用 std::map 代替 hash 表，意思是一样的：

```
using namespace std;
typedef map<string, vector<pair<string, int> > > Map;
```

Map 的 key 是用户名，value 是一个 vector，里边存的是不同 stock 的最小交易间隔，vector 已经排好序，可以用二分查找。

我们的系统要求工作线程的延迟尽可能小，可以容忍背景线程的延迟略大。一天之内，背景线程对数据更新的次数屈指可数，最多一小时一次，更新的数据来自于网络，所以对更新的及时性不敏感。Map 的数据量也不大，大约一千多条数据。

最简单的同步办法是用读写锁：工作线程加读锁，背景线程加写锁。但是读写锁的开销比普通 mutex 要大，而且是写锁优先，会阻塞后面的读锁。如果工作线程能用最普通的非重入 mutex 实现同步，就不必用读写锁，这能降低工作线程延迟。我们借助 shared_ptr 做到了这一点：(recipes/thread/test/Customer.cc)

```
class CustomerData : boost::noncopyable
{
public:
    CustomerData() : data_(new Map)
    { }

    int query(const string& customer, const string& stock) const;

private:
    typedef std::pair<string, int> Entry;
    typedef std::vector<Entry> EntryList;
    typedef std::map<string, EntryList> Map;
    typedef boost::shared_ptr<Map> MapPtr;
```



```

void update(const string& customer, const EntryList& entries);

// 用 lower_bound 在 entries 里找 stock
static int findEntry(const EntryList& entries, const string& stock);

MapPtr getData() const
{
    MutexLockGuard lock(mutex_);
    return data_;
}

mutable MutexLock mutex_;
MapPtr data_;
};

```

CustomerData::query() 就用前面说的引用计数加 1 的办法，用局部 MapPtr data 变量来持有 Map，防止并发修改。

```

int CustomerData::query(const string& customer, const string& stock) const
{
    MapPtr data = getData();
    // data 一旦拿到，就不再需要锁了。
    // 取数据的时候只有 getData() 内部有锁，多线程并发读的性能很好。

    Map::const_iterator entries = data->find(customer);
    if (entries != data->end())
        return findEntry(entries->second, stock);
    else
        return -1;
}

```

关键看 CustomerData::update() 怎么写。既然要更新数据，那肯定得加锁，如果这时候其他线程正在读，那么不能在原来的数据上修改，得创建一个副本，在副本上修改，修改完了再替换。如果没有用户在读，那么就能直接修改，节约一次 Map 拷贝。

```

// 每次收到一个 customer 的数据更新
void CustomerData::update(const string& customer, const EntryList& entries)
{
    MutexLockGuard lock(mutex_); // update 必须全程持锁
    if (!data_.unique())
    {
        MapPtr newData(new Map(*data_));
        // 在这里打印日志，然后统计日志来判断 worst case 发生的次数
        data_.swap(newData);
    }
    assert(data_.unique());
    (*data_)[customer] = entries;
}

```

注意其中用了 `shared_ptr::unique()` 来判断是不是有人在读，如果有人在读，那么我们不能直接修改，因为 `query()` 并没有全程加锁，只在 `getData()` 内部有锁。`shared_ptr::swap()` 把 `data_` 替换为新副本，而且我们还在锁里，不会有别的线程来读，可以放心地更新。如果别的 `reader` 线程已经刚刚通过 `getData()` 拿到了 `MapPtr`，它会读到稍旧的数据。这不是问题，因为数据更新来自网络，如果网络稍有延迟，反正 `reader` 线程也会读到旧的数据。

如果每次都更新全部数据，而且始终是在同一个线程更新数据，临界区还可以进一步缩小。

```
MapPtr parseData(const string& message); // 解析收到的消息，返回新的 MapPtr

// 函数原型有变，此时网络上传来的是完整的 Map 数据
void CustomerData::update(const string& message)
{
    // 解析新数据，在临界区之外
    MapPtr newData = parseData(message);
    if (newData)
    {
        MutexLockGuard lock(mutex_);
        data_.swap(newData); // 不要用 data_ = newData;
    }
    // 旧数据的析构也在临界区外，进一步缩短了临界区
}
```

据我们测试，大多数情况下更新都是在原来数据上进行的，拷贝的比例还不到 1%，很高效。更准确地说，这不是 `copy-on-write`，而是 `copy-on-other-reading`。

我们将来可能会采用无锁数据结构，不过目前这个实现已经非常好，可以满足我们的要求。

本节介绍的做法与 `read-copy-update` 颇有相似之处，但理解起来要容易得多。

第 3 章

多线程服务器的适用场合与 常用编程模型

本章主要讲我个人在多线程开发方面的一些粗浅经验。总结了一两种常用的线程模型，归纳了进程间通信与线程同步的最佳实践，以期用简单规范的方式开发功能正确、线程安全的多线程程序。本章假定读者已经有多线程编程的知识与经验（本书不是一篇入门教程）。

文中的“多线程服务器”是指运行在 Linux 操作系统上的独占式网络应用程序。硬件平台为 Intel x86-64 系列的多核 CPU，单路或双路 SMP 服务器（每台机器一共拥有四个核或八个核，十几 GB 内存），机器之间用千兆以太网连接。这大概是目前民用 PC 服务器的主流配置。不考虑做分布式存储，只考虑分布式计算，系统的规模大约是几十台服务器到几百台服务器之间。

我将要谈的“网络应用程序”的基本功能可以简单归纳为“收到数据，算一算，再发出去”。在这个简化了的模型里，似乎看不出用多线程的必要，单线程应该也能做得很好。“为什么需要写多线程程序”这个问题容易引发口水战，我放到 §3.5 讨论。请允许我先假定“多线程编程”这一背景。

“服务器”这个词有时指程序，有时指进程，有时指硬件（无论虚拟的或真实的），请注意按上下文区分。另外，本书不考虑虚拟化的场景，当我说“两个进程不在同一台机器上”时，指的是逻辑上不在同一个操作系统里运行，虽然物理上可能位于同一机器虚拟出来的两台“虚拟机”上。

3.1 进程与线程

“进程（process）”是操作里最重要的两个概念之一（另一个是文件），粗略地讲，一个进程是“内存中正在运行的程序”。本书的进程指的是 Linux 操作系统通过

fork() 系统调用产生的那个东西，或者 Windows 下 CreateProcess() 的产物，不是 Erlang 里的那种“轻量级进程 (Actor)”。

每个进程有自己独立的地址空间 (address space)，“在同一个进程”还是“不在同一个进程”是系统功能划分的重要决策点。《Erlang 程序设计》[ERL] 把“进程”比喻为“人”，我觉得十分精当，为我们提供了一个思考的框架。

每个人有自己的记忆 (memory)，人与人通过谈话 (消息传递) 来交流，谈话既可以是面谈 (同一台服务器)，也可以在电话里谈 (不同的服务器，有网络通信)。面谈和电话谈的区别在于，面谈可以立即知道对方是否死了 (crash, SIGCHLD)，而电话谈只能通过周期性的心跳来判断对方是否还活着。

有了这些比喻，设计分布式系统时可以采取“角色扮演”，团队里的几个人各自扮演一个进程，人的角色由进程的代码决定 (管登录的、管消息分发的、管买卖的等等)。每个人有自己的记忆，但不知道别人的记忆，要想知道别人的看法，只能通过交谈 (暂不考虑共享内存这种 IPC)。然后就可以思考：

- 容错 万一有人突然死了
- 扩容 新人中途加进来
- 负载均衡 把甲的活儿挪给乙做
- 退休 甲要修复 bug，先别派新任务，等他做完手上的事情就把他重启

等等各种场景，十分便利。

“线程”这个概念大概是在 1993 年以后才慢慢流行起来的，距今不到 20 年，比不得有 40 年光辉历史的 Unix 操作系统。线程的出现给 Unix 添了不少乱，很多 C 库函数 (strtok()、ctime()) 不是线程安全的，需要重新定义 (§4.2)；signal 的语义也大为复杂化。据我所知，最早支持多线程编程的 (民用) 操作系统是 Solaris 2.2 和 Windows NT 3.1，它们均发布于 1993 年。随后在 1995 年，POSIX threads 标准确立。

线程的特点是共享地址空间，从而可以高效地共享数据。一台机器上的多个进程能高效地共享代码段 (操作系统可以映射为同样的物理内存)，但不能共享数据。如果多个进程大量共享内存，等于是把多进程程序当成多线程来写，掩耳盗铃。

“多线程”的价值，我认为为了更好地发挥多核处理器 (multi-cores) 的效能。在单核时代，多线程没有多大价值。Alan Cox 说过：“A computer is a state machine. Threads are for people who can't program state machines.” (计算机是一台状态机。线程是给那些不能编写状态机程序的人准备的。) 如果只有一块 CPU、一个执行单元，那么确实如 Alan Cox 所说，按状态机的思路去写程序是最高效的，这正好也是下一节展示的编程模型。

3.2 单线程服务器的常用编程模型

[UNP] 对此有很好的总结（第 6 章的 IO 模型、第 30 章的客户端/服务器设计范式），这里不再赘述。据我了解，在高性能的网络程序中，使用得最为广泛的恐怕要数“non-blocking IO + IO multiplexing”这种模型，即 Reactor 模式¹，我知道的有：

- lighttpd，单线程服务器。（Nginx 与之类似，每个工作进程有一个 event loop。）
- libevent，libev。
- ACE，Poco C++ libraries。
- Java NIO，包括 Apache Mina 和 Netty。
- POE（Perl）。
- Twisted（Python）。

相反，Boost.Asio 和 Windows I/O Completion Ports 实现了 Proactor 模式²，应用面似乎要窄一些。此外，ACE 也实现了 Proactor 模式。

在“non-blocking IO + IO multiplexing”这种模型中，程序的基本结构是一个事件循环（event loop），以事件驱动（event-driven）和事件回调的方式实现业务逻辑：

```
// 代码仅为示意，没有完整考虑各种情况
while (!done)
{
    int timeout_ms = max(1000, getNextTimedCallback());
    int retval = ::poll(fds, nfds, timeout_ms);
    if (retval < 0) {
        处理错误，回调用户的 error handler
    } else {
        处理到期的 timers，回调用户的 timer handler
        if (retval > 0) {
            处理 IO 事件，回调用户的 IO event handler
        }
    }
}
```

这里 select(2)/poll(2) 有伸缩性方面的不足，Linux 下可替换为 epoll(4)，其他操作系统也有对应的高性能替代品³。

Reactor 模型的优点很明显，编程不难，效率也不错。不仅可以用于读写 socket，连接的建立（connect(2)/accept(2)）甚至 DNS 解析⁴都可以用非阻塞方式进行，以提高并发度和吞吐量（throughput），对于 IO 密集的应用是个不错的选择。lighttpd

¹ <http://www.cs.wustl.edu/~schmidt/PDF/Reactor1-93.pdf>, [Reactor2-93.pdf](http://www.cs.wustl.edu/~schmidt/PDF/Reactor2-93.pdf), [Reactor.pdf](http://www.cs.wustl.edu/~schmidt/PDF/Reactor.pdf)

² <http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf>

³ <http://www.kegel.com/c10k.html>

⁴ gethostbyname(3) 是阻塞的，对陌生域名解析的耗时可长达数秒。

就是这样，它内部的 `fdevent` 结构十分精妙，值得学习。

基于事件驱动的编程模型也有其本质的缺点，它要求事件回调函数必须是非阻塞的。对于涉及网络 IO 的请求响应式协议，它容易割裂业务逻辑，使其散布于多个回调函数之中，相对不容易理解和维护。现代的语言有一些应对方法（例如 `coroutine`），但是本书只关注 C++ 这种传统语言，因此就不展开讨论了。

3.3 多线程服务器的常用编程模型

这方面我能找到的文献⁵不多，大概有这么几种（见 §6.6 更详细的讨论）：

1. 每个请求创建一个线程，使用阻塞式 IO 操作。在 Java 1.4 引入 NIO 之前，这是 Java 网络编程的推荐做法。可惜伸缩性不佳。
2. 使用线程池，同样使用阻塞式 IO 操作。与第 1 种相比，这是提高性能的措施。
3. 使用 non-blocking IO + IO multiplexing。即 Java NIO 的方式。
4. Leader/Follower 等高级模式。

在默认情况下，我会使用第 3 种，即 non-blocking IO + one loop per thread 模式来编写多线程 C++ 网络服务程序。

3.3.1 one loop per thread

此种模型下，程序里的每个 IO 线程有一个 event loop（或者叫 Reactor），用于处理读写和定时事件（无论周期性的还是单次的），代码框架跟 §3.2 一样。

libev 的作者说⁶：

One loop per thread is usually a good model. Doing this is almost never wrong, sometimes a better-performance model exists, but it is always a good start.

这种方式的好处是：

- 线程数目基本固定，可以在程序启动的时候设置，不会频繁创建与销毁。
- 可以很方便地在线程间调配负载。
- IO 事件发生的线程是固定的，同一个 TCP 连接不必考虑事件并发。

⁵ <http://www.cs.uwaterloo.ca/~brecht/pubs.html> <http://hal.inria.fr/docs/00/67/44/75/PDF/paper.pdf>

⁶ http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#THREADS_AND_COROUTINES

Event loop 代表了线程的主循环, 需要让哪个线程干活, 就把 timer 或 IO channel (如 TCP 连接) 注册到哪个线程的 loop 里即可。对实时性有要求的 connection 可以单独用一个线程; 数据量大的 connection 可以独占一个线程, 并把数据处理任务分摊到另几个计算线程中 (用线程池); 其他次要的辅助性 connections 可以共享一个线程。

对于 non-trivial 的服务端程序, 一般会采用 non-blocking IO + IO multiplexing, 每个 connection/acceptor 都会注册到某个 event loop 上, 程序里有多多个 event loop, 每个线程至多有一个 event loop。

多线程程序对 event loop 提出了更高的要求, 那就是“线程安全”。要允许一个线程往别的线程的 loop 里塞东西⁷, 这个 loop 必须得是线程安全的。如何实现一个优质的多线程 Reactor? 可参考第 8 章。

3.3.2 线程池

不过, 对于没有 IO 而光有计算任务的线程, 使用 event loop 有点浪费, 我会用一种补充方案, 即用 blocking queue 实现的任务队列 (TaskQueue):

```
typedef boost::function<void()> Functor;
BlockingQueue<Functor> taskQueue; // 线程安全的阻塞队列

void workerThread()
{
    while (running) // running 变量是个全局标志
    {
        Functor task = taskQueue.take(); // this blocks
        task(); // 在产品代码中需要考虑异常处理
    }
}
```

用这种方式实现线程池特别容易, 以下是启动容量 (并发数) 为 N 的线程池:

```
int N = num_of_computing_threads;
for (int i = 0; i < N; ++i)
{
    create_thread(&workerThread); // 伪代码: 启动线程
}
```

使用起来也很简单:

```
Foo foo; // Foo 有 calc() 成员函数
boost::function<void()> task = boost::bind(&Foo::calc, &foo);
taskQueue.post(task);
```

⁷ 比方说主 IO 线程收到一个新建连接, 分配给某个子 IO 线程处理。

上面十几行代码就实现了一个简单的固定数目的线程池，功能大概相当于 Java 中的 `ThreadPoolExecutor` 的某种“配置”。当然，在真实的项目中，这些代码都应该封装到一个 `class` 中，而不是使用全局对象。另外需要注意一点：`Foo` 对象的生命期，第 1 章详细讨论了这个问题。

`muduo` 的线程池⁸比这个略复杂，因为要提供 `stop()` 操作。

除了任务队列，还可以用 `BlockingQueue<T>` 实现数据的生产者消费者队列，即 `T` 是数据类型⁹而非函数对象，`queue` 的消费者(s)从中拿到数据进行处理。

`BlockingQueue<T>` 是多线程编程的利器，它的实现可参照 Java `util.concurrent` 里的 `(Array|Linked)BlockingQueue`。这份 Java 代码可读性很高，代码的基本结构和教科书一致（1 个 `mutex`，2 个 `condition variables`），健壮性要高得多。如果不想自己实现，用现成的库更好。`muduo` 里有一个基本的实现，包括无界的 `BlockingQueue` 和有界的 `BoundedBlockingQueue` 两个 `class`。有兴趣的读者还可以试试 Intel Threading Building Blocks 里的 `concurrent_queue<T>`，性能估计会更好。

3.3.3 推荐模式

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

- event loop（也叫 IO loop）用作 IO multiplexing，配合 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。

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，`muduo` 正是这样的网络库。

程序里具体用几个 loop、线程池的大小等参数需要根据应用来设定，基本的原则是“阻抗匹配”，使得 CPU 和 IO 都能高效地运作，具体的例子见 p. 80。

此外，程序里或许还有个别执行特殊任务的线程，比如 logging，这对应用程序来说基本是不可见的，但是在分配资源（CPU 和 IO）的时候要算进去，以免高估了系统的容量。

⁸ `muduo/base/ThreadPool{h,cc}`

⁹ 例如 `std::string` 或 `google::protobuf::Message*`。

3.4 进程间通信只用 TCP

Linux 下进程间通信 (IPC) 的方式数不胜数, 光 [UNPv2] 列出的就有: 匿名管道 (pipe)、具名管道 (FIFO)、POSIX 消息队列、共享内存、信号 (signals) 等等, 更不必说 Sockets 了。同步原语 (synchronization primitives) 也很多, 如互斥器 (mutex)、条件变量 (condition variable)、读写锁 (reader-writer lock)、文件锁 (record locking)、信号量 (semaphore) 等等。

如何选择呢? 根据我的个人经验, 贵精不贵多, 认真挑选三四样东西就能完全满足我的工作需要, 而且每样我都能用得很熟, 不容易犯错。

进程间通信我首选 Sockets (主要指 TCP, 我没有用过 UDP, 也不考虑 Unix domain 协议), 其最大的好处在于: 可以跨主机, 具有伸缩性。反正都是多进程了, 如果一台机器的处理能力不够, 很自然地就能用多台机器来处理。把进程分散到同一局域网的多台机器上, 程序改改 host:port 配置就能继续用。相反, 前面列出的其他 IPC 都不能跨机器¹⁰, 这就限制了 scalability。

在编程上, TCP sockets 和 pipe 都是操作文件描述符, 用来收发字节流, 都可以 read/write/fcntl/select/poll 等。不同的是, TCP 是双向的, Linux 的 pipe 是单向的, 进程间双向通信还得开两个文件描述符, 不方便¹¹; 而且进程要有父子关系才能用 pipe, 这些都限制了 pipe 的使用。在收发字节流这一通信模型下, 没有比 Sockets/TCP 更自然的 IPC 了。当然, pipe 也有一个经典应用场景, 那就是写 Reactor/event loop 时用来异步唤醒 select (或等价的 poll/epoll_wait) 调用¹², Sun HotSpot JVM 在 Linux 就是这么做的¹³。

TCP port 由一个进程独占, 且操作系统会自动回收 (listening port 和已建立连接的 TCP socket 都是文件描述符, 在进程结束时操作系统会关闭所有文件描述符)。这说明, 即使程序意外退出, 也不会给系统留下垃圾, 程序重启之后能比较容易地恢复, 而不需要重启操作系统 (用跨进程的 mutex 就有这个风险)。还有一个好处, 既然 port 是独占的, 那么可以防止程序重复启动, 后面那个进程抢不到 port, 自然就没法初始化了, 避免造成意料之外的结果。

两个进程通过 TCP 通信, 如果一个崩溃了, 操作系统会关闭连接, 另一个进程几乎立刻就能感知, 可以快速 failover。当然应用层的心跳也是必不可少的 (§9.3)。

¹⁰ 比如共享内存效率最高, 但受网络带宽及延迟限制, 无论如何也不能高效地共享两台物理机器的内存。

¹¹ 可以用 socketpair(2) 替代。

¹² 在 Linux 下, 可以用 eventfd(2) 代替, 效率更高。

¹³ <http://blog.csdn.net/haole/article/details/2224055>

与其他 IPC 相比, TCP 协议的一个天生的好处是“可记录、可重现”。tcpdump 和 Wireshark 是解决两个进程间协议和状态争端的好帮手,也是性能(吞吐量、延迟)分析的利器。我们可以借此编写分布式程序的自动化回归测试。也可以用 tcpcopy¹⁴之类的工具进行压力测试。TCP 还能跨语言,服务端和客户端不必使用同一种语言。试想如果用共享内存作为 IPC, C++ 程序如何与 Java 通信,难道用 JNI 吗?

另外,如果网络库带“连接重试”功能的话,我们可以不要求系统里的进程以特定的顺序启动,任何一个进程都能单独重启。换句话说, TCP 连接是可再生的,连接的任何一方都可以退出再启动,重建连接之后就能继续工作,这对开发牢靠的分布式系统意义重大。

使用 TCP 这种字节流(byte stream)方式通信,会有 marshal/unmarshal 的开销,这要求我们选用合适的消息格式,准确地说是 wire format,目前我推荐 Google Protocol Buffers。见 §9.6 关于分布式系统消息格式的讨论。

有人或许会说,具体问题具体分析,如果两个进程在同一台机器,就用共享内存,否则就用 TCP,比如 MS SQL Server 就同时支持这两种通信方式。试问,是否值得为那么一点性能提升而让代码的复杂度大大增加呢?何况 TCP 的 local 吞吐量一点都不低,见 §6.5.1 的测试结果。TCP 是字节流协议,只能顺序读取,有写缓冲;共享内存是消息协议,a 进程填好一块内存让 b 进程来读,基本是“停等(stop wait)”方式。要把这两种方式揉到一个程序里,需要建一个抽象层,封装两种 IPC。这会带来不透明性,并且增加测试的复杂度。而且万一通信的某一方崩溃,状态 reconcile 也会比 sockets 麻烦。(数据刚写到一半,怎么办?)为我所不取。再说了,你舍得让几万块买来的 SQL Server 和其他应用程序分享机器资源吗?生产环境下的数据库服务器往往是独立的高配置服务器,一般不会同时运行其他占资源的程序。

TCP 本身是个数据流协议,除了直接使用它来通信外,还可以在此之上构建 RPC/HTTP/SOAP 之类的上层通信协议,这超过了本章的范围。另外,除了点对点的通信之外,应用级的广播协议也是非常有用的,可以方便地构建可观可控的分布式系统,见 §7.11。

分布式系统中使用 TCP 长连接通信

§3.1 提到,分布式系统的软件设计和功能划分一般应该以“进程”为单位。从宏观上看,一个分布式系统是由运行在多台机器上的多个进程组成的,进程之间采用 TCP 长连接通信。本章讨论分布式系统中单个服务进程的设计方法,第 9 章将谈一谈

¹⁴ <http://code.google.com/p/tcpcopy/>

整个系统的设计。我提倡用多线程，并不是说把整个系统放到一个进程里实现，而是指功能划分之后，在实现每一类服务进程时，在必要时可以借助多线程来提高性能。对于整个分布式系统，要做到能 scale out，即享受增加机器带来的好处。

使用 TCP 长连接的好处有两点：一是容易定位分布式系统中的服务之间的依赖关系。只要在机器上运行 `netstat -tpna | grep :port` 就能立刻列出用到某服务的客户端地址（Foreign 列），然后在客户端的机器上用 `netstat` 或 `lsof` 命令找出是哪个进程发起的连接。这样在迁移服务的时候能有效地防止出现 outage。TCP 短连接和 UDP 则不具备这一特性。二是通过接收和发送队列的长度也较容易定位网络或程序故障。在正常运行时，`netstat` 打印的 Recv-Q 和 Send-Q 都应该接近 0，或者在 0 附近摆动。如果 Recv-Q 保持不变或持续增加，则通常意味着服务进程的处理速度变慢，可能发生了死锁或阻塞。如果 Send-Q 保持不变或持续增加，有可能是对方服务器太忙、来不及处理，也有可能是网络中间某个路由器或交换机故障造成丢包，甚至对方服务器掉线，这些因素都可能表现为数据发送不出去。通过持续监控 Recv-Q 和 Send-Q 就能及早预警性能或可用性故障。以下是服务端线程阻塞造成 Recv-Q 和客户端 Send-Q 激增的例子。

```
$ netstat -tn
Proto Recv-Q Send-Q Local Address       Foreign               # 服务端连接
tcp    78393 0      10.0.0.10:2000      10.0.0.10:39748
tcp    0      132608 10.0.0.10:39748     10.0.0.10:2000      # 客户端连接
tcp    0      52    10.0.0.10:22        10.0.0.4:55572
```

3.5 多线程服务器的适用场合

“服务器开发”包罗万象，本书所指的“服务器开发”的含义请见本章开头，用一句话形容是：跑在多核机器上的 Linux 用户态的没有用户界面的长期运行¹⁵的网络应用程序，通常是分布式系统的组成部件。

开发服务端程序的一个基本任务是处理并发连接，现在服务端网络编程处理并发连接主要有两种方式：

- 当“线程”很廉价时，一台机器上可以创建远高于 CPU 数目的“线程”。这时一个线程只处理一个 TCP 连接（甚至半个），通常使用阻塞 IO（至少看起来如此）。例如，Python `gevent`、Go `goroutine`、Erlang `actor`。这里的“线程”由语言的 runtime 自行调度，与操作系统线程不是一回事。

¹⁵ “长期运行”的意思不是指程序 7×24 不重启，而是程序不会因为无事可做而退出，它会等着下一个请求的到来。例如 `wget` 不是长期运行的，`httpd` 是长期运行的。

- 当线程很宝贵时，一台机器上只能创建与 CPU 数目相当的线程。这时一个线程要处理多个 TCP 连接上的 IO，通常使用非阻塞 IO 和 IO multiplexing。例如，libevent、muduo、Netty。这是原生线程，能被操作系统的任务调度器看见。

在处理并发连接的同时，也要充分发挥硬件资源的作用，不能让 CPU 资源闲置。以上列出的库不是每个都能做到这一点。既然本书讨论的是 C++ 编程，那么只考虑后一种方式，这是在 Linux 下使用 native 语言编写用户态高性能网络程序的最成熟的模式。本节主要讨论的是这些“线程”应该属于一个进程（以下模式 2），还是分属多个进程（模式 3）。

与前文相同，本节的“进程”指的是 fork(2) 系统调用的产物。“线程”指的是 pthread_create() 的产物，因此是宝贵的那种原生线程。而且我指的 Pthreads 是 NPTL 的，每个线程由 clone(2) 产生，对应一个内核的 task_struct。

首先，一个由多台机器组成的分布式系统必然是多进程的（字面意义上），因为进程不能跨 OS 边界。在这个前提下，我们把目光集中到一台机器，一台拥有至少 4 个核的普通服务器。如果要在一台多核机器上提供一种服务或执行一个任务，可用的模式有：（这里的“模式”不是 pattern，而是 model，不巧它们的中译文是一样的。）

1. 运行一个单线程的进程；
2. 运行一个多线程的进程；
3. 运行多个单线程的进程；
4. 运行多个多线程的进程。

这些模式之间的比较已经是老生常谈，简单地总结如下。

- 模式 1 是不可伸缩的（scalable），不能发挥多核机器的计算能力。
- 模式 3 是目前公认的主流模式。它有以下两种子模式：
 - 3a 简单地把模式 1 中的进程运行多份¹⁶
 - 3b 主进程 + worker 进程，如果必须绑定到一个 TCP port，比如 httpd+fastcgi
- 模式 2 是被很多人所鄙视的，认为多线程程序难写，而且与模式 3 相比并没有什么优势。
- 模式 4 更是千夫所指，它不但没有结合 2 和 3 的优点，反而汇聚了二者的缺点。

本文主要想讨论的是模式 2 和模式 3b 的优劣，即：什么时候一个服务器程序应该是多线程的。从功能上讲，没有什么多线程能做到而单线程做不到的，反之亦然，都是状态机嘛（我很高兴看到反例）。从性能上讲，无论是 IO bound 还是 CPU bound 的服务，多线程都没有什么优势。

¹⁶ 如果能用多个 TCP port 对外提供服务的话。

Paul E. McKenney 在《Is Parallel Programming Hard, And, If So, What Can You Do About It?》¹⁷ 第 3.5 节指出, “As a rough rule of thumb, use the simplest tool that will get the job done.” 比方说, 使用速率为 50MB/s 的数据压缩库、在进程创建销毁的开销是 800 μ s、线程创建销毁的开销是 50 μ s 的前提下, 考虑如何执行压缩任务:

- 如果要偶尔压缩 1GB 的文本文件, 预计运行时间是 20s, 那么起一个进程去做是合理的, 因为进程启动和销毁的开销远远小于实际任务的耗时。
- 如果要经常压缩 500kB 的文本数据, 预计运行时间是 10ms, 那么每次都起进程似乎有点浪费了, 可以每次单独起一个线程去做。
- 如果要频繁压缩 10kB 的文本数据, 预计运行时间是 200 μ s, 那么每次起线程似乎也很浪费, 不如直接在当前线程搞定。也可以用一个线程池, 每次把压缩任务交给线程池, 避免阻塞当前线程 (特别要避免阻塞 IO 线程)。

由此可见, 多线程并不是万灵丹 (silver bullet), 它有适用的场合。那么究竟什么时候该用多线程? 在回答这个问题之前, 我先谈谈必须用单线程的场合。

3.5.1 必须用单线程的场合

据我所知, 有两种场合必须使用单线程:

1. 程序可能会 fork(2);
2. 限制程序的 CPU 占用率。

只有单线程程序能 fork(2) 根据后面 §4.9 的分析, 一个设计为可能调用 fork(2) 的程序必须是单线程的, 比如后面 §3.5.3 中提到的“看门狗进程”。多线程程序不是不能调用 fork(2), 而是这么做会遇到很多麻烦, 我想不出做的理由。

一个程序 fork(2) 之后一般有两种行为:

1. 立刻执行 exec(), 变身为另一个程序。例如 shell 和 inetd; 又比如 lighttpd fork() 出子进程, 然后运行 fastcgi 程序。或者集群中运行在计算节点上的负责启动 job 的守护进程 (即我所谓的“看门狗进程”)。
2. 不调用 exec(), 继续运行当前程序。要么通过共享的文件描述符与父进程通信, 协同完成任务; 要么接过父进程传来的文件描述符, 独立完成工作, 例如 20 世纪 80 年代的 Web 服务器 NCSA httpd。

这些行为中, 我认为只有“看门狗进程”必须坚持单线程, 其他的均可替换为多线程程序 (从功能上讲)。

¹⁷ <http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>

单线程程序能限制程序的 CPU 占用率 这个很容易理解，比如在一个 8 核的服务器上，一个单线程程序即便发生 busy-wait（无论是因为 bug，还是因为 overload），占满 1 个 core，其 CPU 使用率也只有 12.5%。¹⁸ 在这种最坏的情况下，系统还是有 87.5% 的计算资源可供其他服务进程使用。

因此对于一些辅助性的程序，如果它必须和主要服务进程运行在同一台机器的话（比如它要监控其他服务进程的状态），那么做成单线程的能避免过分抢夺系统的计算资源。比方说如果要把生产服务器上的日志文件压缩后备份到 NFS 上，那么应该使用普通单线程压缩工具（gzip/bzip2）。它们对系统造成的影响较小，在 8 核服务器上最多占满 1 个 core。如果有人为了“提高速度”，开启了多线程压缩或者同时起多个进程来压缩多个日志文件，有可能造成的结果是非关键任务耗尽了 CPU 资源，正常客户的请求响应变慢。这是我们不愿意看到的。

3.5.2 单线程程序的优缺点

从编程的角度，单线程程序的优势无须赘言：简单。程序的结构一般如 §3.2 所言，是一个基于 IO multiplexing 的 event loop。或者如云风所言¹⁹，直接用阻塞 IO。event loop 的典型代码框架见 §3.2。

Event loop 有一个明显的缺点，它是非抢占的（non-preemptive）。假设事件 a 的优先级高于事件 b，处理事件 a 需要 1ms，处理事件 b 需要 10ms。如果事件 b 稍早于 a 发生，那么当事件 a 到来时，程序已经离开了 poll(2) 调用，并开始处理事件 b。事件 a 要等上 10ms 才有机会被处理，总的响应时间为 11ms。这等于发生了优先级反转。这个缺点可以用多线程来克服，这也是多线程的主要优势。

多线程程序有性能优势吗

前面我说，无论是 IO bound 还是 CPU bound 的服务，多线程都没有什么绝对意义上的性能优势。这句话是说，如果用很少的 CPU 负载就能让 IO 跑满，或者用很少的 IO 流量就能让 CPU 跑满，那么多线程没啥用处。举例来说：

- 对于静态 Web 服务器，或者 FTP 服务器，CPU 的负载较轻，主要瓶颈在磁盘 IO 和网络 IO 方面。这时候往往一个单线程的程序（模式 1）就能撑满 IO。用多线程并不能提高吞吐量，因为 IO 硬件容量已经饱和了。同理，这时增加 CPU 数目也不能提高吞吐量。

¹⁸ 本书中，当句尾是百分号“%”时，句号改用句点“.”，以避免与千分号“‰”相混淆。

¹⁹ http://blog.codingnow.com/2006/04/iocp_kqueue_epoll.html

- CPU 跑满的情况比较少见，这里我只好虚构一个例子。假设有一个服务，它的输入是 n 个整数，问能否从中选出 m 个整数，使其和为 0（这里 $n < 100$, $m > 0$ ）。这是著名的 subset sum 问题，是 NP-Complete 的。对于这样一个“服务”，哪怕很小的 n 值也会让 CPU 算死。比如 $n = 30$ ，一次的输入不过 200 字节（32-bit 整数），CPU 的运算时间却能长达几分钟。对于这种应用，模式 3a 是最适合的，能发挥多核的优势，程序也简单。

也就是说，无论任何一方早早地先到达瓶颈，多线程程序都没啥优势。

说到这里，可能已经有读者不耐烦了：你讲了这么多，都在说单线程的好处，那么多线程究竟有什么用？

3.5.3 适用多线程程序的场景

我认为多线程的适用场景是：提高响应速度，让 IO 和“计算”相互重叠，降低 latency。虽然多线程不能提高绝对性能，但能提高平均响应性能。

一个程序要做成多线程的，大致要满足：

- 有多个 CPU 可用。单机机器上多线程没有性能优势（但或许能简化并发业务逻辑的实现）。
- 线程间有共享数据，即内存中的全局状态。如果没有共享数据，用模型 3b 就行。虽然我们应该把线程间的共享数据降到最低，但不代表没有。
- 共享的数据是可以修改的，而不是静态的常量表。如果数据不能修改，那么可以在进程间用 shared memory，模式 3 就能胜任。
- 提供非均质的服务。即，事件的响应有优先级差异，我们可以用专门的线程来处理优先级高的事件。防止优先级反转。
- latency 和 throughput 同样重要，不是逻辑简单的 IO bound 或 CPU bound 程序。换言之，程序要有相当的计算量。
- 利用异步操作。比如 logging。无论往磁盘写 log file，还是往 log server 发送消息都不应该阻塞 critical path。
- 能 scale up。一个好的多线程程序应该能享受增加 CPU 数目带来的好处，目前主流是 8 核，很快就会用到 16 核的机器了。
- 具有可预测的性能。随着负载增加，性能缓慢下降，超过某个临界点之后会急速下降。线程数目一般不随负载变化。

- 多线程能有效地划分责任与功能，让每个线程的逻辑比较简单，任务单一，便于编码。而不是把所有逻辑都塞到一个 event loop 里，不同类别的事件之间相互影响。

这些条件比较抽象，这里举两个具体的（虽然是虚构的）例子。

假设要管理一个 Linux 服务器机群，这个机群里有 8 个计算节点，1 个控制节点。机器的配置都是一样的，双路四核 CPU，千兆网互联。现在需要编写一个简单的机群管理软件（参考 LLNL 的 SLURM²⁰），这个软件由 3 个程序组成：

1. 运行在控制节点上的 master，这个程序监视并控制整个机群的状态。
2. 运行在每个计算节点上的 slave，负责启动和终止 job，并监控本机的资源。²¹
3. 供最终用户使用的 client 命令行工具，用于提交 job。

根据前面的分析，slave 是个“看门狗进程”，它会启动别的 job 进程，因此必须是个单线程程序。另外它不应该占用太多的 CPU 资源，这也适合单线程模型。master 应该是个模式 2 的多线程程序：

- 它独占一台 8 核的机器，如果用模型 1，等于浪费了 87.5% 的 CPU 资源。
- 整个机群的状态应该能完全放在内存中，这些状态是共享且可变的。如果用模式 3，那么进程之间的状态同步会成问题。而如果大量使用共享内存，则等于是掩耳盗铃，是披着多进程外衣的多线程程序。因为一个进程一旦在临界区内阻塞或 crash，其他进程会全部死锁。
- master 的主要性能指标不是 throughput，而是 latency，即尽快地响应各种事件。它几乎不会出现把 IO 或 CPU 跑满的情况。
- master 监控的事件有优先级区别，一个程序正常运行结束和异常崩溃的处理优先级不同，计算节点的磁盘满了和机箱温度过高这两种报警条件的优先级也不同。如果用单线程，则可能会出现优先级反转。
- 假设 master 和每个 slave 之间用一个 TCP 连接，那么 master 采用 2 个或 4 个 IO 线程来处理 8 个 TCP connections 能有效地降低延迟。
- master 要异步地往本地硬盘写 log，这要求 logging library 有自己的 IO 线程。
- master 有可能要读写数据库，那么数据库连接这个第三方 library 可能有自己的线程，并回调 master 的代码。

²⁰ <https://computing.lln.gov/linux/slurm/>

²¹ slave 的实现要点见 <http://www.slideshare.net/chenshuo/zurg-part-1>。

- master 要服务于多个 clients，用多线程也能降低客户响应时间。也就是说它可以再用 2 个 IO 线程专门处理和 clients 的通信。
- master 还可以提供一个 monitor 接口，用来广播推送（pushing）机群的状态，这样用户不用主动轮询（polling）。这个功能如果用单独的线程来做，会比较容易实现，不会搞乱其他主要功能。
- master 一共开了 10 个线程：
 - ▶ 4 个用于和 slaves 通信的 IO 线程。
 - ▶ 1 个 logging 线程。
 - ▶ 1 个数据库 IO 线程。
 - ▶ 2 个和 clients 通信的 IO 线程。
 - ▶ 1 个主线程，用于做些背景工作，比如 job 调度。
 - ▶ 1 个 pushing 线程，用于主动广播机群的状态。
- 虽然线程数目略多于 core 数目，但是这些线程很多时候都是空闲的，可以依赖 OS 的进程调度来保证可控的延迟。

综上所述，master 用多线程方式编写是自然且高效的。

再举一个 TCP 聊天服务器的例子，这里的“聊天”不完全指人与人聊天，也可能是机器与机器“聊天”。这种服务的特点是并发连接之间有数据交换，从一个连接收到的数据要转发给其他多个连接。因此我们不能按模式 3 的做法，把多个连接分到多个进程中分别处理（这会带来复杂的进程间通信），而只能用模式 1 或者模式 2。如果纯粹只有数据交换，那么我想模式 1 也能工作得很好，因为现在的 CPU 足够快，单线程应付几百个连接不在话下。

如果功能进一步复杂化，加上关键字过滤、黑名单、防灌水等等功能，甚至要给聊天内容自动加上相关连接，每一项功能都会占用 CPU 资源。这时就要考虑模式 2 了，因为单个 CPU 的处理能力显得捉襟见肘，顺序处理导致消息转发的延迟增加。这时我们考虑把空闲的多个 CPU 利用起来，自然的做法是把连接分散到多个线程上，例如按 round-robin 的方式把 1000 个客户连接分配到 4 个 IO 线程上。这样充分利用多核加速。具体的例子见 §6.6 的方案 9，以及 p. 260 的实现。

线程的分类

据我的经验，一个多线程服务程序中的线程大致可分为 3 类：

1. IO 线程，这类线程的主循环是 IO multiplexing，阻塞地等在 select/poll/epoll_wait 系统调用上。这类线程也处理定时事件。当然它的功能不止 IO，有些简单计算也可以放入其中，比如消息的编码或解码。
2. 计算线程，这类线程的主循环是 blocking queue，阻塞地等在 condition variable 上。这类线程一般位于 thread pool 中。这种线程通常不涉及 IO，一般要避免任何阻塞操作。
3. 第三方库所用的线程，比如 logging，又比如 database connection。

服务器程序一般不会频繁地启动和终止线程。甚至，在我写过的程序里，create thread 只在程序启动的时候调用，在服务运行期间是不调用的。

在多核时代，要想充分发挥 CPU 性能，多线程编程是不可避免的，“鸵鸟算法”不是办法。在学会多线程编程之前，我也一直认为单线程服务程序才是王道。在接触多线程编程之后，经过一段时间的训练和适应，我已能比较自如地编写正确且足够高效的多线程程序。学习多线程编程还有一个好处，即训练异步思维，提高分析并发事件的能力。这对设计分布式系统帮助巨大，因为运行在多台机器上的服务进程本质上是异步的。熟悉多线程编程的话，很容易就能发现分布式系统在消息和事件处理方面的 race condition。

3.6 “多线程服务器的适用场合”例释与答疑

《多线程服务器的适用场合》一文在博客²²登出之后，有热心读者提出质疑，我自己也觉得原文没有把道理说通、说透，下面用一些实例来解答读者的疑问。为方便阅读，本节以问答体呈现。以下“连接、端口”均指 TCP 协议。

1. Linux 能同时启动多少个线程？

对于 32-bit Linux，一个进程的地址空间是 4GiB，其中用户态能访问 3GiB 左右，而一个线程的默认栈（stack）大小是 10MB，心算可知，一个进程大约最多能同时启动 300 个线程。如果不改线程的调用栈大小的话，300 左右是上限，因为程序的其他部分（数据段、代码段、堆、动态库等等）同样要占用内存（地址空间）。

对于 64-bit 系统，线程数目可大大增加，具体数字我没有测试过，因为我在实际项目中一台机器上最多只用到过几十个用户线程，其中大部分还是空闲的。

下面的第 2 问关于线程数目的讨论以 32-bit Linux 为例。

²² <http://blog.csdn.net/Solstice/article/details/5334243>

2. 多线程能提高并发度吗？

如果指的是“并发连接数”，则不能。

由问题 1 可知，假如单纯采用 `thread per connection` 的模型，那么并发连接数最多 300，这远远低于基于事件的单线程程序所能轻松达到的并发连接数（几千乃至上万，甚至几万）。所谓“基于事件”，指的是用 `IO multiplexing event loop` 的编程模型，又称 `Reactor` 模式，在前文中已有介绍。

那么采用前文中推荐的 `one loop per thread` 呢？至少不逊于单线程程序。实际上单个 `event loop` 处理 1 万个并发长连接并不罕见，一个 `multi-loop` 的多线程程序应该能轻松支持 5 万并发链接。

小结：`thread per connection` 不适合高并发场合，其 `scalability` 不佳。`one loop per thread` 的并发度足够大，且与 CPU 数目成正比。

3. 多线程能提高吞吐量吗？

对于计算密集型服务，不能。

假设有一个耗时的计算服务，用单线程需要 0.8s。在一台 8 核的机器上，我们可以启动 8 个线程一起对外服务（如果内存够用，启动 8 个进程也一样）。这样完成单个计算仍然要 0.8s，但是由于这些进程的计算可以同时进行，理想情况下吞吐量可以从单线程的 1.25qps（`query per second`）上升到 10qps。（实际情况可能要打个八折——如果不是打对折的话。）

假如改用并行算法，用 8 个核一起算，理论上如果完全并行，加速比高达 8，那么计算时间是 0.1s，吞吐量还是 10qps，但是首次请求的响应时间却降低了很多。实际上根据 `Amdahl's law`，即便算法的并行度高达 95%，8 核的加速比也只有 6，计算时间为 0.133s，这样会造成吞吐量下降为 7.5qps。不过以此为代价，换得响应时间的提升，在有些应用场合也是值得的。

再举一个例子，如果要在 8 核机器上压缩 100 个 1GB 的文本文件，每个 `core` 的处理能力为 200MB/s。那么“每次起 8 个进程，每个进程压缩 1 个文件”与“依次压缩每个文件，每个文件用 8 个线程并行压缩”这两种方式的总耗时相当，因为 CPU 都是满载的。但是第 2 种方式能较快地拿到第一个压缩完的文件，也就是首次响应的延时更小。

这也回答了问题 4。

如果用 thread per request 的模型，每个客户请求用一个线程去处理，那么当并发请求数大于某个临界值 T' 时，吞吐量反而会下降，因为线程多了以后上下文切换的开销也随之增加（分析与数据请见《A Design Framework for Highly Concurrent Systems》²³）。thread per request 是最简单地使用线程的方式，编程最容易，简单地把多线程程序当成一堆串行程序，用同步的方式顺序编程，比如在 Java Servlet 2.x 中，一次页面请求由一个函数

```
HttpServlet.service(HttpServletRequest req, HttpServletResponse resp)
```

同步地完成。

为了在并发请求数很高时也能保持稳定的吞吐量，我们可以用线程池，线程池的大小应该满足“阻抗匹配原则”，见问题 7。

线程池也不是万能的，如果响应一次请求需要做比较多的计算（比如计算的时间占整个 response time 的 1/5 强），那么用线程池是合理的，能简化编程。如果在一次请求响应中，主要时间是在等待 IO，那么为了进一步提高吞吐量，往往要用其他编程模型，比如 Proactor，见问题 8。

4. 多线程能降低响应时间吗？

如果设计合理，充分利用多核资源的话，可以。在突发（burst）请求时效果尤为明显。

例 1：多线程处理输入 以 memcached 服务端为例。memcached 一次请求响应大概可以分为 3 步：

1. 读取并解析客户端输入；
2. 操作 hashtable；
3. 返回客户端。

在单线程模式下，这 3 步是串行执行的。在启用多线程模式时，它会启用多个输入线程（默认是 4 个），并在建立连接时按 round-robin 法把新连接分派给其中一个输入线程，这正好是我说的 one loop per thread 模型。这样一来，第 1 步的操作就能多线程并行，在多核机器上提高多用户的响应速度。第 2 步用了全局锁，还是单线程的，这可算是一个值得继续改进的地方。

比如，有两个用户同时发出了请求，这两个用户的连接正好分配在两个 IO 线程上，那么两个请求的第 1 步操作可以在两个线程上并行执行，然后汇总到第 2 步串行执行，这样总的响应时间比完全串行执行要短一些（在“读取并解析”所占的比重较大的时候，效果更为明显）。请继续看下面这个例子。

²³ by Matt Welsh et al. <http://www.cs.berkeley.edu/~culler/papers/events.pdf>

例 2: 多线程分担负载 假设我们要做一个求解 Sudoku 的服务²⁴, 这个服务程序在 9981 端口接受请求, 输入为一行 81 个数字 (待填数字用 0 表示), 输出为填好之后的 81 个数字 (1~9), 如果无解, 输出 “NO\n”。

由于输入格式很简单, 用单个线程做 IO 就行了。先假设每次求解的计算用时为 10ms, 用前面的方法计算, 单线程程序能达到的吞吐量上限为 100qps; 在 8 核机器上, 如果用线程池来做计算, 能达到的吞吐量上限为 800qps。下面我们看看多线程如何降低响应时间。

假设 1 个用户在极短的时间内发出了 10 个请求, 如果用单线程 “来一个处理一个” 的模型, 这些 reqs 会排在队列里依次处理 (这个队列是操作系统的 TCP 缓冲区, 不是程序里自己的任务队列)。在不考虑网络延迟的情况下, 第 1 个请求的响应时间是 10ms; 第 2 个请求要等第 1 个算完了才能获得 CPU 资源, 它等了 10ms, 算了 10ms, 响应时间是 20ms; 依此类推, 第 10 个请求的响应时间为 100ms; 这 10 个请求的平均响应时间为 55ms。

如果 Sudoku 服务在每个请求到达时开始计时, 会发现每个请求都是 10ms 响应时间; 而从用户的观点来看, 10 个请求的平均响应时间为 55ms, 请读者想想为什么会有这个差异。

下面改用多线程: 1 个 IO 线程, 8 个计算线程 (线程池)。二者之间用 BlockingQueue 沟通。同样是 10 个并发请求, 第 1 个请求被分配到计算线程 1, 第 2 个请求被分配到计算线程 2, 依此类推, 直到第 8 个请求被第 8 个计算线程承担。第 9 和第 10 号请求会等在 BlockingQueue 里, 直到有计算线程回到空闲状态其才能被处理。(请注意, 这里的分配实际上由操作系统来做, 操作系统会从处于 waiting 状态的线程里挑一个, 不一定是 round-robin 的。)这样一来, 前 8 个请求的响应时间差不多都是 10ms, 后 2 个请求属于第二批, 其响应时间大约会是 20ms, 总的平均响应时间是 12ms。可以看出这比单线程快了不少。

由于每道 Sudoku 题目的难度不一, 对于简单的题目, 可能 1ms 就能算出来, 复杂的题目最多用 10ms。那么线程池方案的优势就更明显, 它能有效地降低 “简单任务被复杂任务压住” 的出现概率。

以上举的都是计算密集的例子, 即线程在响应一次请求时不会等待 IO。下面谈谈更复杂的情况。

²⁴ 见笔者的博客《谈谈数独》(<http://blog.csdn.net/Solstice/article/details/2096209>)。

5. 多线程程序如何让 IO 和“计算”相互重叠，降低 latency?

基本思路是，把 IO 操作（通常是写操作）通过 `BlockingQueue` 交给别的线程去做，自己不必等待。

例 1: 日志 (logging) 在多线程服务器程序中，日志 (logging) 至关重要，本例仅考虑写 log file 的情况，不考虑 log server。

在一次请求响应中，可能要写多条日志消息，而如果用同步的方式写文件 (`fprintf` 或 `fwrite`)，多半会降低性能，因为：

- 文件操作一般比较慢，服务线程会等在 IO 上，让 CPU 闲置，增加响应时间。
- 就算有 buffer，还是不灵。多个线程一起写，为了不至于把 buffer 写错乱，往往要加锁。这会让服务线程互相等待，降低并发度。（同时用多个 log 文件不是办法，除非你有多个磁盘，且保证 log files 分散在不同的磁盘上，否则还是要受到磁盘 IO 瓶颈的制约。）

解决办法是单独用一个 logging 线程，负责写磁盘文件，通过一个或多个 `BlockingQueue` 对外提供接口。别的线程要写日志的时候，先把消息（字符串）准备好，然后往 queue 里一塞就行，基本不用等待。这样服务线程的计算就和 logging 线程的磁盘 IO 相互重叠，降低了服务线程的响应时间。

尽管 logging 很重要，但它不是程序的主要逻辑，因此对程序的结构影响越小越好，最好能简单到如同一条 `printf` 语句，且不用担心其他性能开销。而一个好的多线程异步 logging 库能帮我们做到这一点，见第 5 章。（Apache 的 `log4cxx` 和 `log4j` 都支持 `AsyncAppender` 这种异步 logging 方式。）

例 2: memcached 客户端 假设我们用 memcached 来保存用户最后发帖的时间，那么每次响应用户发帖的请求时，程序里要去设置一下 memcached 里的值。这一步如果用同步 IO，会增加延迟。

对于“设置一个值”这样的 write-only idempotent 操作，我们其实不用等 memcached 返回操作结果，这里也不用在乎 set 操作失败，那么可以借助多线程来降低响应延迟。比方说我们可以写一个多线程版的 memcached 的客户端，对于 set 操作，调用方只要把 key 和 value 准备好，调用一下 `asyncSet()` 函数，把数据往 `BlockingQueue` 上一放就能立即返回，延迟很小。剩下的事就留给 memcached 客户端的线程去操心，而服务线程不受阻碍。

其实所有的网络写操作都可以这么异步地做，不过这也有一个缺点，那就是每次 `asyncWrite()` 都要在线程间传递数据。其实如果 TCP 缓冲区是空的，我们就可以在本线程写完，不用劳烦专门的 IO 线程。Netty 就使用了这个办法来进一步降低延迟。

以上都仅讨论了“打一枪就跑”的情况，如果是一问一答，比如从 memcached 取一个值，那么“重叠 IO”并不能降低响应时间，因为你无论如何要等 memcached 的回复。这时我们可以用别的方式来提高并发度，见问题 8。（虽然不能降低响应时间，但也不要浪费线程在空等上。）

以上的例子也说明，`BlockingQueue` 是构建多线程程序的利器。另见 §12.8.3。

6. 为什么第三方库往往要用自己的线程？

event loop 模型没有标准实现。如果自己写代码，尽可以按所用 Reactor 的推荐方式来编程。但是第三方库不一定能很好地适应并融入这个 event loop framework，有时需要用线程来做一些串并转换。比方说检测串口上的数据到达可以用文件描述符的可读事件，因此可以方便地融入 event loop。但是检测串口上的某些控制信号（例如 DCD）只能用轮询（`ioctl(fd, TIOCMGET, &flags)`）或阻塞等待（`ioctl(fd, TIOCMWAIT, TIOCM_CAR)`）；要想融入 event loop，需要单独起一个线程来查询串口信号翻转，再转换为文件描述符的读写事件（可以通过 `pipe(2)`）。

对于 Java，这个问题还好办一些，因为 thread pool 在 Java 里有标准实现，叫 `ExecutorService`。如果第三方库支持线程池，那么它可以和主程序共享一个 `ExecutorService`，而不是自己创建一堆线程。（比如在初始化时传入主程序的 obj。）对于 C++，情况麻烦得多，Reactor 和 thread pool 都没有标准库。

例 1：libmemcached 只支持同步操作 libmemcached 支持所谓的“非阻塞操作”，但没有暴露一个能被 `select/poll/epoll` 的 file describer，它的 `memcached_fetch` 始终会阻塞。它号称 `memcached_set` 可以是非阻塞的，实际意思是不必等待结果返回，但实际上这个函数会阻塞地调用 `write(2)`，仍可能阻塞在网络 IO 上。

如果在我们的 Reactor event handler 里调用了 libmemcached 的函数，那么 latency 就堪忧了。如果想继续用 libmemcached，我们可以为它做一次线程封装，按问题 5 例 2 的办法，同额外的线程专门做 memcached 的 IO，而程序主体还是 Reactor。甚至可以把 memcached 的“数据就绪”作为一个 event，注入我们的 event loop 中，以进一步提高并发度。（例子留待问题 8 讲。）

万幸的是，memcached 的协议非常简单，大不了可以自己写一个基于 Reactor 的客户端，但是数据库客户端就没那么幸运了。

例 2: MySQL 的官方 C API 不支持异步操作 MySQL 的官方客户端²⁵只支持同步操作,对于 UPDATE/INSERT/DELETE 之类只要行为不管结果的操作(如果代码需要得知其执行结果,则另当别论),我们可以用一个单独的线程来做,以降低服务线程的延迟。可仿照前面 memcached_set 的例子,不再赘言。麻烦的是 SELECT,如果要把它也异步化,就得动用更复杂的模式了,见问题 8。

相比之下,PostgreSQL 的 C 客户端 libpq 的设计要好得多,我们可以用 PQsendQuery() 来发起一次查询,然后用标准的 select/poll/epoll 来等待 PQsocket。如果有数据可读,那么用 PQconsumeInput 处理之,并用 PQisBusy 判断查询结果是否已就绪。最后用 PQgetResult 来获取结果。借助这套异步 API,我们可以很容易地为 libpq 写一套 wrapper,使之融入程序所用的 event loop 模型中。

7. 什么是线程池大小的阻抗匹配原则?

我在前文中提到“阻抗匹配原则”,这里大致讲一讲。

如果池中线程在执行任务时,密集计算所占的时间比重为 P ($0 < P \leq 1$),而系统一共有 C 个 CPU,为了让这 C 个 CPU 跑满而又不过载,线程池大小的经验公式 $T = C/P$ 。 T 是个 hint,考虑到 P 值的估计不是很准确, T 的最佳值可以上下浮动 50%。这个经验公式的原理很简单, T 个线程,每个线程占用 P 的 CPU 时间,如果刚好占满 C 个 CPU,那么必有 $T \times P = C$ 。下面验证一下边界条件的正确性。

假设 $C = 8$, $P = 1.0$,线程池的任务完全是密集计算,那么 $T = 8$ 。只要 8 个活动线程就能让 8 个 CPU 饱和,再多也没用,因为 CPU 资源已经耗光了。

假设 $C = 8$, $P = 0.5$,线程池的任务有一半是计算,有一半等在 IO 上,那么 $T = 16$ 。考虑操作系统能灵活、合理地调度 sleeping/writing/running 线程,那么大概 16 个“50% 繁忙的线程”能让 8 个 CPU 忙个不停。启动更多的线程并不能提高吞吐量,反而因为增加上下文切换的开销而降低性能。

如果 $P < 0.2$,这个公式就不适用了, T 可以取一个固定值,比如 $5 \times C$ 。另外,公式里的 C 不一定是 CPU 总数,可以是“分配给这项任务的 CPU 数目”,比如在 8 核机器上分出 4 个核来做一项任务,那么 $C = 4$ 。

8. 除了你推荐的 Reactor + thread poll, 还有别的 non-trivial 多线程编程模型吗?

有, Proactor。如果一次请求响应中要和别的进程打多次交道,那么 Proactor 模型往往能做到更高的并发度。当然,代价是代码变得支离破碎,难以理解。

²⁵ 非官方的 libdrizzle 似乎支持异步操作,见 <https://github.com/chaoslawful/drizzle-nginx-module>。

这里举 HTTP proxy 为例，一次 HTTP proxy 的请求如果没有命中本地 cache，那么它多半会：

1. 解析域名（不要小看这一步，对于一个陌生的域名，解析可能要花几秒的时间）；
2. 建立连接；
3. 发送 HTTP 请求；
4. 等待对方回应；
5. 把结果返回给客户。

这 5 步中跟 2 个 server 发生了 3 次 round-trip，每次都可能花几百毫秒：

1. 向 DNS 问域名，等待回复；
2. 向对方的 HTTP 服务器发起连接，等待 TCP 三路握手完成；
3. 向对方发送 HTTP request，等待对方 response。

而实际上 HTTP proxy 本身的运算量不大，如果用线程池，池中线程的数目会很庞大，不利于操作系统的管理调度。

这时我们有两个解决思路：

1. 把“域名已解析”、“连接已建立”、“对方已完成响应”做成 event，继续按照 Reactor 的方式来编程。这样一来，每次客户请求就不能用一个函数从头到尾执行完成，而要分成多个阶段，并且要管理好请求的状态（“目前到了第几步？”）。
 2. 用回调函数，让系统来把任务串起来。比如收到用户请求，如果没有命中本地缓存，那么需要执行：
 - a. 立刻发起异步的 DNS 解析 `startDNSResolve()`，告诉系统在解析完之后调用 `DNSResolved()` 函数；
 - b. 在 `DNSResolved()` 中，发起 TCP 连接请求，告诉系统在连接建立之后调用 `connectionEstablished()`；
 - c. 在 `connectionEstablished()` 中发送 HTTP request，告诉系统在收到响应之后调用 `httpResponded()`；
 - d. 最后，在 `httpResponded()` 里把结果返回给客户。
- .NET 大量采用的 `BeginInvoke/EndInvoke` 操作也是这个编程模式。当然，对于不熟悉这种编程方式的人，代码会显得很难看。有关 Proactor 模式的例子可参看 Boost.Asio 的文档，这里不再多说。

Proactor 模式依赖操作系统或库来高效地调度这些子任务，每个子任务都不会阻塞，因此能用比较少的线程达到很高的 IO 并发度。

Proactor 能提高吞吐，但不能降低延迟，所以我没有深入研究。另外，在没有语言直接支持的情况下²⁶，Proactor 模式让代码非常破碎，在 C++ 中使用 Proactor 是很痛苦的。因此最好在“线程”很廉价的语言中使用这种方式，这时 runtime 往往会屏蔽细节，程序用单线程阻塞 IO 的方式来处理 TCP 连接。

9. 模式 2 和模式 3a 该如何取舍？

§3.5 中提到，模式 2 是一个多线程的进程，模式 3a 是多个相同的单线程进程。

我认为，在其他条件相同的情况下，可以根据工作集（work set）的大小来取舍。工作集是指服务程序响应一次请求所访问的内存大小。

如果工作集较大，那么就用多线程，避免 CPU cache 换入换出，影响性能；否则，就用单线程多进程，享受单线程编程的便利。举例来说

- 如果程序有一个较大的本地 cache，用于缓存一些基础参考数据（in-memory look-up table），几乎每次请求都会访问 cache，那么多线程更适合一些，因为可以避免每个进程都自己保留一份 cache，增加内存使用。
- memcached 这个内存消耗大户用多线程服务端就比在同一台机器上运行多个 memcached instance 要好。（但是如果你在 16GiB 内存的机器上运行 32-bit memcached，那么此时多 instance 是必需的。）
- 求解 Sudoku 用不了多大内存。如果单线程编程更方便的话，可以用单线程多进程来做。再在前面加一个单线程的 load balancer，仿 lighttpd + fastcgi 的成例。

线程不能减少工作量，即不能减少 CPU 时间。如果解决一个问题需要执行一亿条指令（这个数字不大，不要被吓到），那么用多线程只会让这个数字增加。但是通过合理调配这一亿条指令在多个核上的执行情况，我们能让工期提早结束。这听上去像统筹方法，其实也确实是统筹方法。

²⁶ 有的语言能通过库扩展，例如 <http://jscex.info/zh-cn/>。

第 4 章

C++ 多线程系统编程精要

学习多线程编程面临的最大的思维方式的转变有两点：

- 当前线程可能随时会被切换出去，或者说被抢占（preempt）了。
- 多线程程序中事件的发生顺序不再有全局统一的先后关系¹。

当线程被切换回来继续执行下一条语句（指令）的时候，全局数据（包括当前进程在操作系统内核中的状态）可能已经被其他线程修改了。例如，在没有为指针 `p` 加锁的情况下，`if (p && p->next) { /* ... */ }` 有可能导致 segfault，因为在逻辑与（&&）的前一个分支 evaluate 为 true 之后的一刹那，`p` 可能被其他线程置为 NULL 或是被释放，后一个分支就访问了非法地址。

在单 CPU 系统中，理论上我们可以通过记录 CPU 上执行的指令的先后顺序来推演多线程的实际交织（interweaving）运行的情况。在多核系统中，多个线程是并行执行的，我们甚至没有统一的全局时钟来为每个事件编号。在没有适当同步的情况下，多个 CPU 上运行的多个线程中的事件发生先后顺序是无法确定的²。在引入适当同步后，事件之间才有了 happens-before 关系³。

多线程程序的正确性不能依赖于任何一个线程的执行速度，不能通过原地等待（sleep()）来假定其他线程的事件已经发生，而必须通过适当的同步来让当前线程能看到其他线程的事件的结果。无论线程执行得快与慢（被操作系统切换出去得越多，执行越慢），程序都应该能正常工作。例如下面这段代码就有这方面的问题。

¹ 这意味着时空观的转变，从牛顿的绝对时空观转变为爱因斯坦的相对论时空观，分布式系统也面临类似的思维方式转变，见第 9 章。

² 在多 CPU 机器上，假设主板上两个物理 CPU 的距离为 15cm，CPU 主频是 2.4GHz，电信号在电路中的传播速度按 $2 \times 10^8 \text{m/s}$ 估算，那么在 1 个时钟周期（0.42ns）之内，电信号不能从一个 CPU 到达另一个 CPU。因此对于每个 CPU 自己这个观察者来说，它看到的事件发生的顺序没有全局一致性。

³ Leslie Lamport: 《Time, Clocks and the Ordering of Events in a Distributed System》
(<http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf>)。

```
bool running = false; // 全局标志

void threadFunc()
{
    while (running)
    {
        // get task from queue
    }
}

void start()
{
    muduo::Thread t(threadFunc);
    t.start();
    running = true; // 应该放到 t.start() 之前。
}
```

这段代码暗中假定线程函数的启动慢于 `running` 变量的赋值⁴，因此线程函数能进入 `while` 循环执行我们想要的功能。如果上机测试运行这段代码，十有八九会按我们预期的那样工作。但是，直到有一天，系统负载很高，`Thread::start()` 调用 `pthread_create()` 陷入内核后返回时，内核决定换另外一个就绪任务来执行。于是 `running` 的赋值就推迟了，这时线程函数就可能不进入 `while` 循环而直接退出了。

或许有人会认为在 `while` 之前加一小段延时（`sleep`）就能解决问题，但这是错的，无论加多大的延时，系统都有可能先执行 `while` 的条件判断，然后再执行 `running` 的赋值。正确的做法是把 `running` 的赋值放到 `t.start()` 之前，这样借助 `pthread_create()` 的 `happens-before` 语意来保证 `running` 的新值能被线程看到。

4.1 基本线程原语的选用

我认为用 C/C++ 编写跨平台⁵的多线程程序不是普遍的需求，因此本书只谈现代 Linux⁶ 下的多线程编程。POSIX threads 的函数有 110 多个，真正常用的不过十几个。而且在 C++ 程序中通常会有更为易用的 `wrapper`，不会直接调用 Pthreads 函数。这 11 个最基本的 Pthreads 函数是：

2 个：线程的创建和等待结束（`join`）。封装为 `muduo::Thread`。

4 个：`mutex` 的创建、销毁、加锁、解锁。封装为 `muduo::MutexLock`。

5 个：条件变量的创建、销毁、等待、通知、广播。封装为 `muduo::Condition`。

⁴ 严格来说，全局 `running` 的赋值和读取应该用 `mutex` 或者 `memory barrier`，但不影响这里的讨论。

⁵ 包括只针对 POSIX 操作系统（Linux、Solaris、FreeBSD 等等）的跨平台。

⁶ 2004 年 Linux 2.6 内核发布之后，NPTL 线程库。

这些封装 class 都很直截了当，加起来也就一两百行代码，却已经构成了多线程编程的全部必备原语。用这三样东西（thread、mutex、condition）可以完成任何多线程编程任务。当然我们一般也不会直接使用它们（mutex 除外），而是使用更高层的封装，例如 `mutex::ThreadPool` 和 `mutex::CountDownLatch` 等，见第 2 章。

除此之外，Pthreads 还提供了其他一些原语，有些是可以酌情使用的，有些则是不推荐使用的。可以酌情使用的有：

- `pthread_once`，封装为 `muduo::Singleton<T>`。其实不如直接用全局变量。
- `pthread_key*`，封装为 `muduo::ThreadLocal<T>`。可以考虑用 `__thread` 替换之。

不建议使用：

- `pthread_rwlock`，读写锁通常应慎用。`muduo` 没有封装读写锁，这是有意的。
- `sem_*`，避免用信号量（semaphore）。它的功能与条件变量重合，但容易用错。
- `pthread_[cancel, kill]`。程序中出现了它们，则通常意味着设计出了问题。

不推荐使用读写锁的原因是它往往造成提高性能的错觉（允许多个线程并发读），实际上在很多情况下，与使用最简单的 `mutex` 相比，它实际上降低了性能。另外，写操作会阻塞读操作，如果要求优化读操作的延迟，用读写锁是不合适的。

多线程系统编程的难点不在于学习线程原语（primitives），而在于理解多线程与现有的 C/C++ 库函数和系统调用的交互关系，以进一步学习如何设计并实现线程安全且高效的程序。

4.2 C/C++ 系统库的线程安全性

现行的 C/C++ 标准（C89/C99/C++03）并没有涉及线程，新版的 C/C++ 标准（C11 和 C++11）规定了程序在多线程下的语意，C++11 还定义了一个线程库（`std::thread`）。

对于标准而言，关键的不是定义线程库，而是规定内存模型（memory model）。特别是规定一个线程对某个共享变量的修改何时能被其他线程看到，这称为内存序（memory ordering）或者内存能见度（memory visibility）。从理论上讲，如果没有合适的内存模型，编写正确的多线程程序属于撞大运行为，见 Hans-J. Boehm 的论文《Threads Cannot be Implemented as a Library》⁷。不过我认为不必担心这篇文章提到的问题，标准的滞后不会对实践构成影响。因为从操作系统开始支持多线程到现在

⁷ <http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf>

已经过去了近 20 年，人们已经编写了不计其数的运行于关键生产环境的多线程程序，甚至 Linux 操作系统内核本身也可以是抢占的（preemptive）。因此可以认为每个支持多线程的操作系统上自带的 C/C++ 编译器对本平台的多线程支持都足够好。现在多线程程序工作不正常很难归结于编译器 bug，毕竟 POSIX threads 线程标准在 20 世纪 90 年代中期就制定了。当然，新标准的积极意义在于让编写跨平台的多线程程序更有保障了。

Unix 系统库（libc 和系统调用）的接口风格是在 20 世纪 70 年代早期确立的，而第一个支持用户态线程的 Unix 操作系统出现在 20 世纪 90 年代早期。线程的出现立刻给系统函数库带来了冲击，破坏了 20 年来一贯的编程传统和假定。例如：

- `errno` 不再是一个全局变量，因为每个线程可能会执行不同的系统库函数。
- 有些“纯函数”不受影响，例如 `memset/strcpy/sprintf` 等等。
- 有些影响全局状态或者有副作用的函数可以通过加锁来实现线程安全，例如 `malloc/free`、`printf`、`fread/fseek` 等等。
- 有些返回或使用静态空间的函数不可能做到线程安全，因此要提供另外的版本，例如 `asctime_r/ctime_r/gmtime_r/stderror_r/strtok_r` 等等。
- 传统的 `fork()` 并发模型不再适用于多线程程序（§4.9）。

现在 Linux glibc 把 `errno` 定义为一个宏，注意 `errno` 是一个 lvalue，因此不能简单定义为某个函数的返回值，而必须定义为对函数返回指针的 dereference。

```
extern int *__errno_location(void);  
#define errno (*__errno_location())
```

值得一提的是，操作系统支持多线程已有近 20 年，早先一些性能方面的缺陷都基本被弥补了。例如最早的 SGI STL 自己定制了内存分配器，而现在 g++ 自带的 STL 已经直接使用 `malloc` 来分配内存，`std::allocator` 已经变成了鸡肋（§12.2）。原先 Google tcmalloc 相对于 glibc 2.3 中的 `ptmalloc2` 有很大的性能提升，现在最新的 glibc 中的 `ptmalloc3` 已经把差距大大缩小了。

我们不必担心系统调用的线程安全性，因为系统调用对于用户态程序来说是原子的。但是要注意系统调用对于内核状态的改变可能影响其他线程，这个话题留到 §4.6 再细说。

与直觉相反，POSIX 标准列出的是一份非线程安全的函数的黑名单⁸，而不是一份线程安全的函数的白名单（All functions defined by this volume of POSIX.1-2008 shall be thread-safe, except that the following functions need not be thread-safe）。在这份黑名单中，`system`、`getenv/putenv/setenv` 等等函数都是不安全的。

⁸ http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_09

因此，可以说现在 glibc 库函数大部分都是线程安全的。特别是 FILE* 系列函数是安全的，glibc 甚至提供了非线程安全的版本⁹ 以应对某些特殊场合的性能需求。尽管单个函数是线程安全的，但两个或多个函数放到一起就不再安全了。例如 fseek() 和 fread() 都是安全的，但是对某个文件“先 seek 再 read”这两步操作中间有可能会被打断，其他线程有可能趁机修改了文件的当前位置，让程序逻辑无法正确执行。在这种情况下，我们可以用 flockfile(FILE*) 和 funlockfile(FILE*) 函数来显式地加锁。并且由于 FILE* 的锁是可重入的，加锁之后再调用 fread() 不会造成死锁。

如果程序直接使用 lseek(2) 和 read(2) 这两个系统调用来随机读取文件，也存在“先 seek 再 read”这种 race condition，但是似乎我们无法高效地对系统调用加锁。解决办法是改用 pread(2) 系统调用，它不会改变文件的当前位置。

由此可见，编写线程安全程序的一个难点在于线程安全是不可组合的 (composable)¹⁰，一个函数 foo() 调用了两个线程安全的函数，而这个 foo() 函数本身很可能不是线程安全的。即便现在大多数 glibc 库函数是线程安全的，我们也不能像写单线程程序那样编写代码。例如，在单线程程序中，如果我们要临时转换时区，可以用 tzset() 函数，这个函数会改变程序全局的“当前时区”。

```
// 获取伦敦的当前时间
string oldTz = getenv("TZ");           // save TZ, assumeing non-NULL
putenv("TZ=Europe/London");           // set TZ to London
tzset();                               // load London time zone

struct tm localTimeInLN;
time_t now = time(NULL);               // get time in UTC
localtime_r(&now, &localTimeInLN);    // convert to London local time
setenv("TZ", oldTz.c_str(), 1);       // restore old TZ
tzset();                               // local old time zone
```

但是在多线程程序中，这么做不是线程安全的，即便 tzset() 本身是线程安全的。因为它改变了全局状态（当前时区），这有可能影响其他线程转换当前时间，或者被其他进行类似操作的线程影响。解决办法是使用 muduo::TimeZone class，每个 immutable instance 对应一个时区，这样时间转换就不需要修改全局状态了。例如：

```
class TimeZone
{
public:
    explicit TimeZone(const char* zonefile);

    struct tm toLocalTime(time_t secondsSinceEpoch) const;
    time_t fromLocalTime(const struct tm&) const;
```

⁹ fread_unlocked、fwrite_unlocked 等等，见 man unlocked_stdio。

¹⁰ 就跟 C++ 异常安全也是不可组合的一样。


```

    // default copy ctor/assignment/dtor are okay.
    // ...
};

const TimeZone kNewYorkTz("/usr/share/zoneinfo/America/New_York");
const TimeZone kLondonTz("/usr/share/zoneinfo/Europe/London");

time_t now = time(NULL);
struct tm localTimeInNY = kNewYorkTz.toLocalTime(now);
struct tm localTimeInLN = kLondonTz.toLocalTime(now);

```

对于 C/C++ 库的作者来说，如何设计线程安全的接口也成了一大考验，值得仿效的例子并不多。一个基本思路是尽量把 class 设计成 `immutable` 的，这样用起来就不必为线程安全操心了。

尽管 C++03 标准没有明说标准库的线程安全性，但我们可以遵循一个基本原则：凡是非共享的对象都是彼此独立的，如果一个对象从始至终只被一个线程用到，那么它就是安全的。另外一个事实标准是：共享的对象的 `read-only` 操作是安全的¹¹，前提是不能有并发的写操作。例如两个线程各自访问自己的局部 `vector` 对象是安全的；同时访问共享的 `const vector` 对象也是安全的，但是这个 `vector` 不能被第三个线程修改。一旦有 `writer`，那么 `read-only` 操作也必须加锁，例如 `vector::size()`。

根据 §1.1.1 对线程安全的定义，C++ 的标准库容器和 `std::string` 都不是线程安全的，只有 `std::allocator` 保证是线程安全的。一方面的原因是为了避免不必要的性能开销，另一方面的原因是单个成员函数的线程安全并不具备可组合性（`composable`）。假设有 `safe_vector<T>` class，它的接口与 `std::vector` 相同，不过每个成员函数都是线程安全的（类似 Java `synchronized` 方法）。但是用 `safe_vector<T>` 并不一定能写出线程安全的代码。例如：

```

safe_vector<int> vec;    // 全局可见

if (!vec.empty())        // 没有加锁保护
{
    int x = vec[0];       // 这两步在多线程下是不安全的
}

```

在 `if` 语句判断 `vec` 非空之后，别的线程可能清空其元素，从而造成 `vec[0]` 失效。

C++ 标准库中的绝大多数泛型算法是线程安全的¹²，因为这些都是无状态纯函数。只要输入区间是线程安全的，那么泛型函数就是线程安全的。

¹¹ 这意味着标准库容器不能采用自调整（`self-adjusting`）的数据结构，比如 `splay tree`，这种数据结构在 `read` 的时候也会修改状态，见 <http://www.cs.au.dk/~gerth/aa11/slides/selfadjusting.pdf>。

¹² `std::random_shuffle()` 可能是个例外，它用到了随机数发生器。

C++ 的 `iostream` 不是线程安全的，因为流式输出

```
std::cout << "Now is " << time(NULL);
```

等价于两个函数调用

```
std::cout.operator<<("Now is ")  
    .operator<<(time(NULL));
```

即便 `ostream::operator<<()` 做到了线程安全，也不能保证其他线程不会在两次函数调用之前向 `stdout` 输出其他字符。

对于“线程安全的 `stdout` 输出”这个需求，我们可以改用 `printf`，以达到安全性和输出的原子性。但是这等于用了全局锁，任何时刻只能有一个线程调用 `printf`，恐怕不见得高效。在多线程程序中高效的日志需要特殊设计，见第 5 章。

4.3 Linux 上的线程标识

POSIX threads 库提供了 `pthread_self` 函数用于返回当前进程的标识符，其类型为 `pthread_t`。`pthread_t` 不一定是一个数值类型（整数或指针），也有可能是一个结构体，因此 Pthreads 专门提供了 `pthread_equal` 函数用于对比两个线程标识符是否相等。这就带来一系列问题，包括：

- 无法打印输出 `pthread_t`，因为不知道其确切类型。也就没法在日志中用它表示当前线程的 id。
- 无法比较 `pthread_t` 的大小或计算其 hash 值，因此无法用作关联容器的 key。
- 无法定义一个非法的 `pthread_t` 值，用来表示绝对不可能存在的线程 id，因此 `MutexLock` class 没有办法有效判断当前线程是否已经持有本锁。
- `pthread_t` 值只在进程内有意义，与操作系统的任务调度之间无法建立有效关联。比方说在 `/proc` 文件系统中找不到 `pthread_t` 对应的 task。

另外，glibc 的 Pthreads 实现实际上把 `pthread_t` 用作一个结构体指针（它的类型是 `unsigned long`），指向一块动态分配的内存，而且这块内存是反复使用的。这就造成 `pthread_t` 的值很容易重复。Pthreads 只保证同一进程之内，同一时刻的各个线程的 id 不同；不能保证同一进程先后多个线程具有不同的 id，更不要说一台机器上多个进程之间的 id 唯一性了。

例如下面这段代码中先后两个线程的标识符是相同的：

```
int main()
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, threadFunc, NULL);
    printf("%lx\n", t1);
    pthread_join(t1, NULL);

    pthread_create(&t2, NULL, threadFunc, NULL);
    printf("%lx\n", t2);
    pthread_join(t2, NULL);
}
```

一次运行结果如下：

```
$ ./a.out
7fad11787700
7fad11787700
```

因此，`pthread_t` 并不适合用作程序中对线程的标识符。

在 Linux 上，我建议使用 `gettid(2)` 系统调用的返回值作为线程 id，这么做的好处有：

- 它的类型是 `pid_t`，其值通常是一个小整数¹³，便于在日志中输出。
- 在现代 Linux 中，它直接表示内核的任务调度 id，因此在 `/proc` 文件系统中可以轻易找到对应项：`/proc/tid` 或 `/proc/pid/task/tid`。
- 在其他系统工具中也容易定位到具体某一个线程，例如在 `top(1)` 中我们可以按线程列出任务，然后找出 CPU 使用率最高的线程 id，再根据程序日志判断到底哪一个线程在耗用 CPU。
- 任何时刻都是全局唯一的，并且由于 Linux 分配新 `pid` 采用递增轮回办法，短时间内启动的多个线程也会具有不同的线程 id。
- 0 是非法值，因为操作系统第一个进程 `init` 的 `pid` 是 1。

但是 `glibc` 并没有封装这个系统调用，需要我们自己实现。封装 `gettid(2)` 很简单，但是每次都执行一次系统调用似乎有些浪费，如何才能做到更高效呢？

`muduo::CurrentThread::tid()` 采取的办法是用 `__thread` 变量来缓存 `gettid(2)` 的返回值，这样只有在本线程第一次调用的时候才进行系统调用，以后都是直接从 `thread local` 缓存的线程 id 拿到结果¹⁴，效率无忧。多线程程序在打日志的时候可以

¹³ 最大值是 `/proc/sys/kernel/pid_max`，默认情况下是 32768。

¹⁴ 这个做法是受了 `glibc` 封装 `getpid()` 的启发。

在每一条日志消息中包含当前线程的 id，不必担心有效率损失。读者有兴趣的话可以对比一下 `boost::this_thread::get_id()` 的实现效率。

还有一个小问题，万一程序执行了 `fork(2)`，那么子进程会不会看到 stale 的缓存结果呢？解决办法是用 `pthread_atfork()` 注册一个回调，用于清空缓存的线程 id。具体代码见 `muduo/base/CurrentThread.h` 和 `Thread.cc`。

4.4 线程的创建与销毁的守则

线程的创建和销毁是编写多线程程序的基本要素，线程的创建比销毁要容易得多，只需要遵循几条简单的原则：

- 程序库不应该在未提前告知的情况下创建自己的“背景线程”。
- 尽量用相同的方式创建线程，例如 `muduo::Thread`。
- 在进入 `main()` 函数之前不应该启动线程。
- 程序中线程的创建最好能在初始化阶段全部完成。

以下分别谈一谈这几个观点。

线程是稀缺资源，一个进程可以创建的并发线程数目受限于地址空间的大小和内核参数，一台机器可以同时并行运行的线程数目受限于 CPU 的数目。因此我们在设计一个服务端程序的时候要精心规划线程的数目，特别是根据机器的 CPU 数目来设置工作线程的数目，并为关键任务保留足够的计算资源。如果程序库在背地里使用了额外的线程来执行任务，我们这种资源规划就漏算了。可能会导致高估系统的可用资源，结果处理关键任务不及时，达不到预设的性能指标。

还有一个重要原因是，一旦程序中有不止一个线程，就很难安全地 `fork()` 了 (§4.9)。因此“库”不能偷偷创建线程。如果确实有必要使用背景线程，至少应该让使用者知道。另外，如果有可能，可以让使用者在初始化库的时候传入线程池或 event loop 对象，这样程序可以统筹线程的数目和用途，避免低优先级的任务独占某个线程。

理想情况下，程序里的线程都是用同一个 class 创建的（`muduo::Thread`），这样容易在线程的启动和销毁阶段做一些统一的簿记（bookkeeping）工作。比如说调用一次 `muduo::CurrentThread::tid()` 把当前线程 id 缓存起来，以后再取线程 id 就不会陷入内核了。也可以统计当前有多少活动线程¹⁵，进程一共创建了多少线程，每个

¹⁵ 线程数目可以从 `/proc/pid/status` 拿到。

线程的用途分别是什么。C/C++ 的线程不像 Java 线程那样有名字，但是我们可以通过 Thread class 实现类似的效果。如果每个线程都是通过 muduo::Thread 启动的，这些都不难做到。必要的话可以写一个 ThreadManager singleton class，用它来记录当前活动线程，可以方便调试与监控。

但是这不是总能做到的，有些第三方库（C 语言库）会自己启动线程，这样的“野生”线程就没有纳入全局的 ThreadManager 管理之中。muduo::CurrentThread::tid() 必须要考虑被这种“野生”线程调用的可能，因此它必须每次都检查缓存的线程 id 是否有效，而不能假定在线程启动阶段已经缓存好了 id，直接返回缓存值就行了。如果库提供异步回调，一定要明确说明会在哪个（哪些）线程调用用户提供的回调函数，这样用户可以知道在回调函数中能不能执行耗时的操作，会不会阻塞其他任务的执行。

在 main() 函数之前不应该启动线程，因为这会影响全局对象的安全构造。我们知道，C++ 保证在进入 main() 之前完成全局对象¹⁶的构造。同时，各个编译单元之间的对象构造顺序是不确定的，我们也有一些办法来影响初始化顺序，保证在初始化某个全局对象时使用到的其他全局对象都是构造完成的。但无论如何这些全局对象的构造是依次进行的，都在主线程中完成，无须考虑并发与线程安全。如果其中一个全局对象创建了线程，那就危险了。因为这破坏了初始化全局对象的基本假设。万一将来代码改动之后造成该线程访问了未经初始化的全局对象，那么这种隐晦错误查起来就很费劲了。或许你想用锁来保证全局对象初始化完成，但是怎么保证这个全局的锁对象的构造能在线程启动之前完成呢？因此，全局对象不能创建线程。如果一个库需要创建线程，那么应该进入 main() 函数之后再调用库的初始化函数去做。

不要为了每个计算任务，每次请求去创建线程。一般也不会为每个网络连接创建线程，除非并发连接数与 CPU 数相近。一个服务程序的线程数目应该与当前负载无关，而应该与机器的 CPU 数目有关，即 load average 有比较小（最好不大于 CPU 数目）的上限。这样尽量避免出现 thrashing，不会因为负载急剧增加而导致机器失去正常响应。这么做的重要原因是，在机器失去响应期间，我们无法探查它究竟在做什么，也没办法立刻终止有问题的进程，防止损害进一步扩大。如果有实时性方面的要求，线程数目不应该超过 CPU 数目，这样可以基本保证新任务总能及时得到执行，因为总有 CPU 是空闲的。最好在程序的初始化阶段创建全部工作线程，在程序运行期间不再创建或销毁线程。借助 muduo::ThreadPool 和 muduo::EventLoop，我们很容易就能把计算任务和 IO 任务分配到已有的线程，代价只有新建线程的几分之一。

¹⁶ 本章所指的“全局对象”也包括 namespace 级全局对象、文件级静态对象、class 的静态对象，但不包括函数内的静态对象。

线程的销毁有几种方式¹⁷：

- 自然死亡。从线程主函数返回，线程正常退出。
- 非正常死亡。从线程主函数抛出异常或线程触发 `segfault` 信号等非法操作¹⁸。
- 自杀。在线程中调用 `pthread_exit()` 来立刻退出线程。
- 他杀。其他线程调用 `pthread_cancel()` 来强制终止某个线程。

`pthread_kill()` 是往线程发信号，留到 §4.10 再讨论。

线程正常退出的方式只有一种，即自然死亡。任何从外部强行终止线程的做法和想法都是错的^{19 20}。佐证有：Java 的 `Thread` class 把 `stop()`、`suspend()`、`destroy()` 等函数都废弃（`deprecated`）了，Boost.Threads 根本就不提供 `thread::cancel()` 成员函数²¹。因为强行终止线程的话（无论是自杀还是他杀），它没有机会清理资源。也没有机会释放已经持有的锁，其他线程如果再想对同一个 `mutex` 加锁，那么就会立刻死锁。因此我认为不用去研究 `cancellation point` 这种“鸡肋”概念（见下一页）。

如果确实需要强行终止一个耗时很长的计算任务，而又不想在计算期间周期性地检查某个全局退出标志，那么可以考虑把那一部分代码 `fork()` 为新的进程，这样杀（`kill(2)`）一个进程比杀本进程内的线程要安全得多。当然，`fork()` 的新进程与本进程的通信方式也要慎重选取，最好用文件描述符（`pipe(2)`/`socketpair(2)`/`TCP socket`）来收发数据，而不要用共享内存和跨进程的互斥器等 IPC，因为这样仍然有死锁的可能。

`muduo::Thread` 不是传统意义上的 RAII class，因为它析构的时候没有销毁持有的 Pthreads 线程句柄（`pthread_t`），也就是说 `Thread` 的析构不会等待线程结束。一般而言，我们会让 `Thread` 对象的生命期长于线程，然后通过 `Thread::join()` 来等待线程结束并释放线程资源。如果 `Thread` 对象的生命期短于线程，那么就没有机会释放 `pthread_t` 了。`muduo::Thread` 没有提供 `detach()` 成员函数，因为我不认为这是必要的。

最后，我认为如果能做到前面提到的“程序中线程的创建最好能在初始化阶段全部完成”，则线程是不必销毁的，伴随进程一直运行，彻底避开了线程安全退出可能面临的各种困难，包括 `Thread` 对象生命期管理、资源释放等等。

¹⁷ http://blog.csdn.net/program_think/article/details/3991107

¹⁸ 通常伴随进程死亡。如果程序中的某个线程意外终止，我不认为让进程继续带伤运行下去有何必要。

¹⁹ <http://www.cppblog.com/lymons/archive/2008/12/19/69810.html>

²⁰ <http://www.cppblog.com/lymons/archive/2008/12/25/70227.html>

²¹ http://www.boost.org/doc/libs/1_34_0/doc/html/thread/faq.html

4.4.1 pthread_cancel 与 C++

POSIX threads 有 cancellation point 这个概念，意思是线程执行到这里有可能会被终止（cancel）（如果别的线程对它调用了 pthread_cancel() 的话）。POSIX 标准列出了必须或者可能是 cancellation point 的函数^{22 23}。

在 C++ 中，cancellation point 的实现与 C 语言有所不同，线程不是执行到此函数就立刻终止，而是该函数会抛出异常。这样可以有机会执行 stack unwind，析构栈上对象（特别是释放持有的锁）。如果一定要使用 cancellation point，建议读一读 Ulrich Drepper 写的 Cancellation and C++ Exceptions 这篇短文²⁴。不过按我的观点，不应该从外部杀死线程。

4.4.2 exit(3) 在 C++ 中不是线程安全的

exit(3) 函数在 C++ 中的作用除了终止进程，还会析构全局对象和已经构造完的函数静态对象。这有潜在的死锁可能，考虑下面这个例子。

```
void someFunctionMayCallExit()
{
    exit(1);
}

class GlobalObject // : boost::noncopyable
{
public:
    void doit()
    {
        MutexLockGuard lock(mutex_);
        someFunctionMayCallExit();
    }

    ~GlobalObject()
    {
        printf("GlobalObject::~GlobalObject\n");
        MutexLockGuard lock(mutex_); // 此处发生死锁
        // clean up
        printf("GlobalObject::~GlobalObject cleaning\n");
    }

private:
    MutexLock mutex_;
};
```

²² <http://stackoverflow.com/questions/433989/posix-cancellation-points>

²³ http://pubs.opengroup.org/onlinepubs/000095399/functions/xsh_chap02_09.html#tag_02_09_05_02

²⁴ <http://udrepper.livejournal.com/21541.html>

```
GlobalObject g_obj;

int main()
{
    g_obj.doit();
}
```

GlobalObject::doit() 函数辗转调用了 exit(), 从而触发了全局对象 g_obj 的析构。GlobalObject 的析构函数会试图加锁 mutex_, 而此时 mutex_ 已经被 GlobalObject::doit() 锁住了, 于是造成了死锁。

再举一个调用纯虚函数导致程序崩溃的例子。假如有一个策略基类, 在运行时我们会根据情况使用不同的无状态策略 (派生类对象)。由于策略是无状态的, 因此可以共享派生类对象, 不必每次都新建。这里以日历 (Calendar) 基类和不同国家的假期 (AmericanCalendar 和 BritishCalendar) 为例, factory 函数返回某个全局对象的引用, 而不是每次都创建新的派生类对象。

```
class Calendar : boost::noncopyable
{
public:
    virtual bool isHoliday(muduo::Date d) const = 0; // 纯虚函数
    virtual ~Calendar() {}
};

class AmericanCalendar : public Calendar
{
public:
    virtual bool isHoliday(muduo::Date d) const;
};

class BritishCalendar : public Calendar
{
public:
    virtual bool isHoliday(muduo::Date d) const;
};

AmericanCalendar americanCalendar; // 全局对象
BritishCalendar britishCalendar;

// factory method returns americanCalendar or britishCalendar
Calendar& getCalendar(const string& region);
```

通常的使用方式是通过 factory 拿到具体国家的日历, 再判断某一天是不是假期:

```
void processRequest(const Request& req)
{
    Calendar& calendar = getCalendar(req.region);
    // 如果别的线程在此时调用了 exit() .....
    if (calendar.isHoliday(req.settlement_date))
```



```
{
    // do something
}
```

这一切都工作得很好，直到有一天我们想主动退出这个服务程序，于是某个线程调用了 `exit()`，析构了全局对象，结果造成另一个线程在调用 `Calendar::isHoliday` 时发生崩溃：

```
pure virtual method called
terminate called without an active exception
Aborted (core dumped)
```

当然，这只是举例说明“用全局对象实现无状态策略”在多线程中析构可能有危险。在真实的项目中，`Calendar` 应该在运行的时候从外部配置读入²⁵，而不能写死在代码中。

这其实不是 `exit()` 的过错，而是全局对象析构的问题。C++ 标准没有照顾全局对象在多线程环境下的析构，据我看似乎也没有更好的办法。如果确实需要主动结束线程，则可以考虑用 `_exit(2)` 系统调用。它不会试图析构全局对象，但是也不会执行其他任何清理工作，比如 `flush` 标准输出。

由此可见，安全地退出一个多线程的程序并不是一件容易的事情。何况这里还没有涉及如何安全地退出其他正在运行的线程，这需要精心设计共享对象的析构顺序，防止各个线程在退出时访问已失效的对象。在编写长期运行的多线程服务程序的时候，可以不必追求安全地退出，而是让进程进入拒绝服务状态，然后就可以直接杀掉了 (§9.3)。

4.5 善用 `__thread` 关键字

`__thread` 是 GCC 内置的线程局部存储设施（thread local storage）。它的实现非常高效，比 `pthread_key_t` 快很多，见 Ulrich Drepper 写的《ELF Handling For Thread-Local Storage》²⁶。`__thread` 变量的存取效率可与全局变量相比：

```
int g_var;           // 全局变量
__thread int t_var;   // __thread 变量
```

²⁵ 比如，中国每年几大节日放假安排要等到头一年年底才由国务院假日办公布。又比如 2012 年英女王登基 60 周年，英国新加了一两个节日。

²⁶ <http://www.akkadia.org/drepper/tls.pdf>

```

void foo() // 交替显示源代码和汇编代码
{
8048494:      55                push    %ebp
8048495:      89 e5             mov     %esp, %ebp
    g_var = 1;           // 直接寻址
8048497:      c7 05 1c 97 04 08 movl    $0x1, 0x804971c
804849d:      01 00 00 00
    t_var = 2;           // 也是直接寻址, 用了段寄存器 gs
80484a1:      65 c7 05 fc ff ff movl    $0x2, %gs:0xffffffffc
80484a8:      02 00 00 00
}
80484ac:      5d                pop     %ebp
80484ad:      c3                ret

```

__thread 使用规则²⁷: 只能用于修饰 POD 类型, 不能修饰 class 类型, 因为无法自动调用构造函数和析构函数。__thread 可以用于修饰全局变量、函数内的静态变量, 但是不能用于修饰函数的局部变量或者 class 的普通成员变量。另外, __thread 变量的初始化只能用编译期常量。例如:

```

__thread string t_obj1("Chen Shuo"); // 错误, 不能调用对象的构造函数
__thread string* t_obj2 = new string; // 错误, 初始化必须用编译期常量
__thread string* t_obj3 = NULL;      // 正确, 但是需要手工初始化并销毁对象

```

__thread 变量是每个线程有一份独立实体, 各个线程的变量值互不干扰。除了这个主要用途, 它还可以修饰那些“值可能会变, 带有全局性, 但是又不值得用全局锁保护”的变量。muduo 代码中用到了好几处 __thread, 简单列举如下:

- muduo/base/Logging.cc 缓存最近一条日志时间的年月日时分秒, 如果一秒之内输出多条日志, 可避免重复格式化。另外, muduo::strerror_tl 把 strerror_r(3) 做成如同 strerror(3) 一样好用, 而且是线程安全的。
- muduo/base/ProcessInfo.cc 用线程局部变量来简化 ::scandir(3) 的使用。
- muduo/base/Thread.cc 缓存每个线程的 id。
- muduo/net/EventLoop.cc 用于判断当前线程是否只有一个 EventLoop 对象。

以上例子都是 __thread 修饰 POD 类型的变量。

如果要用到 thread local 的 class 对象, 可以考虑使用 muduo::ThreadLocal<T> 和 muduo::ThreadLocalSingleton<T> 这两个 class, 它能在线程退出时销毁 class 对象。例如 examples/asio/chat/server_threaded_highperformance.cc 用 ThreadLocalSingleton 来保存每个 EventLoop 线程所管辖的客户连接, 以实现高效的消息转发 (p. 260)。

²⁷ http://gcc.gnu.org/onlinedocs/gcc/Thread_002dLocal.html

4.6 多线程与 IO

这可算是本章最为关键的一节。本书只讨论同步 IO，包括阻塞与非阻塞，不讨论异步 IO（AIO）。在进行多线程网络编程的时候，几个自然的问题是：如何处理 IO？能否多个线程同时读写同一个 socket 文件描述符²⁸？我们知道用多线程同时处理多个 socket 通常可以提高效率，那么用多线程处理同一个 socket 也可以提高效率吗？

首先，操作文件描述符的系统调用本身是线程安全的，我们不用担心多个线程同时操作文件描述符会造成进程崩溃或内核崩溃。

但是，多个线程同时操作同一个 socket 文件描述符确实很麻烦，我认为是得不偿失的。需要考虑的情况如下：

- 如果一个线程正在阻塞地 read(2) 某个 socket，而另一个线程 close(2) 了此 socket。
- 如果一个线程正在阻塞地 accept(2) 某个 listening socket，而另一个线程 close(2) 了此 socket。
- 更糟糕的是，一个线程正准备 read(2) 某个 socket，而另一个线程 close(2) 了此 socket；第三个线程又恰好 open(2) 了另一个文件描述符，其 fd 号码正好与前面的 socket 相同。这样程序的逻辑就混乱了 (§4.7)。

我认为以上这几种情况都反映了程序逻辑设计上有问题。

现在假设不考虑关闭文件描述符，只考虑读和写，情况也不见得好多。因为 socket 读写的特点是不保证完整性，读 100 字节有可能只返回 20 字节，写操作也是一样的。

- 如果两个线程同时 read 同一个 TCP socket，两个线程几乎同时各自收到一部分数据，如何把数据拼成完整的消息？如何知道哪部分数据先到达？
- 如果两个线程同时 write 同一个 TCP socket，每个线程都只发出去半条消息，那接收方收到数据如何处理？
- 如果给每个 TCP socket 配一把锁，让同时只能有一个线程读或写此 socket，似乎可以“解决”问题，但这样还不如直接始终让同一个线程来操作此 socket 来得简单。
- 对于非阻塞 IO，情况是一样的，而且收发消息的完整性与原子性几乎不可能用锁来保证，因为这样会阻塞其他 IO 线程。

²⁸ 没有特殊说明时，指 TCP socket。

如此看来，理论上只有 `read` 和 `write` 可以分到两个线程去，因为 TCP socket 是双向 IO。问题是真的值得把 `read` 和 `write` 拆开成两个线程吗？

以上讨论的都是网络 IO，那么多线程可以加速磁盘 IO 吗？首先要避免 `lseek(2)/read(2)` 的 `race condition` (§4.2)。做到这一点之后，据我看，用多个线程 `read` 或 `write` 同一个文件也不会提速。不仅如此，多个线程分别 `read` 或 `write` 同一个磁盘上的多个文件也不见得能提速。因为每块磁盘都有一个操作队列，多个线程的读写请求到了内核是排队执行的。只有在内核缓存了大部分数据的情况下，多线程读这些热数据才可能比单线程快。多线程磁盘 IO 的一个思路是每个磁盘配一个线程，把所有针对此磁盘的 IO 都挪到同一个线程，这样或许能避免或减少内核中的锁争用。我认为应该用“显然是正确”的方式来编写程序，一个文件只由一个进程中的一个线程来读写，这种做法显然是正确的。

为了简单起见，我认为多线程程序应该遵循的原则是：每个文件描述符只由一个线程操作，从而轻松解决消息收发的顺序性问题，也避免了关闭文件描述符的各种 `race condition`。一个线程可以操作多个文件描述符，但一个线程不能操作别的线程拥有的文件描述符。这一点不难做到，`muduo` 网络库已经把这些细节封装了。

`epoll` 也遵循相同的原则。Linux 文档并没有说明：当一个线程正阻塞在 `epoll_wait()` 上时，另一个线程往此 `epoll fd` 添加一个新的监视 `fd` 会发生什么。新 `fd` 上的事件会不会在此次 `epoll_wait()` 调用中返回？为了稳妥起见，我们应该把对同一个 `epoll fd` 的操作（添加、删除、修改、等待）都放到同一个线程中执行，这正是我们需要 `muduo::EventLoop::wakeup()` 的原因。

当然，一般的程序不会直接使用 `epoll`、`read`、`write`，这些底层操作都由网络库代劳了。

这条规则有两个例外：对于磁盘文件，在必要的时候多个线程可以同时调用 `pread(2)/pwrite(2)` 来读写同一个文件；对于 UDP，由于协议本身保证消息的原子性，在适当的条件下（比如消息之间彼此独立）可以多个线程同时读写同一个 UDP 文件描述符。

4.7 用 RAII 包装文件描述符

本节谈一谈在多线程程序中如何管理文件描述符。Linux 的文件描述符（file descriptor）是小整数，在程序刚刚启动的时候，0 是标准输入，1 是标准输出，2 是

标准错误。这时如果我们新打开一个文件，它的文件描述符会是 3，因为 POSIX 标准要求每次新打开文件（含 socket）的时候必须使用当前最小可用的文件描述符号码。

POSIX 这种分配文件描述符的方式稍不注意就会造成串话。比如前面举过的例子，一个线程正准备 `read(2)` 某个 socket，而第二个线程几乎同时 `close(2)` 了此 socket；第三个线程又恰好 `open(2)` 了另一个文件描述符，其号码正好与前面的 socket 相同（因为比它小的号码都被占用了）。这时第一个线程可能会读到不属于它的数据，不仅如此，还把第三个线程的功能也破坏了，因为第一个线程把数据读走了（TCP 连接的数据只能读一次，磁盘文件会移动当前位置）。另外一种情况，一个线程从 `fd=8` 收到了比较耗时的请求，它开始处理这个请求，并记住要把响应结果发给 `fd=8`。但是在处理过程中，`fd=8` 断开连接，被关闭了，又有新的连接到来，碰巧使用了相同的 `fd=8`。当线程完成响应的计算，把结果发给 `fd=8` 时，接收方已经物是人非，后果难以预料。

在单线程程序中，或许可以通过某种全局表来避免串话；在多线程程序中，我不认为这种做法会是高效的（通常意味着每次读写都要对全局表加锁）。

在 C++ 里解决这个问题的办法很简单：RAII。用 Socket 对象包装文件描述符，所有对此文件描述符的读写操作都通过此对象进行，在对象的析构函数里关闭文件描述符。这样一来，只要 Socket 对象还活着，就不会有其他 Socket 对象跟它有一样的文件描述符，也就不可能串话。剩下的问题就是做好多线程中的对象生命期管理，这在第 1 章已经完美解决了。

引申问题：为什么服务端程序不应该关闭标准输出（`fd=1`）和标准错误（`fd=2`）？因为有些第三方库在特殊紧急情况下会往 `stdout` 或 `stderr` 打印出错信息，如果我们的程序关闭了标准输出（`fd=1`）和标准错误（`fd=2`），这两个文件描述符有可能被网络连接占用，结果造成对方收到莫名其妙的数据。正确的做法是把 `stdout` 或 `stderr` 重定向到磁盘文件（最好不要是 `/dev/null`），这样我们不至于丢失关键的诊断信息。当然，这应该由启动服务程序的看门狗进程完成²⁹，对服务程序本身是透明的。

现代 C++ 的一个特点是对象生命期管理的进步，体现在不需要手工 `delete` 对象。在网络编程中，有的对象是长命的（例如 `TcpServer`），有的对象是短命的（例如 `TcpConnection`）。长命的对象的生命期往往和整个程序一样长，那就很容易处理，直接使用全局对象（或 `scoped_ptr`）或者做成 `main()` 的栈上对象都行。对于短命的对象，其生命期不一定完全由我们控制，比如对方客户端断开了某个 TCP socket，它对应的服务端进程中的 `TcpConnection` 对象（其必然是个 heap 对象，不可能是 stack

²⁹ 参考 <http://github.com/chenshuo/muduo-protorc> 的 Zurg slave 示例。

对象)的生命也即将走到尽头。但是这时我们并不能立刻 `delete` 这个对象,因为其他地方可能还持有它的引用,贸然 `delete` 会造成空悬指针。只有确保其他地方没有持有该对象的引用的时候,才能安全地销毁对象,这自然会用到引用计数。在多线程程序中,安全地销毁对象不是一件轻而易举的事情,见第 1 章。

在非阻塞网络编程中,我们常常要面临这样一种场景:从某个 TCP 连接 A 收到了一个 `request`,程序开始处理这个 `request`;处理可能要花一定的时间,为了避免耽误(阻塞)处理其他 `request`,程序记住了发来 `request` 的 TCP 连接,在某个线程池中处理这个请求;在处理完之后,会把 `response` 发回 TCP 连接 A。但是,在处理 `request` 的过程中,客户端断开了 TCP 连接 A,而另一个客户端刚好创建了新连接 B。我们的程序不能只记住 TCP 连接 A 的文件描述符,而应该持有封装 socket 连接的 `TcpConnection` 对象,保证在处理 `request` 期间 TCP 连接 A 的文件描述符不会被关闭。或者持有 `TcpConnection` 对象的弱引用(`weak_ptr`),这样能知道 socket 连接在处理 `request` 期间是否已经关闭了,fd=8 的文件描述符到底是“前世”还是“今生”。

否则的话,旧的 TCP 连接 A 一断开,`TcpConnection` 对象销毁,关闭了旧的文件描述符(RAII),而且新连接 B 的 socket 文件描述符有可能等于之前断开的 TCP 连接(这是完全可能的,POSIX 要求每次新建文件描述符时选取当前最小的可用的整数)。当程序处理完旧连接的 `request` 时,就有可能把 `response` 发给新的 TCP 连接 B,造成串话。

为了应对这种情况,防止访问失效的对象或者发生网络串话,`muduo` 使用 `shared_ptr` 来管理 `TcpConnection` 的生命期。这是唯一一个采用引用计数方式管理生命期的对象。如果不用 `shared_ptr`,我想不出其他安全且高效的办法来管理多线程网络服务端程序中的并发连接。

4.8 RAII 与 fork()

在编写 C++ 程序的时候,我们总是设法保证对象的构造和析构是成对出现的,否则就几乎一定会有内存泄漏。在现代 C++ 中,这一点不难做到 (§1.7)。利用这一特性,我们可以用对象来包装资源,把资源管理与对象生命期管理统一起来(RAII)。但是,假如程序会 `fork()`,这一假设就会被破坏了。考虑下面这个例子, `Foo` 对象构造了一次,但是析构了两次。

```
int main()
{
    Foo foo;    // 调用构造函数
    fork();    // fork 为两个进程
    foo.doit() // 在父子进程中都使用 foo
    // 析构函数会被调用两次，父进程和子进程各一次
}
```

如果 Foo class 封装了某种资源，而这个资源没有被子进程继承，那么 Foo::doit() 的功能在子进程中是错乱的。而我们没有办法自动预防这一点，总不能每次申请一个资源就去调用一次 pthread_atfork() 吧？

fork() 之后，子进程继承了父进程的几乎全部状态，但也有少数例外。子进程会继承地址空间和文件描述符，因此用于管理动态内存和文件描述符的 RAII class 都能正常工作。但是子进程不会继承：

- 父进程的内存锁，mlock(2)、mlockall(2)。
- 父进程的文件锁，fcntl(2)。
- 父进程的某些定时器，setitimer(2)、alarm(2)、timer_create(2) 等等。
- 其他，见 man 2 fork。

通常我们会用 RAII 手法来管理以上种类的资源（加锁解锁、创建销毁定时器等），但是在 fork() 出来的子进程中不一定正常工作，因为资源在 fork() 时已经被释放了。比方说用 RAII 技法封装 timer_create()/timer_delete()，在子进程中析构函数调用 timer_delete() 可能会出错，因为试图释放一个不存在的资源。或者更糟糕地把其他对象持有的 timer 给释放了（如果碰巧新建的 timer_t 与之重复的话）。

因此，我们在编写服务端程序的时候，“是否允许 fork()”是在一开始就应该慎重考虑的问题，在一个没有为 fork() 做好准备的程序中使用 fork()，会遇到难以预料的问题。

4.9 多线程与 fork()

多线程与 fork()³⁰ 的协作性很差。这是 POSIX 系列操作系统的历史包袱。因为长期以来程序都是单线程的，fork() 运转正常。当 20 世纪 90 年代初期引入多线程之后，fork() 的适用范围大为缩减。

³⁰ 在现代 Linux glibc 中，fork(3) 不是直接使用 fork(2) 系统调用，而是使用 clone(2) syscall，不过不影响这里的讨论。

`fork()` 一般不能在多线程程序中调用^{31 32}, 因为 Linux 的 `fork()` 只克隆当前线程的 `thread of control`, 不克隆其他线程。`fork()` 之后, 除了当前线程之外, 其他线程都消失了。也就是说不能一下子 `fork()` 出一个和父进程一样的多线程子进程。Linux 没有 `forkall()` 这样的系统调用, `forkall()` 其实也是很难办的 (从语意上), 因为其他线程可能等在 `condition variable` 上, 可能阻塞在系统调用上, 可能等着 `mutex` 以跨入临界区, 还可能在密集的计算中, 这些都不好全盘搬到子进程里。

`fork()` 之后子进程中只有一个线程, 其他线程都消失了, 这就造成一个危险的局面。其他线程可能正好位于临界区之内, 持有了某个锁, 而它突然死亡, 再也没有机会去解锁了。如果子进程试图再对同一个 `mutex` 加锁, 就会立刻死锁。在 `fork()` 之后, 子进程就相当于处于 `signal handler` 之中, 你不能调用线程安全的函数 (除非它是可重入的), 而只能调用异步信号安全 (`async-signal-safe`) 的函数。比方说, `fork()` 之后, 子进程不能调用:

- `malloc(3)`。因为 `malloc()` 在访问全局状态时几乎肯定会加锁。
- 任何可能分配或释放内存的函数, 包括 `new`、`map::insert()`、`snprintf`³³ ……
- 任何 Pthreads 函数。你不能用 `pthread_cond_signal()` 去通知父进程, 只能通过读写 `pipe(2)` 来同步³⁴。
- `printf()` 系列函数, 因为其他线程可能恰好持有 `stdout/stderr` 的锁。
- 除了 `man 7 signal` 中明确列出的“signal 安全”函数之外的任何函数。

照此看来, 唯一安全的做法是在 `fork()` 之后立即调用 `exec()` 执行另一个程序, 彻底隔断子进程与父进程的联系。

不得不说, 同样是创建进程, Windows 的 `CreateProcess()` 函数的顾虑要少得多, 因为它创建的进程跟当前进程关联较少。

4.10 多线程与 signal

Linux/Unix 的信号 (`signal`) 与多线程可谓是不水火不容³⁵。在单线程时代, 编写信号处理函数 (`signal handler`) 就是一件棘手的事情, 由于 `signal` 打断了正在运行

³¹ <http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them>

³² <http://www.cppblog.com/lymons/archive/2008/06/01/51836.html>

³³ 在浮点数转换为字符串的时候有可能需要动态分配内存。

³⁴ 见 <http://github.com/chenshuo/muduo-protorpc> 中 `Zurg slave` 示例的 `Process::start()`。

³⁵ <http://www.linuxprogrammingblog.com/all-about-linux-signals?page=11>

的 thread of control, 在 signal handler 中只能调用 async-signal-safe 的函数³⁶, 即所谓的“可重入 (reentrant)”函数, 就好比在 DOS 时代编写中断处理例程 (ISR)³⁷ 一样。不是每个线程安全的函数都是可重入的, 见 §4.9 举的例子。

还有一点, 如果 signal handler 中需要修改全局数据, 那么被修改的变量必须是 sig_atomic_t 类型的³⁸。否则被打断的函数在恢复执行后很可能不能立刻看到 signal handler 改动后的数据, 因为编译器有可能假定这个变量不会被他处修改, 从而优化了内存访问。

在多线程时代, signal 的语义更为复杂。信号分为两类: 发送给某一线程 (SIGSEGV), 发送给进程中的任一线程 (SIGTERM), 还要考虑掩码 (mask) 对信号的屏蔽等。特别是在 signal handler 中不能调用任何 Pthreads 函数, 不能通过 condition variable 来通知其他线程。

在多线程程序中, 使用 signal 的第一原则是**不要使用 signal**³⁹。包括

- 不要用 signal 作为 IPC 的手段, 包括不要用 SIGUSR1 等信号来触发服务端的行为。如果确实需要, 可以用 §9.5 介绍的增加监听端口的方式来实现双向的、可远程访问的进程控制。
- 也不要使用基于 signal 实现的定时函数, 包括 alarm/ualarm/setitimer/timer_create、sleep/usleep 等等。
- 不主动处理各种异常信号 (SIGTERM、SIGINT 等等), 只用默认语义: 结束进程。有一个例外: SIGPIPE, 服务器程序通常的做法是忽略此信号⁴⁰, 否则如果对方断开连接, 而本机继续 write 的话, 会导致程序意外终止。
- 在没有别的替代方法的情况下 (比方说需要处理 SIGCHLD 信号), 把异步信号转换为同步的文件描述符事件。传统的做法是在 signal handler 里往一个特定的 pipe(2) 写一个字节, 在主程序中从这个 pipe 读取, 从而纳入统一的 IO 事件处理框架中去。现代 Linux 的做法是采用 signalfd(2) 把信号直接转换为文件描述符事件, 从而从根本上避免使用 signal handler⁴¹。

³⁶ http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_04_03

³⁷ http://en.wikipedia.org/wiki/Interrupt_handler

³⁸ http://www.gnu.org/software/libc/manual/html_mono/libc.html#Atomic-Data-Access

³⁹ <http://www.cppblog.com/lymons/archive/2008/06/01/51838.html> 和 [51837.html](http://www.cppblog.com/lymons/archive/2008/06/01/51837.html)

⁴⁰ 在命令行程序中, 默认的 SIGPIPE 行为非常有用。例如查看日志中的前 10 条错误信息, 可以用管道将命令串起来: `gunzip -c log.gz | grep ERROR | head`, 由于 head 关闭了管道的写入端, grep 会遇到 SIGPIPE 而终止, 同理 gunzip 也就不需要解压缩整个巨大的日志文件。这也可能是 Unix 默认使用阻塞 IO 的历史原因之一。

⁴¹ 例子见 <http://github.com/chenshuo/muduo-protorpc> 中 Zurg slave 示例的 ChildManager class。

4.11 Linux 新增系统调用的启示

本节的内容源自我的一篇同名博客⁴²，省略了 `signalfd`、`timerfd`、`eventfd` 等内容，对此感兴趣的读者可阅读原文。

大致从 Linux 内核 2.6.27 起，凡是会创建文件描述符的 `syscall` 一般都增加了额外的 `flags` 参数，可以直接指定 `O_NONBLOCK` 和 `FD_CLOEXEC`，例如：

- `accept4` - 2.6.28
- `eventfd2` - 2.6.27
- `inotify_init1` - 2.6.27
- `pipe2` - 2.6.27
- `signalfd4` - 2.6.27
- `timerfd_create` - 2.6.25

以上 6 个 `syscall`，除了最后一个是 2.6.25 的新功能，其余的都是增强原有的调用，把数字尾号去掉就是原来的 `syscall`。

`O_NONBLOCK` 的功能是开启“非阻塞 IO”，而文件描述符默认是阻塞的。这些创建文件描述符的系统调用能直接设定 `O_NONBLOCK` 选项，其或许能反映当前 Linux（服务端）开发的风向，即我在 §3.3 里推荐的 `one loop per thread + (non-blocking IO with IO multiplexing)`。从这些内核改动来看，`non-blocking IO` 已经主流到让内核增加 `syscall` 以节省一次 `fcntl(2)` 调用的程度了。

另外，以下新系统调用可以在创建文件描述符时开启 `FD_CLOEXEC` 选项：

- `dup3` - 2.6.27
- `epoll_create1` - 2.6.27
- `socket` - 2.6.27

`FD_CLOEXEC` 的功能是让程序 `exec()` 时，进程会自动关闭这个文件描述符。而文件描述默认是被子进程继承的（这是传统 Unix 的一种典型 IPC，比如用 `pipe(2)` 在父子进程间单向通信）。

以上 8 个新 `syscall` 都允许直接指定 `FD_CLOEXEC`，或许说明 `fork()` 的主要目的已经不再是创建 `worker process` 并通过共享的文件描述符和父进程保持通信，而是像 Windows 的 `CreateProcess` 那样创建“干净”的进程（`fork()` 之后立刻 `exec()`），

⁴² <http://blog.csdn.net/Solstice/article/details/5327881>

其与父进程没有多少瓜葛。为了回避 `fork()+exec()` 之间文件描述符泄漏的 `race condition`，这才在几乎所有能新建文件描述符的系统调用上引入了 `FD_CLOEXEC` 参数，参见 Ulrich Drepper 的短文《Secure File Descriptor Handling》⁴³。

以上两个 `flags` 在我看来，说明 Linux 服务器开发的主流模型正在由 `fork() + worker processes` 模型转变为第3章推荐的多线程模型。`fork()` 的使用频度会大大降低，将来或许只有专门负责启动别的进程的“看门狗程序”才会调用 `fork()`，而一般的网络服务器程序不会再 `fork()` 出子进程了。原因之一是，`fork()` 一般不能在多线程程序中调用 (§4.9)。

小结

本章只讨论了多线程编程的技术方面，没有讨论设计方面，特别是没有讨论该如何规划一个多线程服务程序的线程数目及用途。我个人遵循的编写多线程 C++ 程序的原则如下：

- 线程是宝贵的，一个程序可以使用几个或十几个线程。一台机器上不应该同时运行几百个、几千个用户线程，这会大大增加内核 `scheduler` 的负担，降低整体性能。
- 线程的创建和销毁是有代价的，一个程序最好在一开始创建所需的线程，并一直反复使用。不要在运行期间反复创建、销毁线程，如果必须这么做，其频度最好能降到1分钟1次（或更低）。
- 每个线程应该有明确的职责，例如 IO 线程（运行 `EventLoop::loop()`，处理 IO 事件）、计算线程（位于 `ThreadPool` 中，负责计算）等等（p. 73）。
- 线程之间的交互应该尽量简单，理想情况下，线程之间只用消息传递（例如 `BlockingQueue`）方式交互。如果必须用锁，那么最好避免一个线程同时持有两把或更多的锁，这样可彻底防止死锁。
- 要预先考虑清楚一个 `mutable shared` 对象将会暴露给哪些线程，每个线程是读还是写，读写有无可能并发进行。

⁴³ <http://udrepper.livejournal.com/20407.html>

第 5 章

高效的多线程日志

“日志 (logging)” 有两个意思：

- 诊断日志 (**diagnostic log**) 即 log4j、logback、slf4j、glog、g2log、log4cxx、log4cpp、log4cplus、Pantehios、ezlogger 等常用日志库提供的日志功能。
- 交易日志 (**transaction log**) 即数据库的 write-ahead log¹、文件系统的 journaling² 等，用于记录状态变更，通过回放日志可以逐步恢复每一次修改之后的状态。

本章的“日志”是前一个意思，即文本的、供人阅读的日志，通常用于故障诊断和追踪 (trace)³，也可用于性能分析。日志通常是分布式系统中事故调查时的唯一线索，用来追寻蛛丝马迹，查出元凶。

在服务端编程中，日志是必不可少的，在生产环境中应该做到 “Log Everything All The Time”⁴。对于关键进程，日志通常要记录

1. 收到的每条内部消息的 id（还可以包括关键字段、长度、hash 等）；
2. 收到的每条外部消息的全文⁵；
3. 发出的每条消息的全文，每条消息都有全局唯一的 id⁶；
4. 关键内部状态的变更，等等。

每条日志都有时间戳，这样就能完整追踪分布式系统中一个事件的来龙去脉。也只有这样才能查清楚发生故障时究竟发生了什么，比如业务处理流程卡在了哪一步。

¹ http://en.wikipedia.org/wiki/Write-ahead_logging 不同的数据库有不同的称呼，如 binary log、redo log 等。

² http://en.wikipedia.org/wiki/Journaling_file_system

³ <http://en.wikipedia.org/wiki/Traceability>

⁴ <http://highscalability.com/log-everything-all-time>

⁵ 第 2、3 两条或许不适用于分布式存储系统的 bulk data，但适用于 meta data。

⁶ 可用 §9.4 的办法生成。

诊断日志不光是给程序员看的，更多的时候是给运维人员看的，因此日志的内容应避免造成误解，不要误导调查故障的主攻方向，拖延故障解决的时间。

一个日志库大体可分为前端⁷（frontend）和后端⁸（backend）两部分。前端是供应用程序使用的接口（API），并生成日志消息（log message）；后端则负责把日志消息写到目的地（destination）。这两部分的接口有可能简单到只有一个回调函数：

```
void output(const char* message, int len);
```

其中的 message 字符串是一条完整的日志消息，包含日志级别、时间戳、源文件位置、线程 id 等基本字段，以及程序输出的具体消息内容。

在多线程程序中，前端和后端都与单线程程序无甚区别，无非是每个线程有自己的前端，整个程序共用一个后端。但难点在于将日志数据从多个前端高效地传输到后端⁹。这是一个典型的多生产者 - 单消费者问题，对生产者（前端）而言，要尽量做到低延迟、低 CPU 开销、无阻塞；对消费者（后端）而言，要做到足够大的吞吐量，并占用较少资源。

对 C++ 程序而言，最好整个程序（包括主程序和程序库）都使用相同的日志库，程序有一个整体的日志输出，而不要各个组件有各自的日志输出。从这个意义上讲，日志库是个 singleton。

C++ 日志库的前端大体上有两种 API 风格：

- C/Java 的 printf(fmt, ...) 风格，例如
log_info("Received %d bytes from %s", len, getClientName().c_str());
- C++ 的 stream << 风格，例如
LOG_INFO << "Received " << len << " bytes from " << getClientName();

muduo 日志库是 C++ stream 风格，这样用起来更自然，不必费心保持格式字符串与参数类型的一致性，可以随用随写，而且是类型安全¹⁰的。

stream 风格的另一个好处是当输出的日志级别高于语句的日志级别时，打印日志是个空操作¹¹，运行时开销接近零。比方说当日志级别为 WARNING 时，LOG_INFO <<

⁷ muduo/base/Logging.{h,cc}

⁸ muduo/base/LogFile.{h,cc}

⁹ muduo/base/AsyncLogging.{h,cc}

¹⁰ printf(fmt, ...) 风格在 C++ 中也可以做到类型安全，但是在 C++11 引入 variadic template 之前很费劲。因为 C++ 不允许把 non-POD 对象通过可变参数 (...) 传入函数。Pantheios 日志库用的是重载函数模板的办法 (<http://www.pantheios.org>)。

¹¹ <http://www.drdobbs.com/cpp/201804215>

是空操作，这个语句根本不会调用 `std::string getClientName()` 函数，减小了开销。而 `printf` 风格不易做到这一点。

`muduo` 没有用标准库中的 `iostream`，而是自己写的 `LogStream class`¹²，这主要是出于性能原因 (§11.6.6)。

5.1 功能需求

常规的通用日志库如 `log4j`¹³/`logback`¹⁴ 通常会提供丰富的功能，但这些功能不一定全都是必需的。

1. 日志消息有多种级别 (level)，如 `TRACE`、`DEBUG`、`INFO`、`WARN`、`ERROR`、`FATAL` 等。
2. 日志消息可能有多个目的地 (appender)，如文件、`socket`、`SMTP` 等。
3. 日志消息的格式可配置 (layout)，例如 `org.apache.log4j.PatternLayout`。
4. 可以设置运行时过滤器 (filter)，控制不同组件的日志消息的级别和目的地。

在上面这几项中，我认为除了第一项之外，其余三项都是非必需的功能。

日志的输出级别在运行时可调，这样同一个可执行文件可以分别在 QA 测试环境的时候输出 `DEBUG` 级别的日志，在生产环境输出 `INFO` 级别的日志¹⁵。在必要的时候也可以临时在线调整日志的输出级别。例如某台机器的消息量过大、日志文件太多、磁盘空间紧张，那么可以临时调整为 `WARNING` 级别输出，减少日志数目。又比如某个新上线的进程的行为略显古怪，则可以临时调整为 `DEBUG` 级别输出，打印更细节的日志消息以便分析查错。调整日志的输出级别不需要重新编译，也不需要重启进程，只要调用 `muduo::Logger::setLogLevel()` 就能即时生效。

对于分布式系统中的服务进程而言，日志的目的地 (destination) 只有一个：本地文件。往网络写日志消息是不靠谱的，因为诊断日志的功能之一正是诊断网络故障，比如连接断开 (网卡或交换机故障)、网络暂时不通 (若干秒之内没有收到心跳消息)、网络拥塞 (消息延迟明显加大) 等等。如果日志消息也是通过网络发到另一台机器上的，那岂不是一损俱损？如果接收网络日志消息的服务器 (日志服务器) 发生故障或者出现进程死锁 (阻塞)，通常会导致发送日志的多个服务进程阻塞，或者

¹² `muduo/base/LogStream.{h,cc}`

¹³ <http://logging.apache.org/log4j/1.2/manual.html>

¹⁴ <http://logback.qos.ch/manual/index.html>

¹⁵ `muduo` 默认输出 `INFO` 级别的日志，可以通过环境变量控制输出 `DEBUG` 或 `TRACE` 级别的日志。

内存暴涨（用户态和内核的 TCP 缓存），这无异于放大了单机故障。往网络写日志消息的另一个坏处是增加网络带宽消耗。试想收到一条业务消息、发出一条业务消息时都会写日志，如果写到网络上岂不是让网络带宽消耗翻倍，加剧 trashing？同理，应该避免往网络文件系统（例如 NFS）上写日志，这等于掩耳盗铃。

以本地文件为日志的 destination，那么日志文件的滚动（rolling）是必需的，这样可以简化日志归档（archive）的实现。rolling 的条件通常有两个：文件大小（例如每写满 1GB 就换下一个文件）和时间（例如每天零点新建一个日志文件，不论前一个文件有没有写满）。muduo 日志库的 LogFile 会自动根据文件大小和时间来主动滚动日志文件。既然能主动 rolling，自然也就不必支持 SIGUSR1 了，毕竟多线程程序处理 signal 很麻烦（§4.10）。

一个典型的日志文件的文件名如下：

```
logfile_test.20120603-144022.hostname.3605.log
```

文件名由以下几部分组成：

- 第 1 部分 logfile_test 是进程的名字。通常是 main() 函数参数中 argv[0] 的 basename(3)，这样容易区分究竟是哪个服务程序的日志。必要时还可以把程序版本加进去。
- 第 2 部分是文件的创建时间（GMT 时区）。这样很容易通过文件名来选择某一时间范围内的日志，例如用通配符 *.20120603-14* 表示 2012 年 6 月 3 日下午 2 点（GMT）左右的日志文件(s)。
- 第 3 部分是机器名称。这样即便把日志文件拷贝到别的机器上也能追溯其来源。
- 第 4 部分是进程 id。如果一个程序一秒之内反复重启，那么每次都会生成不同的日志文件，参考 §9.4。
- 第 5 部分是统一的后缀名 .log。同样是为了便于周边配套脚本的编写。

muduo 的日志文件滚动没有采用文件改名的办法，即 dmesg.log 是最新日志，dmesg.log.1 是前一个日志，dmesg.log.2.gz 是更早的日志等。这种做法的一个好处是 dmesg.log 始终是最新日志，便于编写某些及时解析日志的脚本。将来可以增加一个功能，每次滚动日志文件之后立刻创建（更新）一个 symlink，logfile_test.log 始终指向当前最新的日志文件，这样达到相同的效果。

日志文件压缩与归档¹⁶（archive）不是日志库应有的功能，而应该交给专门的脚本去做，这样 C++ 和 Java 的服务程序可以共享这一基础设施。如果想更换日志压

¹⁶ 例如在非繁忙时段把压缩后的日志文件拷贝到某个 NFS 位置，以便集中保存和分析。

缩算法或归档策略也不必动业务程序，改改周边配套脚本就行了。磁盘空间监控也不是日志库的必备功能。有人或许曾经遇到日志文件把磁盘占满的情况，因此希望日志库能限制空间使用，例如只分配 10GB 磁盘空间，用满之后就冲掉旧日志，重复利用空间，就像循环磁带一样。殊不知如果出现程序死循环拼命写日志的异常情况，那么往往是开头的几条日志最关键，它往往反映了引发异常（busy-loop）的原因（例如收到某条非法消息），后面都是无用的垃圾日志。如果日志库具备重复利用空间的“功能”，只会帮倒忙。磁盘写入的带宽按 100MB/s 计算，写满一个 100GB 的磁盘分区需要 16 分钟，这足够监控系统报警并人工干预了（§9.2.1）。

往文件写日志的一个常见问题是，万一程序崩溃，那么最后若干条日志往往就丢失了，因为日志库不能每条消息都 flush 硬盘，更不能每条日志都 open/close 文件，这样性能开销太大。muduo 日志库用两个办法来应对这一点，其一是定期（默认 3 秒）将缓冲区内的日志消息 flush 到硬盘；其二是每条内存中的日志消息都带有 cookie（或者叫哨兵值/sentry），其值为某个函数的地址，这样通过在 core dump 文件中查找 cookie¹⁷ 就能找到尚未来得及写入磁盘的消息。

日志消息的格式是固定的，不需要运行时配置，这样可节省每条日志解析格式字符串的开销。我认为日志的格式在项目的整个生命周期几乎不会改变，因为我们经常会为不同目的编写 parse 日志的脚本，既要解析最近几天的日志文件，也要和几个月之前，甚至一年之前的日志文件的同类数据做对比。如果在此期间日志格式变了，势必会增加很多无谓的工作量。如果真的需要调整消息格式，直接修改代码并重新编译即可。以下是 muduo 日志库的默认消息格式：

```
日期      时间      微秒    线程   级别   正文      源文件名: 行号
20120603 08:02:46.125770Z 23261 INFO  Hello - test.cc:51
20120603 08:02:46.126926Z 23261 WARN  World - test.cc:52
20120603 08:02:46.126997Z 23261 ERROR Error - test.cc:53
```

日志消息格式有几个要点：

- 尽量每条日志占一行。这样很容易用 awk、sed、grep 等命令行工具来快速联机分析日志，比方说要查看“2012-06-03 08:02:00”至“2012-06-03 08:02:59”这 1 分钟内每秒打印日志的条数（直方图），可以运行
`$ grep -o '^20120603 08:02:..' | sort | uniq -c`
- 时间戳精确到微秒。每条消息都通过 gettimeofday(2) 获得当前时间，这么做不会有什么性能损失。因为在 x86-64 Linux 上，gettimeofday(2) 不是系统调用，不会陷入内核¹⁸（可用 strace(1) 验证 muduo/base/tests/Timestamp_unittest.cc）。

¹⁷ 可以用 gdb 的 find 命令。用 strings(1) 命令也能从 core 文件里找到不少有用的信息。

¹⁸ <http://lwn.net/Articles/446528/>

- 始终使用 GMT 时区 (Z)。对于跨洲的分布式系统而言,可省去本地时区转换的麻烦(别忘了主要西方国家大多实行夏令时),更易于追查事件的顺序。
- 打印线程 id。便于分析多线程程序的时序,也可以检测死锁¹⁹。这里的线程 id 是指调用 LOG_INFO << 的线程,线程 id 的获取见 §4.3。
- 打印日志级别。在线查错的时候先看看有无 ERROR 日志,通常可加速定位问题。
- 打印源文件名和行号。修复 bug 的时候不至于搞错对象。

每行日志的前 4 个字段的宽度是固定的,以空格分隔,便于用脚本解析。另外,应该避免在日志格式(特别是消息 id²⁰)中出现正则表达式的元字符(meta character),例如 '[' 和 ']' 等等,这样在用 less(1) 查看日志文件的时候查找字符串更加便捷。

运行时的日志过滤器(filter)或许是有用的,例如控制不同部件(程序库)的输出日志级别,但我认为这应该放到编译期去做,整个程序有一个整体的输出级别就足够好了。同时我认为一个程序同时写多个日志文件²¹是非常罕见的需求,这可以事后留给 log archiver 来分流,不必做到日志库中。不实现 filter 自然也能减小生成每条日志的运行时开销,可以提高日志库的性能。

5.2 性能需求

编写 Linux 服务端程序的时候,我们需要一个高效的日志库。只有日志库足够高效,程序员才敢在代码中输出足够多的诊断信息,减小运维难度,提升效率。高效性体现在几方面:

- 每秒写几千上万条日志的时候没有明显的性能损失。
- 能应对一个进程产生大量日志数据的场景,例如 1GB/min。
- 不阻塞正常的执行流程。
- 在多线程程序中,不造成争用(contention)。

这里列举一些具体的性能指标,考虑往普通 7200rpm SATA 硬盘写日志文件的情况:

- 磁盘带宽约是 110MB/s,日志库应该能瞬时写满这个带宽(不必持续太久)。
- 假如每条日志消息的平均长度是 110 字节,这意味着 1 秒要写 100 万条日志。

以上是“高性能”日志库的最低指标。如果磁盘带宽更高,那么日志库的预期性能指标也会相应提高。反过来说,在磁盘带宽确定的情况下,日志库的性能只要“足

¹⁹ 例如某个繁忙的线程在某一时刻之后突然不再 log 任何消息,往往意味着发生了死锁或阻塞(僵死)。

²⁰ 对于 Base64 编码的消息 id,可以将其中的 '+' 替换为 '-', 见 RFC 4648 第 5 节。

²¹ 例如不同的日志级别或不同的组件写到不同的文件。

够好”就行了。假如某个神奇的日志库 1 秒能往 `/dev/null` 写 1000 MB 数据，那么到哪里去找这么快的磁盘来让程序写诊断日志呢？

这些指标初看起来有些异想天开，什么程序需要 1 秒写 100 万条日志消息呢？换一个角度其实很容易想明白，如果一个程序耗尽全部 CPU 资源和磁盘带宽可以做到 1 秒写 100 万条日志消息，那么当只需要 1 秒写 10 万条日志的时候²²，立刻就能腾出 90% 的资源来干正事（处理业务）。相反，如果一个日志库在满负荷的情况下只能 1 秒写 10 万条日志，真正用到生产环境，恐怕就只能 1 秒写 1 万条日志才不会影响正常业务处理，这其实钳制了服务器的吞吐量。

以下是 muduo 日志库在两台机器上的实测性能数据²³。

硬件→	E5320		i5-2500	
文件↓	消息/s	MiB/s	消息/s	MiB/s
nop	97.1 万	107.6	242.2 万	256.2
<code>/dev/null</code>	91.2 万	101.1	234.2 万	247.7
<code>/tmp/log</code>	81.0 万	89.8	213.0 万	225.3

可见 muduo 日志库在现在的 PC 上能写到每秒 200 万条消息，带宽足够撑满两个千兆网连接或 4 个 SATA 组成的 RAID10，性能是达标的²⁴。

为了实现这样的性能指标，muduo 日志库的实现有几点优化措施值得一提：

- 时间戳字符串中的日期和时间两部分是缓存的，一秒之内的多条日志只需重新格式化微秒部分²⁵。例如 p. 111 出现的 3 条日志消息中，“20120603 08:02:46”是复用的，每条日志只需要格式化微秒部分（“.125770Z”）。
- 日志消息的前 4 个字段是定长的，因此可以避免在运行期求字符串长度（不会反复调用 `strlen`²⁶）。因为编译器认识 `memcpy()` 函数，对于定长的内存复制，会在编译期把它 `inline` 展开为高效的目标代码。
- 线程 id 是预先格式化为字符串，在输出日志消息时只需简单拷贝几个字节。见 `CurrentThread::tidString()`。
- 每行日志消息的源文件名部分采用了编译期计算来获得 `basename`，避免运行期 `strrchr(3)` 开销。见 `SourceFile` class，这里利用了 gcc 的内置函数。

²² 比方说一秒处理两三万条消息，每条消息写三条日志：从哪里收到、计算结果如何、发到哪里。

²³ `muduo/base/tests/Logging_test.cc`

²⁴ 日志文件是顺序写入，是对磁盘最友好的一种负载，对 IOPS 要求不高。

²⁵ 见 `muduo/base/Logging.cc` 中的 `Logger::Impl::formatTime()` 函数。

²⁶ 见 `muduo/base/Logging.cc` 中的 `class T` 和 `operator<<(LogStream& s, T v)`。

5.3 多线程异步日志

多线程程序对日志库提出了新的需求：线程安全，即多个线程可以并发写日志，两个线程的日志消息不会出现交织。线程安全不难办到，简单的办法是用一个全局 `mutex` 保护 IO，或者每个线程单独写一个日志文件²⁷，但这两种做法的高效性就堪忧了。前者会造成全部线程抢一个锁，后者有可能让业务线程阻塞在写磁盘操作上。

我认为一个多线程程序的每个进程最好只写一个日志文件，这样分析日志更容易，不必在多个文件中跳来跳去。再说多线程写多个文件也不一定能提速，见 p. 99 的分析。解决办法不难想到，用一个背景线程负责收集日志消息，并写入日志文件，其他业务线程只管往这个“日志线程”发送日志消息，这称为“异步日志”。

在多线程服务程序中，异步日志（叫“非阻塞日志”似乎更准确）是必需的，因为如果在网络 IO 线程或业务线程中直接往磁盘写数据的话，写操作偶尔可能阻塞长达数秒之久（原因很复杂，可能是磁盘或磁盘控制器复位）。这可能导致请求方超时，或者耽误发送心跳消息，在分布式系统中更可能造成多米诺骨牌效应，例如误报死锁引发自动 failover 等。因此，在正常的实时业务处理流程中应该彻底避免磁盘 IO，这在使用 `one loop per thread` 模型的非阻塞服务端程序中尤为重要，因为线程是复用的，阻塞线程意味着影响多个客户连接。

我们需要一个“队列”来将日志前端的数据传送到后端（日志线程），但这个“队列”不必是现成的 `BlockingQueue<std::string>`，因为不用每次产生一条日志消息都通知（`notify()`）接收方。

`muduo` 日志库采用的是双缓冲（`double buffering`）技术²⁸，基本思路是准备两块 `buffer`：A 和 B，前端负责往 `buffer A` 填数据（日志消息），后端负责将 `buffer B` 的数据写入文件。当 `buffer A` 写满之后，交换 A 和 B，让后端将 `buffer A` 的数据写入文件，而前端则往 `buffer B` 填入新的日志消息，如此往复。用两个 `buffer` 的好处是在新建日志消息的时候不必等待磁盘文件操作，也避免每条新日志消息都触发（唤醒）后端日志线程。换言之，前端不是将一条条日志消息分别传送给后端，而是将多条日志消息拼成一个大的 `buffer` 传送给后端，相当于批处理，减少了线程唤醒的频度，降低开销。另外，为了及时将日志消息写入文件，即便 `buffer A` 未满，日志库也会每 3 秒执行一次上述交换写入操作。

`muduo` 异步日志的性能开销大约是前端每写一条日志消息耗时 $1.0\mu\text{s} \sim 1.6\mu\text{s}$ 。

²⁷ Google C++ 日志库的默认多线程实现即如此。

²⁸ http://en.wikipedia.org/wiki/Multiple_buffering

关键代码

实际实现采用了四个缓冲区，这样可以进一步减少或避免日志前端的等待。数据结构如下（muduo/base/AsyncLogging.h）：

```
typedef boost::ptr_vector<LargeBuffer> BufferVector;
typedef BufferVector::auto_type        BufferPtr;
muduo::MutexLock    mutex_;
muduo::Condition    cond_;
BufferPtr           currentBuffer_; // 当前缓冲
BufferPtr           nextBuffer_;    // 预备缓冲
BufferVector        buffers_;       // 待写入文件的已填满的缓冲
```

其中，LargeBuffer 类型是 FixedBuffer class template 的一份具体实现（instantiation），其大小为 4MB，可以存至少 1000 条日志消息。boost::ptr_vector<T>::auto_type 类型类似 C++11 中的 std::unique_ptr，具备移动语义（move semantics），而且能自动管理对象生命期。mutex_ 用于保护后面的四个数据成员。buffers_ 存放的是供后端写入的 buffer。

先来看发送方代码，即 p. 108 回调函数 output() 的实现。

```

28 void AsyncLogging::append(const char* logline, int len)
29 {
30     muduo::MutexLockGuard lock(mutex_);
31     if (currentBuffer_>avail() > len)
32     { // most common case: buffer is not full, copy data here
33         currentBuffer_>append(logline, len);
34     }
35     else // buffer is full, push it, and find next spare buffer
36     {
37         buffers_.push_back(currentBuffer_.release());
38
39         if (nextBuffer_) // is there is one already, use it
40         {
41             currentBuffer_ = boost::ptr_container::move(nextBuffer_); // 移动，而非复制
42         }
43         else // allocate a new one
44         {
45             currentBuffer_.reset(new LargeBuffer); // Rarely happens
46         }
47         currentBuffer_>append(logline, len);
48         cond_.notify();
49     }
50 }

```

muduo/base/AsyncLogging.cc

前端在生成一条日志消息的时候会调用 AsyncLogging::append()。在这个函数中，如果当前缓冲（currentBuffer_）剩余的空间足够大（L31），则会直接把日志消息拷

贝（追加）到当前缓冲中（ $\mathcal{L}33$ ），这是最常见的情况。这里拷贝一条日志消息并不会带来多大开销（p. 120）。前后端代码的其余部分都没有拷贝，而是简单的指针交换。

否则，说明当前缓冲已经写满，就把它送入（移入）`buffers_`（ $\mathcal{L}37$ ），并试图把预备好的另一块缓冲（`nextBuffer_`）移用（`move`）为当前缓冲（ $\mathcal{L}39\sim\mathcal{L}42$ ），然后追加日志消息并通知（唤醒）后端开始写入日志数据（ $\mathcal{L}47\sim\mathcal{L}48$ ）。以上两种情况在临界区之内都没有耗时的操作，运行时间为常数。

如果前端写入速度太快，一下子把两块缓冲都用完了，那么只好分配一块新的 `buffer`，作为当前缓冲（ $\mathcal{L}43\sim\mathcal{L}46$ ），这是极少发生的情况。

再来看接收方（后端）实现，这里只给出了最关键的临界区内的代码（ $\mathcal{L}59\sim\mathcal{L}72$ ），其他琐事请见源文件。

```

51 void AsyncLogging::threadFunc()
52 {
53     BufferPtr newBuffer1(new LargeBuffer);
54     BufferPtr newBuffer2(new LargeBuffer);
55     BufferVector buffersToWrite; // reserve() 从略
56     while (running_)
57     {
58         // swap out what need to be written, keep CS short
59         {
60             muduo::MutexLockGuard lock(mutex_);
61             if (buffers_.empty()) // unusual usage!
62             {
63                 cond_.waitForSeconds(flushInterval_);
64             }
65             buffers_.push_back(currentBuffer_.release()); // 移动，而非复制
66             currentBuffer_ = boost::ptr_container::move(newBuffer1); // 移动，而非复制
67             buffersToWrite.swap(buffers_); // 内部指针交换，而非复制
68             if (!nextBuffer_)
69             {
70                 nextBuffer_ = boost::ptr_container::move(newBuffer2); // 移动，而非复制
71             }
72         }
73         // output buffersToWrite to file
74         // re-fill newBuffer1 and newBuffer2
75     }
76     // flush output
77 }

```

muduo/base/AsyncLogging.cc

首先准备好两块空闲的 `buffer`，以备在临界区内交换（ $\mathcal{L}53$ 、 $\mathcal{L}54$ ）。在临界区内，等待条件触发（ $\mathcal{L}61\sim\mathcal{L}64$ ），这里的条件有两个：其一是超时，其二是前端写满了一个或多个 `buffer`。注意这里是非常规的 `condition variable` 用法，它没有使用 `while` 循环，而且等待时间有上限。

当“条件”满足时，先将当前缓冲（currentBuffer_）移入 buffers_（L65），并立刻将空闲的 newBuffer1 移为当前缓冲（L66）。注意这整段代码位于临界区之内，因此不会有任何 race condition。接下来将 buffers_ 与 buffersToWrite 交换（L67），后面的代码可以在临界区之外安全地访问 buffersToWrite，将其中的日志数据写入文件（L73）。临界区里最后干的一件事情是用 newBuffer2 替换 nextBuffer_（L68~L71），这样前端始终有一个预备 buffer 可供调配。nextBuffer_ 可以减少前端临界区分配内存的概率，缩短前端临界区长度。注意到后端临界区内也没有耗时的操作，运行时间为常数。

L74 会将 buffersToWrite 内的 buffer 重新填充 newBuffer1 和 newBuffer2，这样下一次执行的时候还有两个空闲 buffer 可用于替换前端的当前缓冲和预备缓冲。最后，这四个缓冲在程序启动的时候会全部填充为 0，这样可以避免程序热身时 page fault 引发性能不稳定。

运行图示

以下再用图表展示前端和后端的具体交互情况。一开始先分配好四个缓冲区 A、B、C、D，前端和后端各持有其中两个。前端和后端各有一个缓冲区数组，初始时都是空的。

第一种情况是前端写日志的频度不高，后端 3 秒超时后将“当前缓冲 currentBuffer_”写入文件，见图 5-1（图中变量名为简写，下同）。

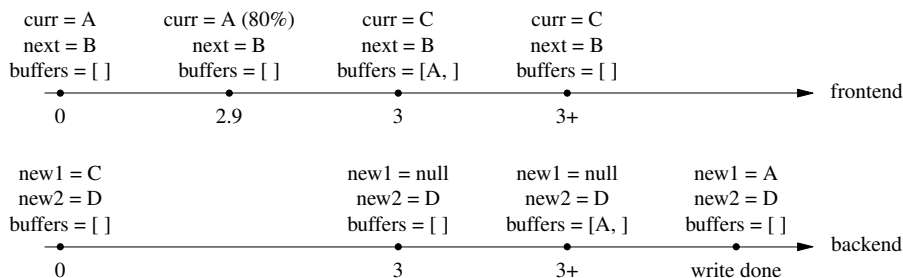


图 5-1

在第 2.9 秒的时候，currentBuffer_ 使用了 80%，在第 3 秒的时候后端线程醒过来，先把 currentBuffer_ 送入 buffers_（L65），再把 newBuffer1 移为 currentBuffer_（L66）。随后第 3+ 秒，交换 buffers_ 和 buffersToWrite（L67），离开临界区，后端开始将 buffer A 写入文件。写完（write done）之后再 newBuffer1 重新填上，等待下一次 cond_.waitForSeconds() 返回。

后面在画图时将有所简化，不再画出 buffers_ 和 buffersToWrite 交换的步骤。

第二种情况，在 3 秒超时之前已经写满了当前缓冲，于是唤醒后端线程开始写入文件，见图 5-2。

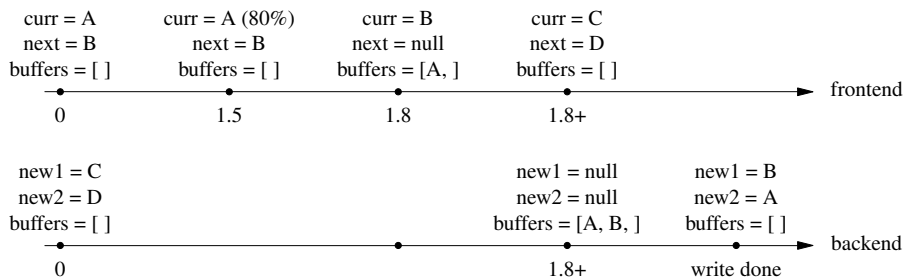


图 5-2

在第 1.5 秒的时候，currentBuffer_ 使用了 80%；第 1.8 秒，currentBuffer_ 写满，于是将当前缓冲送入 buffers_ (L37)，并将 nextBuffer_ 移用为当前缓冲 (L39~L42)，然后唤醒后端线程开始写入。当后端线程唤醒之后 (第 1.8+ 秒)，先将 currentBuffer_ 送入 buffers_ (L65)，再把 newBuffer1 移用为 currentBuffer_ (L66)，然后交换 buffers_ 和 buffersToWrite (L67)，最后用 newBuffer2 替换 nextBuffer_ (L68~L71)，即保证前端有两个空缓冲可用。离开临界区之后，将 buffersToWrite 中的缓冲 A 和 B 写入文件，写完之后重新填充 newBuffer1 和 newBuffer2，完成一次循环。

上面这两种情况都是最常见的，再来看一看前端需要分配新 buffer 的两种情况。

第三种情况，前端在短时间内密集写入日志消息，用完了两个缓冲，并重新分配了一块新的缓冲，见图 5-3。

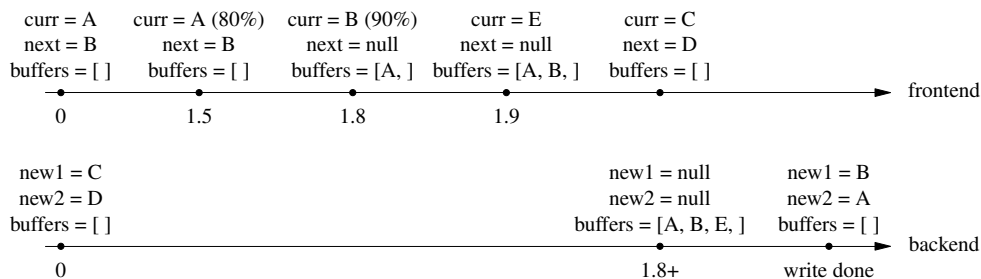


图 5-3

在第 1.8 秒的时候，缓冲 A 已经写满，缓冲 B 也接近写满，并且已经 notify() 了后端线程，但是出于种种原因，后端线程并没有立刻开始工作。到了第 1.9 秒，缓冲 B 也已经写满，前端线程新分配了缓冲 E。到了第 1.8+ 秒，后端线程终于获得控制权，将 C、D 两块缓冲交给前端，并开始将 A、B、E 依次写入文件。一段时间之后，

完成写入操作，用 A、B 重新填充那两块空闲缓冲。注意这里有意用 A 和 B 来填充 newBuffer1/2，而释放了缓冲 E，这是因为使用 A 和 B 不会造成 page fault。

思考题：阅读代码并回答，缓冲 E 是何时在哪个线程释放的？

第四种情况，文件写入速度较慢，导致前端耗尽了两个缓冲，并分配了新缓冲，见图 5-4。

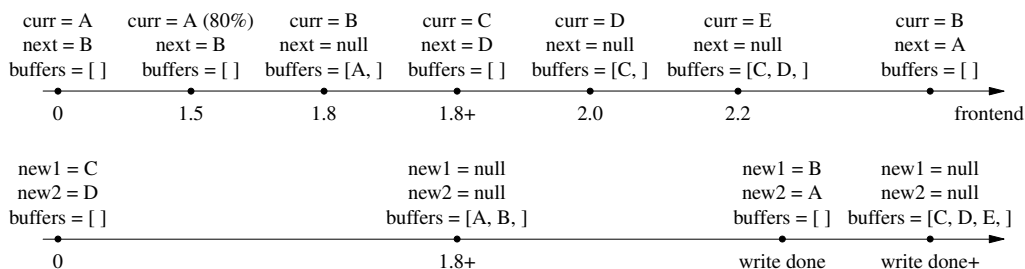


图 5-4

前 1.8+ 秒的场景和前面“第二种情况”相同，前端写满了一个缓冲，唤醒后端线程开始写入文件。之后，后端花了较长时间（大半秒）才将数据写完。这期间前端又用完了两个缓冲，并分配了一个新的缓冲，这期间前端的 notify() 已经丢失。当后端写完（write done）后，发现 buffers_ 不为空（L61），立刻进入下一循环。即替换前端的两个缓冲，并开始一次写入 C、D、E。假定前端在此期间产生的日志较少，请读者补全后续的情况。

改进措施

前面我们一共准备了四块缓冲，应该足以应付日常的需求。如果需要进一步增加 buffer 数目，可以改用下面的数据结构。

```

BufferPtr    currentBuffer_; // 当前缓冲
BufferVector emptyBuffers_;  // 空闲缓冲
BufferVector fullBuffers_;   // 已写满的缓冲

```

初始化时在 emptyBuffers_ 中放入足够多空闲 buffer，这样前端几乎不会遇到需要在临界区内新分配 buffer 的情况，这是一种空间换时间的做法。为了避免短时突发写大量日志造成新分配的 buffer 占用过多内存，后端代码应该保证 emptyBuffers_ 和 fullBuffers_ 的长度之和不超过某个定值。buffer 在前端和后端之间流动，形成一个循环，如图 5-5 所示。

以上改进留作练习。

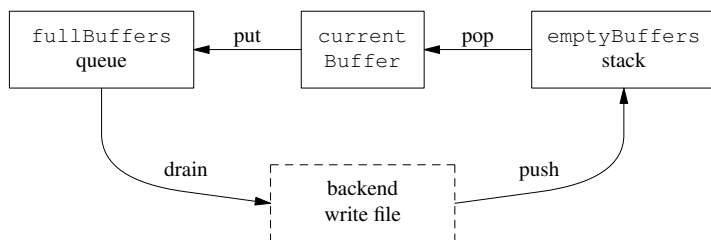


图 5-5

如果日志消息堆积怎么办

万一前端陷入死循环，拼命发送日志消息，超过后端的处理（输出）能力，会导致什么后果？对于同步日志来说，这不是问题，因为阻塞 IO 自然就限制了前端的写入速度，起到了节流阀（throttling）的作用。但是对于异步日志来说，这就是典型的生产速度高于消费速度问题，会造成数据在内存中堆积，严重时引发性能问题（可用内存不足）或程序崩溃（分配内存失败）。

muduo 日志库处理日志堆积的方法很简单：直接丢掉多余的日志 buffer，以腾出内存，见 `muduo/base/AsyncLogging.cc` 第 87 ~ 96 行代码。这样可以防止日志库本身引起程序故障，是一种自我保护措施。将来或许可以加上网络报警功能，通知人工介入，以尽快修复故障。

5.4 其他方案

当然在前端和后端之间高效传递日志消息的办法不止这一种，比方说使用常规的 `muduo::BlockingQueue<std::string>` 或 `muduo::BoundedBlockingQueue<std::string>` 在前后端之间传递日志消息，其中每个 `std::string` 是一条消息。这种做法每条日志消息都要分配内存，特别是在前端线程分配的内存要由后端线程释放，因此对 `malloc` 的实现要求较高，需要针对多线程特别优化。另外，如果用这种方案，那么需要修改 `LogStream` 的 `Buffer`，使之直接将日志写到 `std::string` 中，可节省一次内存拷贝。

相比前面展示的直接拷贝日志消息的做法，这个传递指针的方案似乎会更高效，但是据我测试²⁹，直接拷贝日志数据的做法比传递指针快 3 倍（在每条日志消息不大于 4kB 的时候），估计是内存分配的开销所致。因此 `muduo` 日志库只提供了 §5.3 介

²⁹ 代码见 `recipes/logging/AsyncLogging*`。

绍的这一种异步日志机制。这再次说明“性能”不能凭感觉说了算，一定要有典型场景的测试数据作为支撑。

muduo 现在的异步日志实现用了一个全局锁。尽管临界区很小，但是如果线程数目较多，锁争用（lock contention）也可能影响性能。一种解决办法是像 Java 的 `ConcurrentHashMap` 那样用多个桶子（bucket），前端写日志的时候再按线程 id 哈希到不同的 bucket 中，以减少 contention。这种方案的后端实现较为复杂，有兴趣的读者可以试一试。

为了简化实现，目前 muduo 日志库只允许指定日志文件的名称，不允许指定其路径。日志库会把日志文件写到当前路径，因此可以在启动脚本（shell 脚本）里改变当前路径，以达到相同的目的。

Linux 默认会把 core dump 写到当前目录，而且文件名是固定的 core。为了不让新的 core dump 文件冲掉旧的，我们可以通过 `sysctl` 设置 `kernel.core_pattern` 参数（也可以修改 `/proc/sys/kernel/core_pattern`），让每次 core dump 都产生不同的文件。例如设为 `%e.%t.%p.%u.core`，其中各个参数的意义见 `man 5 core`。另外也可以使用 `Apport` 来收集有用的诊断信息，见 <https://wiki.ubuntu.com/Apport>。

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
1  #!/usr/bin/python
2
3  import socket, time
4
5  serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6  serversocket.bind(('', 8888))
7  serversocket.listen(5)
8
9  while True:
10     (clientsocket, address) = serversocket.accept() # 等待客户端连接
11     data = clientsocket.recv(4096)                 # 接收姓名
12     datetime = time.asctime() + '\n'
13     clientsocket.send('Hello ' + data)              # 发回问候
14     clientsocket.send('My time is ' + datetime)    # 发送服务器当前时间
15     clientsocket.close()                          # 关闭连接
----- hello-server.py

----- hello-client.py
20 # 省略 import 等
21 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22 sock.connect((sys.argv[1], 8888)) # 服务器地址由命令行指定
23 sock.send(os.getlogin() + '\n')   # 发送姓名
24 message = sock.recv(4096)         # 接收响应
25 print message                     # 打印结果
26 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 C22 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 CMake⁶ 为 build system，安装方法如下：

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

muduo 依赖于 Boost⁷，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
```

Linux 多线程服务端编程：使用 muduo C++ 网络库

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
|   |-- Timer.{h,cc}            简单测试
|   |-- TimerId.h              以下几个文件与定时器回调相关
|   |-- 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。

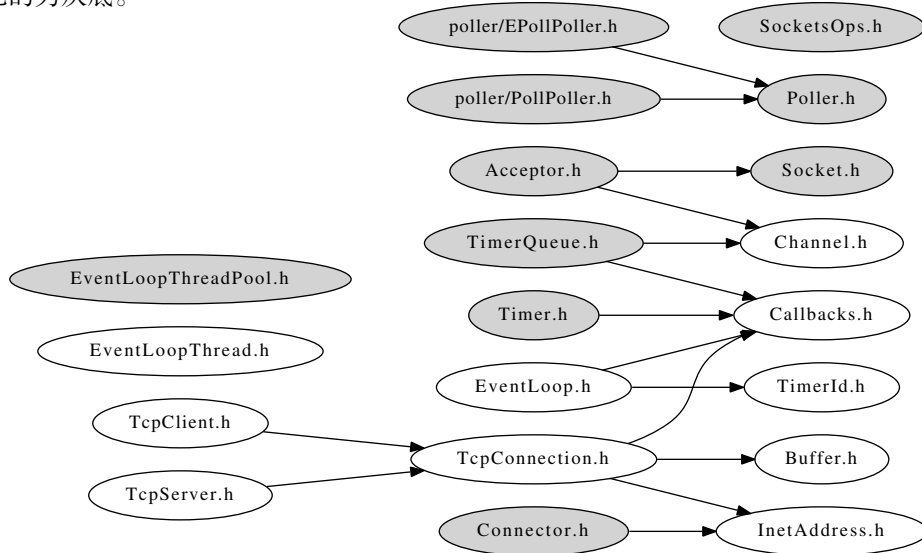


图 6-1

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。

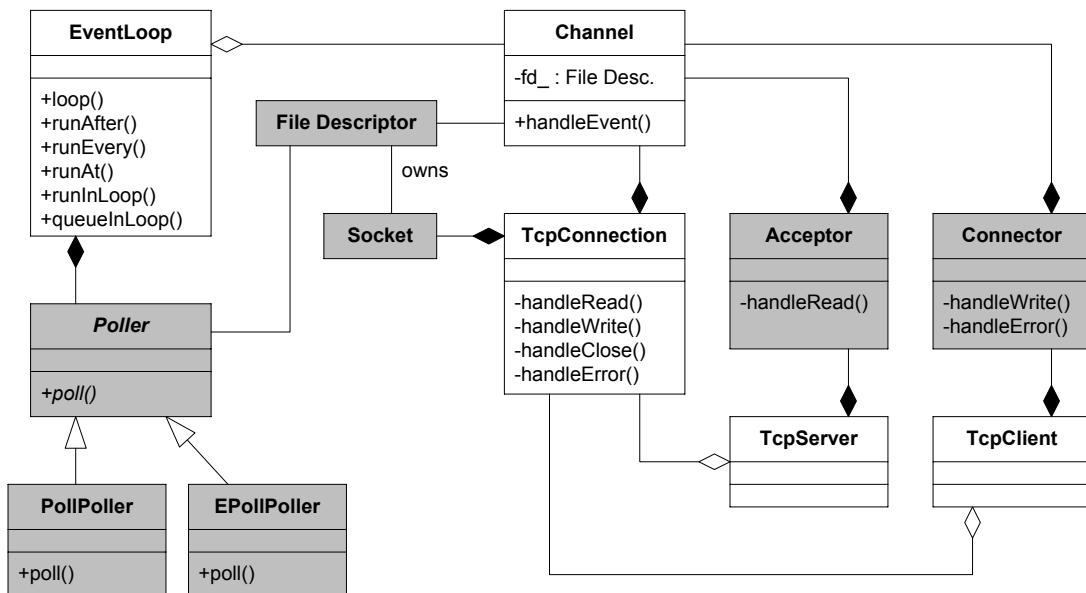


图 6-2

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 EventLoop-ThreadPool, 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 bug¹³。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。

```

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

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

```

L37 和 L40 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 L40 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。

```
1 #include "echo.h"
2
3 #include <muduo/base/Logging.h>
4 #include <muduo/net/EventLoop.h>
5
6 // using namespace muduo;
7 // using namespace muduo::net;
8
9 int main()
10 {
11     LOG_INFO << "pid = " << getpid();
12     muduo::net::EventLoop loop;
13     muduo::net::InetAddress listenAddr(2007);
14     EchoServer server(&loop, listenAddr);
15     server.start();
16     loop.loop();
17 }
```

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`。

Linux 多线程服务端编程: 使用 `muduo C++` 网络库

1. 拒绝连接。 什么都不做，程序空等。

```
1 #include <muduo/net/EventLoop.h>
2
3 using namespace muduo;
4 using namespace muduo::net;
5
6 int main()
7 {
8     EventLoop loop;
9     loop.loop();
10 }
```

examples/twisted/finger/finger01.cc

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。

```
1 #include <muduo/net/EventLoop.h>
2 #include <muduo/net/TcpServer.h>
3
4 using namespace muduo;
5 using namespace muduo::net;
6
7 int main()
8 {
9     EventLoop loop;
10    TcpServer server(&loop, InetAddress(1079), "Finger");
11    server.start();
12    loop.loop();
13 }
```

examples/twisted/finger/finger02.cc

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。

```
7 void onConnection(const TcpConnectionPtr& conn)
8 {
9     if (conn->connected())
10     {
11         conn->shutdown();
12     }
13 }
14
15 int main()
16 {
17     EventLoop loop;
18     TcpServer server(&loop, InetAddress(1079), "Finger");
19     server.setConnectionCallback(onConnection);
```

examples/twisted/finger/finger03.cc

```

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。

```

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

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 (L30)，在 UserMap 里查找，然后返回结果。安全问题同上。

```

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 L39。

```

--- 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。

Linux 多线程服务端编程：使用 muduo C++ 网络库

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>

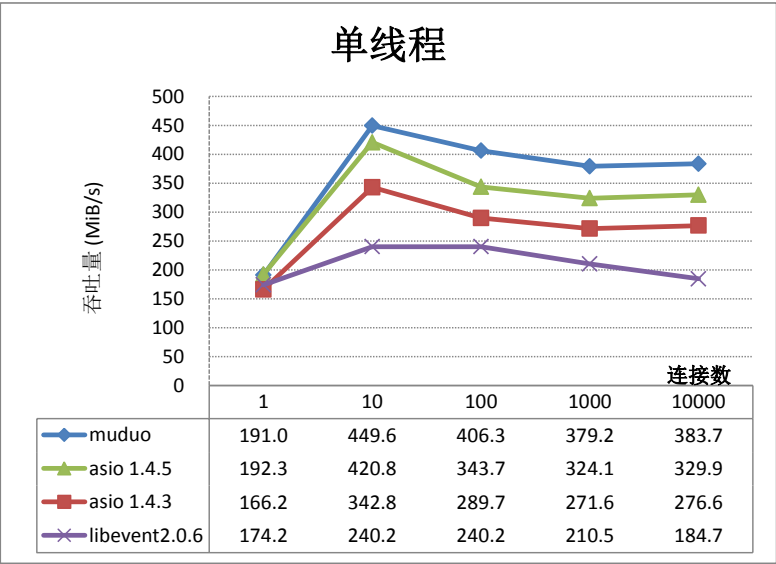


图 6-3

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 16384 字节，系统调用的性价比较高。

为了公平起见，我再测了一次，这回两个库都发送 4096 字节的消息（见图 6-4）。

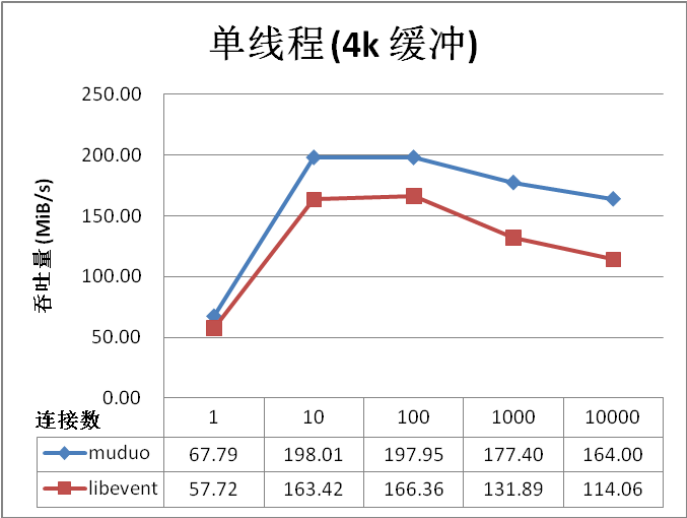


图 6-4

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。

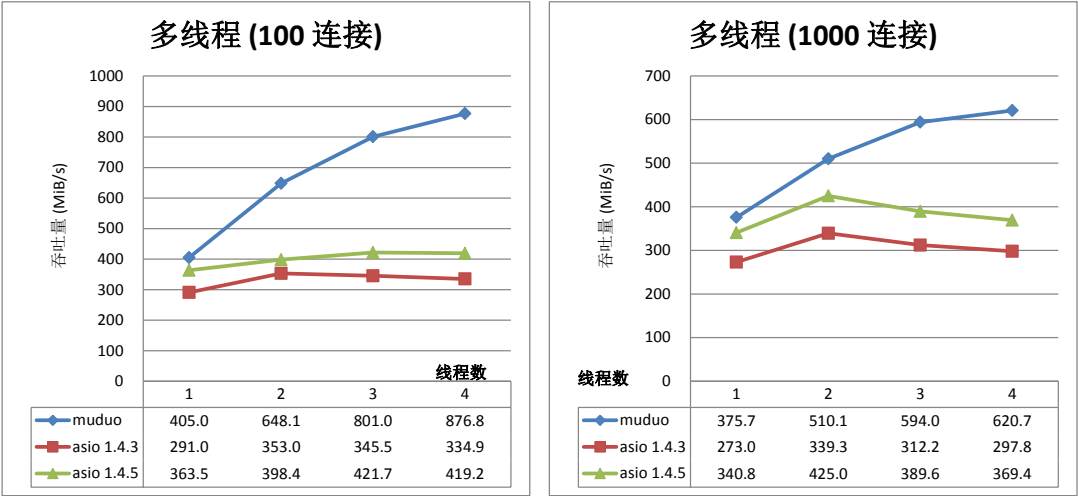


图 6-5

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`

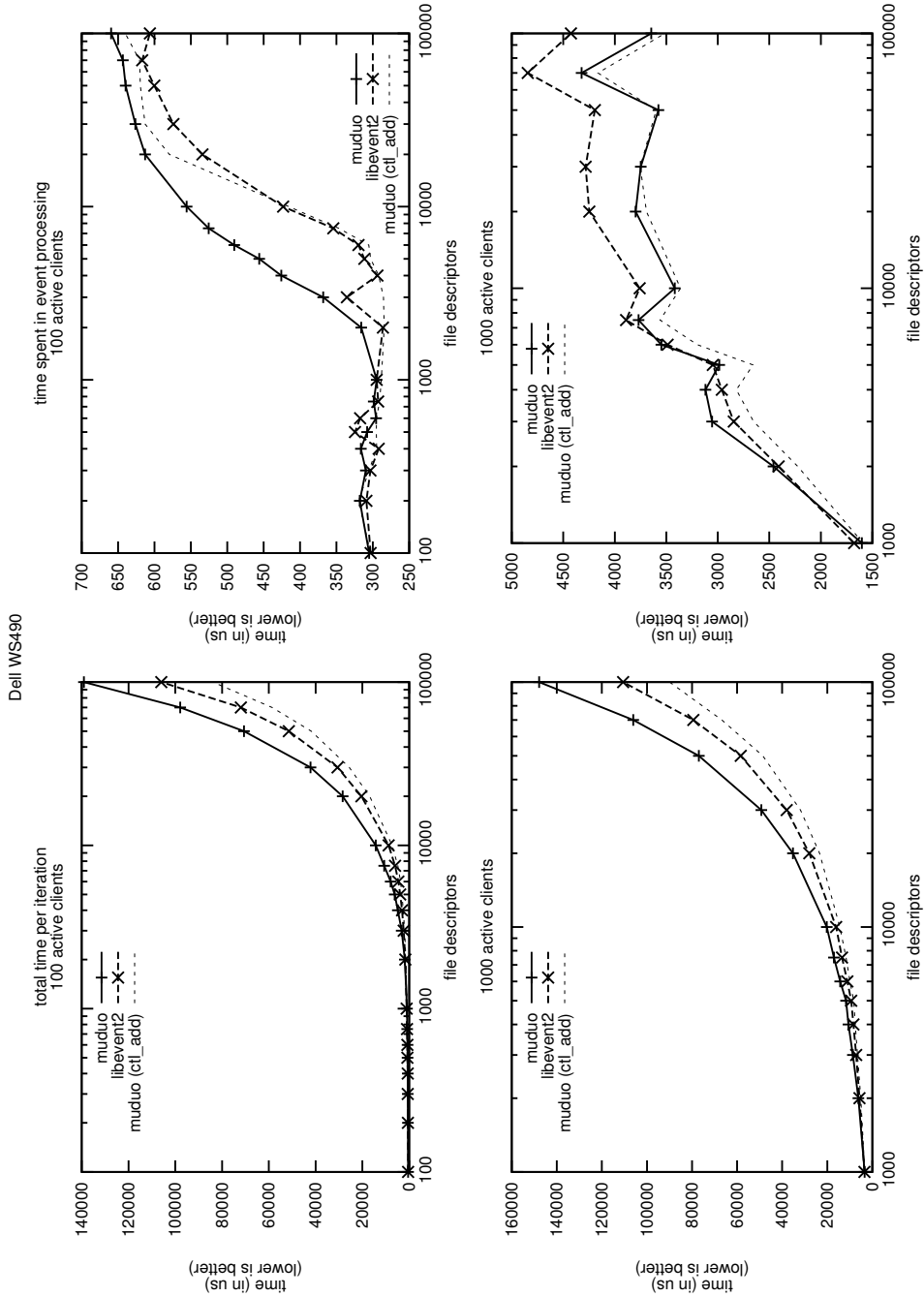


图 6-6

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>

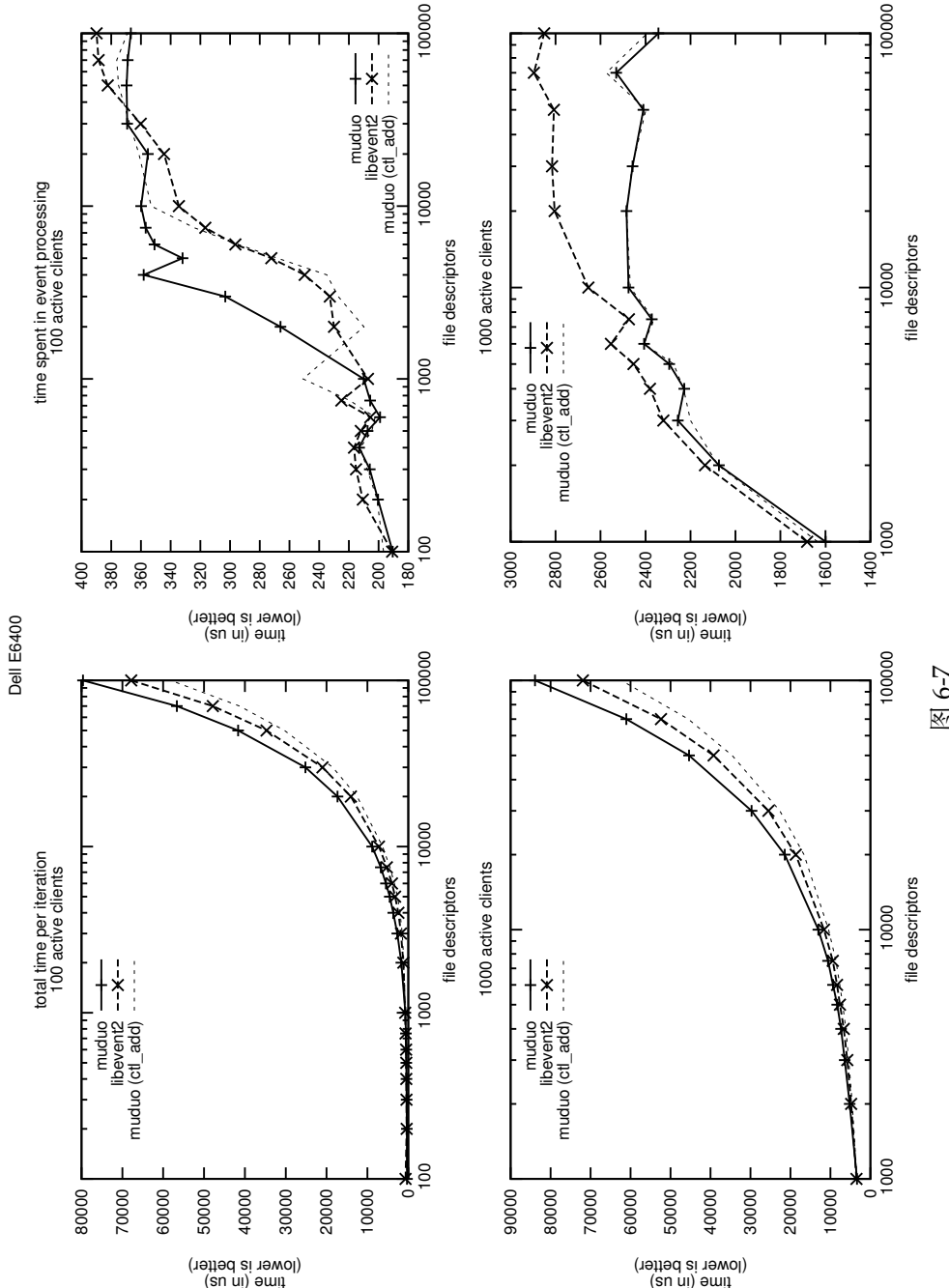


图 6-7

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。

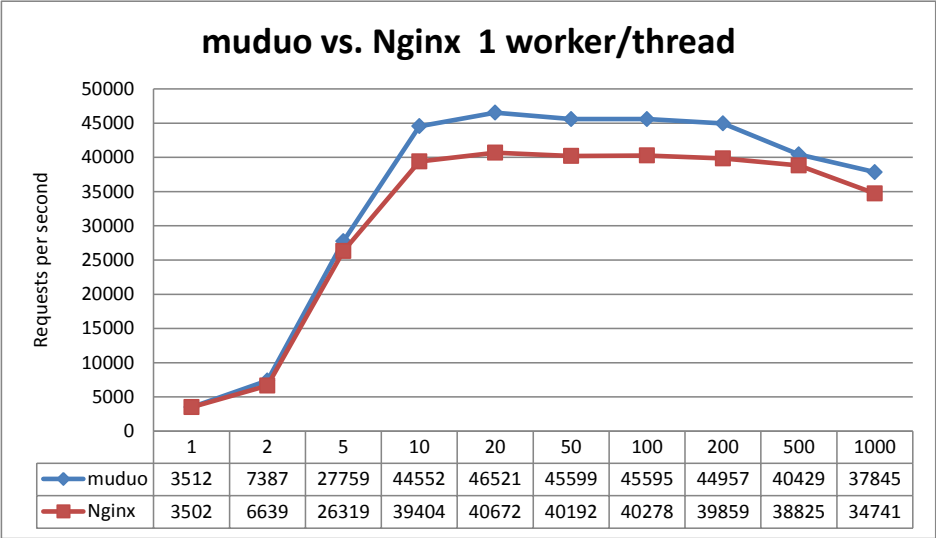


图 6-8

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。

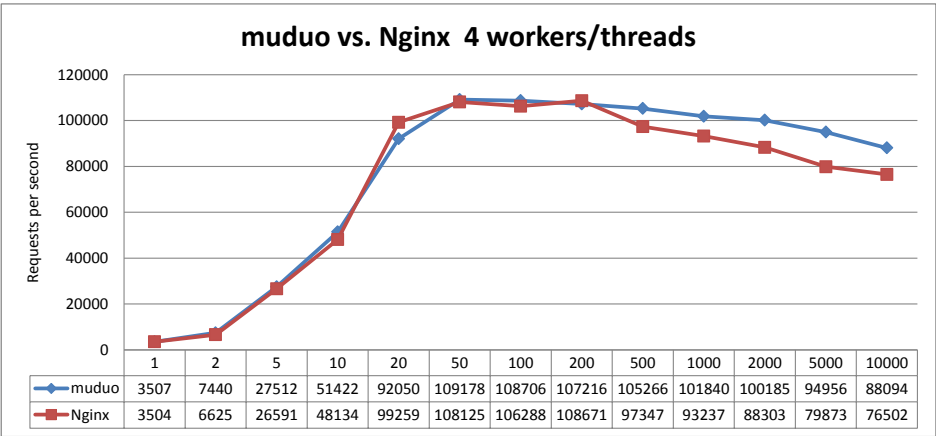


图 6-9

CPU 使用率对比（百分比是 top(1) 显示的数值）：

- 10000 并发连接，4 workers/threads，muduo 是 $4 \times 83\%$ ，Nginx 是 $4 \times 75\%$
- 1000 并发连接，4 workers/threads，muduo 是 $4 \times 85\%$ ，Nginx 是 $4 \times 78\%$

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

测试结果显示 muduo 多数情况下略快，Nginx 和 muduo 在合适的条件下 qps（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。

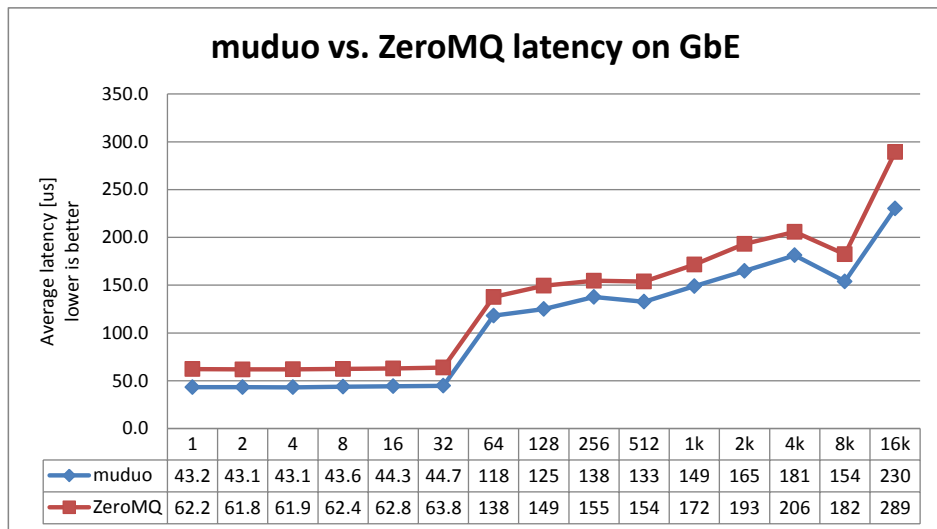


图 6-10

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：

```
00000001040000000002000000000050407008000300001090000300400200050100000000806000\r\n
```

响应：

```
693784512487512936125963874932651487568247391741398625319475268856129743274836159\r\n
```

例子 2 请求：

```
a:00000001040000000002000000000050407008000300001090000300400200050100000000806000\r\n
```

响应：

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

例子 3 请求：

```
b:00000001040000000002000000000050407008000300001090000300400200050100000000806005\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>

```

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

Linux 多线程服务端编程: 使用 muduo C++ 网络库

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）：

```

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。

```

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()`。

Linux 多线程服务端编程：使用 muduo C++ 网络库

```
$ 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 多线程 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（L28）等情况。首先定义一个从文件描述符到 socket 对象的映射（L14），程序的主体是一个事件循环（L15~L32），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（L16, L17）。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（L18~L23）。对于客户连接，则读取并回显数据，并处理连接的关闭（L24~L32）。对于 echo 服务而言，真正的业务逻辑只有 L28：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 L28~L30：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

23         elif event & select.POLLIN:
24             clientsocket = connections[fileno]
25             data = clientsocket.recv(4096)
26             if data:
27 -                 clientsocket.send(data) # sendall() partial?
28 +                 for (fd, othersocket) in connections.iteritems():
29 +                     if othersocket != clientsocket:
30 +                         othersocket.send(data) # sendall() partial?
31         else:
32             poll.unregister(fileno)
33             clientsocket.close()
34             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。）

```
1  LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
2  {
3      switch (message)
4      {
5          case WM_DESTROY:
6              PostQuitMessage(0);
7              return 0;
8              // many more cases
9      }
10     return DefWindowProc (hwnd, message, wParam, lParam) ;
11 }
```

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。

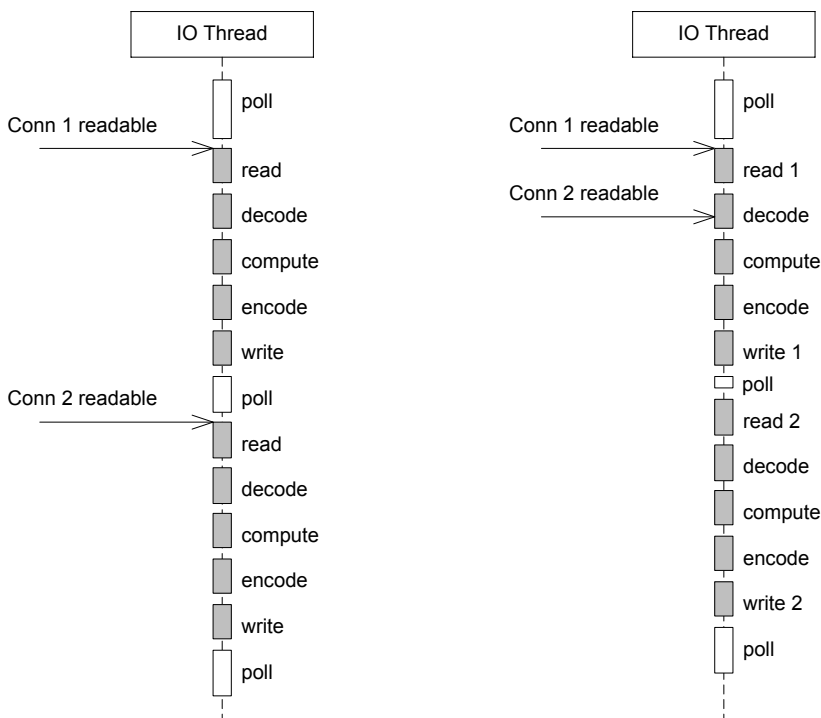


图 6-11

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 # server_socket.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](#)。

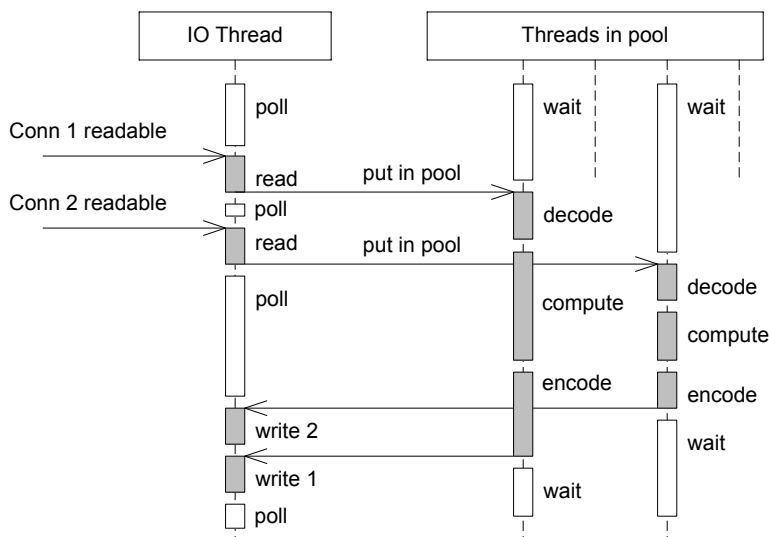


图 6-12

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>。

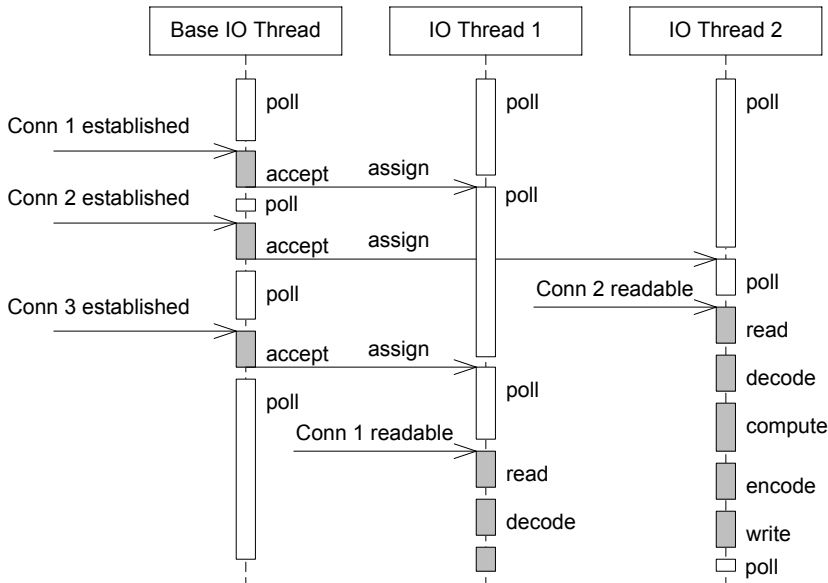


图 6-13

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。

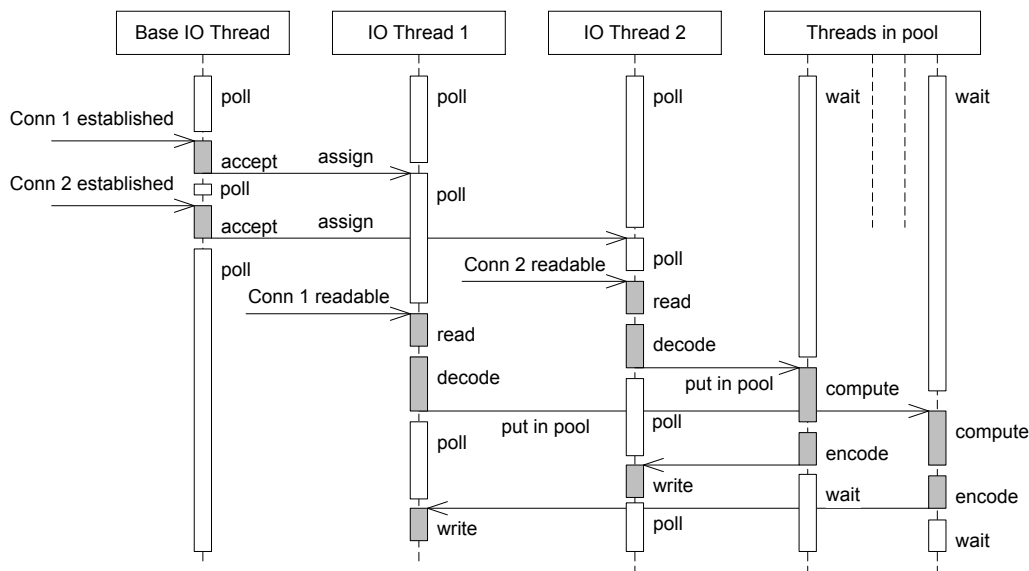


图 6-14

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，也不要再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 7 章

muduo 编程示例

本章将介绍如何用 muduo 网络库完成常见的 TCP 网络编程任务。内容如下：

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

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

7.1 五个简单 TCP 示例

本节将介绍五个简单 TCP 网络服务程序，包括 echo (RFC 862)、discard (RFC 863)、chargen (RFC 864)、daytime (RFC 867)、time (RFC 868) 这五个协议，以及 time 协议的客户端。各程序的协议简介如下。

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

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

discard

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

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

examples/simple/discard/discard.cc

与前面 p. 139 的 echo 服务相比，除了省略 namespace 外，关键的区别在于少了 L40：将收到的数据发回客户端。

剩下的都是例行公事的代码，此处从略，读者可对比参考 echo 服务。

daytime

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

```

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

```

L34 发送时间字符串，L35 主动断开连接。剩下的都是例行公事的代码，为节省篇幅，此处从略。

用 netcat 扮演客户端，运行结果如下：

```

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

```

time

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

```

examples/simple/time/time.cc
27 void TimeServer::onConnection(const muduo::net::TcpConnectionPtr& conn)
28 {
29     LOG_INFO << "TimeServer - " << conn->peerAddress().toIpPort() << " -> "
30             << conn->localAddress().toIpPort() << " is "
31             << (conn->connected() ? "UP" : "DOWN");
32     if (conn->connected())
33     {
34         time_t now = ::time(NULL);
35         int32_t be32 = sockets::hostToNetwork32(static_cast<int32_t>(now));
36         conn->send(&be32, sizeof be32);
37         conn->shutdown();
38     }
39 }
examples/simple/time/time.cc

```

L34、L35 取当前时间并转换为网络字节序 (Big Endian), L36 发送 32-bit 整数, L37 主动断开连接。剩下的都是例行公事的代码, 为节省篇幅, 此处从略。

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

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

time 客户端

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

```

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

```

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

这是我们第一次用到 TcpClient class, 完整的代码如下:

```

examples/simple/timeclient/timeclient.cc
17 class TimeClient : boost::noncopyable
18 {
19     public:
20         TimeClient(EventLoop* loop, const InetAddress& serverAddr)
21             : loop_(loop),
22               client_(loop, serverAddr, "TimeClient")

```

```

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

```

以上 L49 表示如果连接断开，就退出事件循环（L82），程序也就终止了。

```

72 int main(int argc, char* argv[])
73 {
74     LOG_INFO << "pid = " << getpid();
75     if (argc > 1)
76     {
77         EventLoop loop;
78         InetAddress serverAddr(argv[1], 2037);
79
80         TimeClient timeClient(&loop, serverAddr);
81         timeClient.connect();
82         loop.loop();
83     }
84     else
85     {
86         printf("Usage: %s host_ip\n", argv[0]);
87     }
88 }
89

```

examples/simple/timeclient/timeclient.cc

注意 `TcpConnection` 对象表示“一次”TCP 连接，连接断开之后不能重建。`TcpClient` 重试之后新建的连接会是另一个 `TcpConnection` 对象。

程序的运行结果如下（有折行），假设 `time server` 运行在本机：

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

echo

前面几个协议都是单向接收或发送数据，`echo` 是我们遇到的第一个双向的协议：服务端把客户端发过来的数据原封不动地传回去。它只需要关注“三个半事件”中的“消息/数据到达”事件，事件处理函数已在 p. 139 列出，这里复制一遍。

```

examples/simple/echo/echo.cc
33 void EchoServer::onMessage(const muduo::net::TcpConnectionPtr& conn,
34                             muduo::net::Buffer* buf,
35                             muduo::Timestamp time)
36 {
37     muduo::string msg(buf->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
```

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

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

练习 2：修改 `EchoServer::onMessage()`，实现 ROT13 加密¹。

¹ <http://en.wikipedia.org/wiki/ROT13>

chargen

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

```

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

```

L57 在连接建立时发生第一次数据；L73 继续发送数据。剩下的都是例行公事的代码，为节省篇幅，此处从略。

完整的 chargen 服务端还带流量统计功能，用到了定时器，我们会在 §7.8 介绍定时器的使用，到时候再回头来看相关代码。

用 netcat 扮演客户端，运行结果如下：

```

$ nc localhost 2019 | head
! "$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_`abcdefg h
"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_`abcdefg h
i
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_`abcdefg h
j
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_`abcdefg h
k
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_`abcdefg h
l
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_`abcdefg h
m

```



```
'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
)*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
```

五合一

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

```

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

```

examples/simple/allinone/allinone.cc

examples/simple/allinone/allinone.cc

这个例子充分展示了 Reactor 模式复用线程的能力，让一个单线程程序同时具备多个网络服务功能。一个容易想到的例子是 httpd 同时侦听 80 端口和 443 端口，另一个例子是程序中有多个 TcpClient，分别和数据库、Redis、Sudoku Solver 等后台服务打交道。对于初次接触这种编程模型的读者，值得跟踪代码运行的详细过程，弄清楚每个事件每个回调发生的时机与条件。

以上几个协议的消息格式都非常简单，没有涉及 TCP 网络编程中常见的分包处理，在后文 §7.3 讲 Boost.Asio 的聊天服务器时我们再来讨论这个问题。

Linux 多线程服务端编程：使用 muduo C++ 网络库

7.2 文件传输

本节用发送文件的例子来说明 `TcpConnection::send()` 的使用。到目前为止，我们用到了 `TcpConnection::send()` 的两个重载，分别是 `send(const string&)`² 和 `send(const void* message, size_t len)`³。

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

```

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

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

};

```

muduo/net/TcpConnection.h

在非阻塞网络编程中，发送消息通常是由网络库完成的，用户代码不会直接调用 `write(2)` 或 `send(2)` 等系统调用。原因见 p. 205 “`TcpConnection` 必须要有 `output buffer`”。在使用 `TcpConnection::send()` 时值得注意的有几点：

- `send()` 的返回类型是 `void`，意味着用户不必关心调用 `send()` 时成功发送了多少字节，`muduo` 库会保证把数据发送给对方。
- `send()` 是非阻塞的。意味着客户代码只管把一条消息准备好，调用 `send()` 来发送，即便 TCP 的发送窗口满了，也绝对不会阻塞当前调用线程。
- `send()` 是线程安全、原子的。多个线程可以同时调用 `send()`，消息之间不会混叠或交织。但是多个线程同时发送的消息 (s) 的先后顺序是不确定的，`muduo` 只能保证每个消息本身的完整性⁴。另外，`send()` 在多线程下仍然是非阻塞的。
- `send(const void* message, size_t len)` 这个重载最平淡无奇，可以发送任意字节序列。

² p. 139 L40。

³ p. 179, `time` 示例中的 L36。

⁴ 假设两个线程同时各自发送了一条任意长度的消息，那么这两条消息 a、b 的发送顺序要是先 a 后 b，要是先 b 后 a，不会出现“a 的前一半，b，a 的后一半”这种交织情况。

- `send(const StringPiece& message)` 这个重载可以发送 `std::string` 和 `const char*`，其中 `StringPiece`⁵ 是 Google 发明的专门用于传递字符串参数的 class，这样程序里就不必为 `const char*` 和 `const std::string&` 提供两份重载了。
- `send(Buffer*)` 有点特殊，它以指针为参数，而不是常见的 `const` 引用，因为函数中可能用 `Buffer::swap()` 来高效地交换数据，避免内存拷贝⁶，起到类似 C++ 右值引用的效果。
- 如果将来支持 C++11，那么可以增加对右值引用的重载，这样可以用 `move` 语义来避免内存拷贝。

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

如果不考虑并发，那么这个功能用 `netcat` 加重定向就能实现。这里展示的版本更加健壮，比方说发送 100MB 的文件，支持上万个并发客户连接；内存消耗只与并发连接数有关，跟文件大小无关；任何连接可以在任何时候断开，程序不会有内存泄漏或崩溃⁷。

我一共写了三个版本，代码位于 `examples/filetransfer`。

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

版本一

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

⁵ 代码位于 `muduo/base/StringPiece.h`。

⁶ 目前的实现尚未照此办理。

⁷ 我用 Java 实现了压力测试，代码位于 `examples/filetransfer/loadtest`。

```

const char* g_file = NULL;

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

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

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

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

```

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

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

版本二

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

```

examples/filetransfer/download2.cc
15 const int kBufSize = 64*1024;
16 const char* g_file = NULL;
17
18 void onConnection(const TcpConnectionPtr& conn)
19 {
20     LOG_INFO << "FileServer - " << conn->peerAddress().toIpPort() << " -> "
21             << conn->localAddress().toIpPort() << " is "
22             << (conn->connected() ? "UP" : "DOWN");
23     if (conn->connected())
24     {
25         LOG_INFO << "FileServer - Sending file " << g_file
26                 << " to " << conn->peerAddress().toIpPort();
27         conn->setHighWaterMarkCallback(onHighWaterMark, kBufSize+1);
28
29         FILE* fp = ::fopen(g_file, "rb");
30         if (fp)
31         {
32             conn->setContext(fp);
33             char buf[kBufSize];
34             size_t nread = ::fread(buf, 1, sizeof buf, fp);
35             conn->send(buf, nread);
36         }
37         else
38         {
39             conn->shutdown();
40             LOG_INFO << "FileServer - no such file";
41         }
42     }
43     else
44     {
45         if (!conn->getContext().empty())
46         {
47             FILE* fp = boost::any_cast<FILE*>(conn->getContext());
48             if (fp)
49             {
50                 ::fclose(fp);
51             }
52         }
53     }
54 }

```

在 `onWriteComplete()` 回调函数中读取下一块文件数据，继续发送。

```

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

```

examples/filetransfer/download2.cc

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

这个版本也存在一个问题，如果客户端故意只发起连接，不接收数据，那么要么把服务器进程的文件描述符耗尽，要么占用很多服务端内存（因为每个连接有 64KiB 的发送缓冲区）。解决办法可参考后文 §7.7 “限制服务器的最大并发连接数”和 §7.10 “用 `timing wheel` 踢掉空闲连接”。必须说明的是，`muduo` 并不是设计来编写面向公网的网络服务程序，这种服务程序需要在安全性方面下很多工夫，我个人对此不在行，我更关心实现内网（不一定是局域网）的高效服务程序。

版本三

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

```

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

void onConnection(const TcpConnectionPtr& conn)
{

```

examples/filetransfer/download3.cc

```

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

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

```

examples/filetransfer/download3.cc

以上代码体现了现代 C++ 的资源管理思路，即无须手动释放资源，而是通过将资源与对象生命期绑定，在对象析构的时候自动释放资源，从而把资源管理转换为对象生命期管理，而后者是早已解决了的问题。这正是 C++ 最重要的编程技法：RAII。

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

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

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

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

void TcpConnection::shutdownInLoop()
{
    loop_->assertInLoopThread();
    if (!channel_->isWriting()) // 如果当前没有发送数据
    {
        // we are not writing
        socket_->shutdownWrite(); // 调用 Socket::shutdownWrite()
    }
}

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

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

笔者答复如下：

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

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

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

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

练习：阅读代码，回答“如果被动关闭连接，`muduo` 的行为如何？”⁸

另外，如果当前 `output buffer` 里还有数据尚未发出的话，`muduo` 也不会立刻调用 `shutdownWrite`，而是等到数据发送完毕再 `shutdown`，可以避免对方漏收数据。

```

252 void TcpConnection::handleWrite()
253 {
254     loop_>assertInLoopThread();
255     if (channel_>isWriting())
256     {
257         ssize_t n = sockets::write(channel_>fd(),           // 思考：为什么只写一次？
258                                   outputBuffer_.peek(),
259                                   outputBuffer_.readableBytes());
260         if (n > 0)
261         {
262             outputBuffer_.retrieve(n);
263             if (outputBuffer_.readableBytes() == 0)
264             {
265                 channel_>disableWriting();
266                 if (writeCompleteCallback_)
267                 {
268                     loop_>queueInLoop(boost::bind(writeCompleteCallback_,
269                                                     shared_from_this()));
270                 }
271                 if (state_ == kDisconnecting)
272                 {
273                     shutdownInLoop();
274                 }
275             }
276         }
277     }
278 }

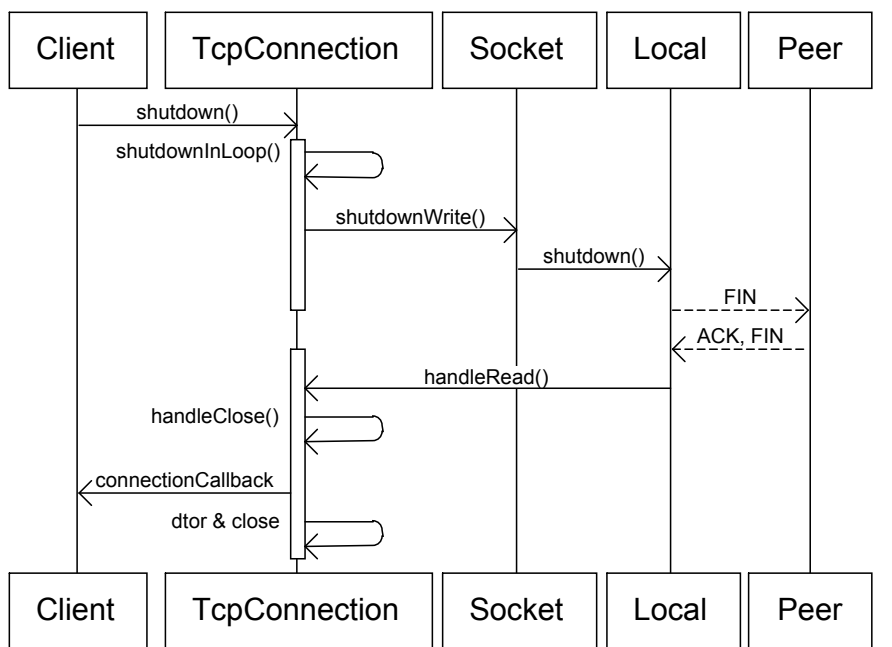
```

muduo/net/TcpConnection.cc

⁸ 提示：在 `read(2)` 返回 0 的时候会回调 `connection callback`，这样客户代码就知道对方断开连接了。

muduo 这种关闭连接的方式对对方也有要求，那就是对方 `read()` 到 0 字节之后会主动关闭连接（无论 `shutdownWrite()` 还是 `close()`），一般的网络程序都会这样，不是什么问题。当然，这么做有一个潜在的安全漏洞，万一对方故意不关闭连接，那么 muduo 的连接就一直半开着，消耗系统资源。必要时可以调用 `TcpConnection::handleClose()` 来强行关闭连接，这需要将 `handleClose()` 改为 `public` 成员函数。

完整的流程见图 7-1。我们发完了数据，于是 `shutdownWrite`，发送 TCP FIN 分节，对方会读到 0 字节，然后对方通常会关闭连接。这样 muduo 会读到 0 字节，然后 muduo 关闭连接。（思考题：在 `shutdown()` 之后，muduo 回调 `connection callback` 的时间间隔大约是一个 `round-trip time`，为什么？）



www.websequencediagrams.com

图 7-1

如果有必要，对方可以在 `read()` 返回 0 之后继续发送数据，这是直接利用了 `half-close TCP` 连接。muduo 不会漏收这些数据。

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

muduo 在 `read()` 返回 0 的时候会回调 `connection callback`, `TcpServer` 或 `TcpClient` 把 `TcpConnection` 的引用计数减一。如果引用计数降到零, 则表明用户代码也不持有 `TcpConnection`, 它就会析构了。

参考: 《TCP/IP 详解》[TCPv1] 18.5 节 “TCP Half-Close” 和《UNIX 网络编程 (第 3 版)》[UNP] 6.6 节 “`shutdown()` 函数”。

在网络编程中, 应用程序发送数据往往比接收数据简单 (实现非阻塞网络库正相反, 发送比接收难), 下一节我们再谈接收并解析消息的要领。

7.3 Boost.Asio 的聊天服务器

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

7.3.1 TCP 分包

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

对于短连接的 TCP 服务, 分包不是一个问题, 只要发送方主动关闭连接, 就表示一条消息发送完毕, 接收方 `read()` 返回 0, 从而知道消息的结尾。例如 §7.1 里的 `daytime` 和 `time` 协议。

对于长连接的 TCP 服务, 分包有四种方法:

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

在后文的代码讲解中还会仔细讨论用长度字段分包的常见陷阱。

聊天服务

本节实现的聊天服务非常简单，由服务端程序和客户端程序组成，协议如下：

- 服务端程序在某个端口侦听（listen）新的连接。
- 客户端向服务端发起连接。
- 连接建立之后，客户端随时准备接收服务端的消息并在屏幕上显示出来。
- 客户端接受键盘输入，以回车为界，把消息发送给服务端。
- 服务端接收到消息之后，依次发送给每个连接到它的客户端；原来发送消息的客户端进程也会收到这条消息。
- 一个服务端进程可以同时服务多个客户端进程。当有消息到达服务端后，每个客户端进程都会收到同一条消息，服务端广播发送消息的顺序是任意的，不一定哪个客户端会先收到这条消息。
- （可选）如果消息 A 先于消息 B 到达服务端，那么每个客户端都会先收到 A 再收到 B。

这实际上是一个简单的基于 TCP 的应用层广播协议，由服务端负责把消息发送给每个连接到它的客户端。参与“聊天”的既可以是人，也可以是程序。在后文 §7.11 中，我将介绍一个稍微复杂一点的例子 hub，它有“聊天室”的功能，客户端可以注册特定的 topic(s)，并往某个 topic 发送消息，这样代码更有意思。

我在“谈一谈网络编程学习经验”（附录 A）中把聊天服务列为“最主要的三个例子”之一，其与前面的“五个简单 TCP 协议”不同，聊天服务的特点是“连接之间的数据有交流，从 a 连接收到的数据要发给 b 连接。这样对连接管理提出了更高的要求：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？”muduo 的这个例子充分展示了解决以上问题的手法。

7.3.2 消息格式

本聊天服务的消息格式非常简单，“消息”本身是一个字符串，每条消息有一个 4 字节的头部，以网络序存放字符串的长度。消息之间没有间隙，字符串也不要求以 '\0' 结尾。比方说有两条消息“hello”和“chenshuo”，那么打包后的字节流共有 21 字节：

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

Linux 多线程服务端编程：使用 muduo C++ 网络库

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

```

55 void send(muduo::net::TcpConnection* conn,
56           const muduo::StringPiece& message)
57 {
58     muduo::net::Buffer buf;
59     buf.append(message.data(), message.size());
60     int32_t len = static_cast<int32_t>(message.size());
61     int32_t be32 = muduo::net::sockets::hostToNetwork32(len);
62     buf.prepend(&be32, sizeof be32);
63     conn->send(&buf);
64 }

```

examples/asio/chat/codec.h

`muduo Buffer` 有一个很好的功能，它在头部预留了 8 个字节的空间，这样 `L62` 的 `prepend()` 操作就不需要移动已有的数据，效率较高。

分包的代码 解析数据往往比生成数据更复杂，分包、打包也不例外。

```

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

```

examples/asio/chat/codec.h

onMessage() 中 L43 构造完整的消息, L44 通过 messageCallback_ 回调用户代码。L32 有潜在的问题, 在某些不支持非对齐内存访问的体系结构上会造成 SIGBUS core dump, 读取消息长度应该改用 Buffer::peekInt32()。上面这段代码的 L28 用了 while 循环来反复读取数据, 直到 Buffer 中的数据不够一条完整的消息。请读者思考, 如果换成 if (buf->readableBytes() >= kHeaderLen) 会有什么后果。

以前面提到的两条消息的字节流为例:

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

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

1. 每次收到一个字节的数据, onMessage() 被调用 21 次;
2. 数据分两次到达, 第一次收到 2 个字节, 不足消息的长度字段;
3. 数据分两次到达, 第一次收到 4 个字节, 刚好够长度字段, 但是没有 body;
4. 数据分两次到达, 第一次收到 8 个字节, 长度完整, 但 body 不完整;
5. 数据分两次到达, 第一次收到 9 个字节, 长度完整, body 也完整;
6. 数据分两次到达, 第一次收到 10 个字节, 第一条消息的长度完整、body 也完整, 第二条消息长度不完整;
7. 请自行移动和增加分割点, 验证各种情况; 一共有超过 100 万种可能 ($2^{21}-1$)。
8. 数据一次就全部到达, 这时必须用 while 循环来读出两条消息, 否则消息会堆积在 Buffer 中。

请读者验证 onMessage() 是否做到了以上几点。这个例子充分说明了 non-blocking read 必须和 input buffer 一起使用。而且在写 decoder 的时候一定要在收到完整的消息 (L40) 之后再 retrieve 整条消息 (L42 和 L45), 除非接收方使用复杂的状态机来解码。

7.3.3 编解码器 LengthHeaderCodec

有人评论 muduo 的接收缓冲区不能设置回调函数的触发条件⁹, 确实如此。每当 socket 可读时, muduo 的 TcpConnection 会读取数据并存入 input buffer, 然后回调用户的函数。不过, 一个简单的间接层就能解决问题, 让用户代码只关心“消息到达”而不是“数据到达”, 如本例中的 LengthHeaderCodec 所展示的那样。

⁹ <http://www.cnblogs.com/Solstice/archive/2011/02/02/1948839.html#2022206>

```

12 class LengthHeaderCode : boost::noncopyable
13 {
14 public:
15     typedef boost::function<void (const muduo::net::TcpConnectionPtr&,
16                                   const muduo::string& message,
17                                   muduo::Timestamp)> StringMessageCallback;
18
19     explicit LengthHeaderCode(const StringMessageCallback& cb)
20         : messageCallback_(cb)
21     {
22     }

    onMessage() 和 send() 同前。

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

```

examples/asio/chat/codec.h

这段代码把以 `Buffer*` 为参数的 `MessageCallback` 转换成了以 `const string&` 为参数的 `StringMessageCallback`，让用户代码不必关心分包操作，具体的调用时序图见 p. 231 图 7-29。如果编程语言相同，客户端和服务端可以（应该）共享同一个 `codec`，这样既节省工作量，又避免因对协议理解不一致而导致的错误。

7.3.4 服务端的实现

聊天服务器的服务端代码小于 100 行，不到 `asio` 的一半。

请先阅读 p. 200 L65~L69 的数据成员的定义。除了经常见到的 `EventLoop` 和 `TcpServer`，`ChatServer` 还定义了 `codec_` 和 `connections_` 作为成员，后者存放目前已建立的客户连接。在收到消息之后，服务器会遍历整个容器，把消息广播给其中的每一个 TCP 连接（`onStringMessage()`）。

首先，在构造函数里注册回调：

```

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

```

examples/asio/chat/server.cc

```

25     server_.setConnectionCallback(
26         boost::bind(&ChatServer::onConnection, this, _1));
27     server_.setMessageCallback(
28         boost::bind(&LengthHeaderCode::onMessage, &codec_, _1, _2, _3));
29 }
30
31 void start()
32 {
33     server_.start();
34 }

```

examples/asio/chat/server.cc

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

另外，server_.start() 绝对不能在构造函数里调用，这么做将来会有线程安全的问题，见 §1.2 的论述。

以下是处理连接的建立和断开的代码，注意它把新建的连接加入到 connections_ 容器中，把已断开的连接从容器中删除。这么做是为了避免内存和资源泄漏，TcpConnectionPtr 是 boost::shared_ptr<TcpConnection>，是 muduo 里唯一一个默认采用 shared_ptr 来管理生命期的对象。§4.7 谈了这么做的原因。

```

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

```

examples/asio/chat/server.cc

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


```

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

```

数据成员：

```

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

```

examples/asio/chat/server.cc

main() 函数中是例行公事的代码：

```

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

```

examples/asio/chat/server.cc

examples/asio/chat/server.cc

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

7.3.5 客户端的实现

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

我用了两个线程：main() 函数所在的线程负责读键盘，另外用一个 EventLoopThread 来处理网络 IO。¹⁰

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

examples/asio/chat/client.cc

```

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

```

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

```

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

```

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

```

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

```

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

¹⁰ 我暂时没有把标准输入输出融入 Reactor 的想法，因为服务器程序很少用到 stdin 和 stdout。

```

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

```

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

```

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

```

数据成员：

```

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

```

examples/asio/chat/client.cc

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

```

83 int main(int argc, char* argv[])
84 {
85     LOG_INFO << "pid = " << getpid();
86     if (argc > 2)
87     {
88         EventLoopThread loopThread;
89         uint16_t port = static_cast<uint16_t>(atoi(argv[2]));
90         InetAddress serverAddr(argv[1], port);
91
92         ChatClient client(loopThread.startLoop(), serverAddr);
93         client.connect();

```

examples/asio/chat/client.cc

```
94     std::string line;
95     while (std::getline(std::cin, line))
96     {
97         client.write(line);
98     }
99     client.disconnect();
100 }
101 else
102 {
103     printf("Usage: %s host_ip port\n", argv[0]);
104 }
105 }
```

examples/asio/chat/client.cc

L92 ChatClient 使用 EventLoopThread 的 EventLoop，而不是通常的主线程的 EventLoop。L97 发送数据行。

简单测试

打开三个命令行窗口，在第一个窗口运行：

```
$ ./asio_chat_server 3000
```

在第二个窗口运行：

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

在第三个窗口运行同样的命令：

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

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

代码示例中还有另外三个 server 程序，都是多线程的，详细介绍在 p. 260。

- server_threaded.cc 使用多线程 TcpServer，并用 mutex 来保护共享数据。
- server_threaded_efficient.cc 对共享数据以 §2.8 “借 shared_ptr 实现 copy-on-write”的手法来降低锁竞争。
- server_threaded_highperformance.cc 采用 thread local 变量，实现多线程高效转发，这个例子值得仔细阅读理解。

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

7.4 muduo Buffer 类的设计与使用

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

7.4.1 muduo 的 IO 模型

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

C10k 问题¹¹的页面介绍了五种 IO 策略，把线程也纳入考量。（现在 C10k 已经不是什么问题，C100k 也不是大问题，C1000k 才算得上挑战）。

在这个多核时代，线程是不可避免的。那么服务端网络编程该如何选择线程模型呢？我赞同 libev 作者的观点¹²：one loop per thread is usually a good model。之前我也不止一次表述过这个观点，参见 §3.3 “多线程服务器的常用编程模型”和 §6.6 “详解 muduo 多线程模型”。

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

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

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

¹¹ <http://www.kegel.com/c10k.html>

¹² http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#THREADS_AND_COROUTINES

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

当然，non-blocking 编程比 blocking 难得多，见 §6.4.1 “TCP 网络编程本质论”列举的难点。基于 event loop 的网络编程跟直接用 C/C++ 编写单线程 Windows 程序颇为相像：程序不能阻塞，否则窗口就失去响应了；在 event handler 中，程序要尽快交出控制权，返回窗口的事件循环。

7.4.2 为什么 non-blocking 网络编程中应用层 buffer 是必需的

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

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

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

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

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

库不能立刻关闭连接，而要等数据发送完毕，见 p. 191 “为什么 `TcpConnection::shutdown()` 没有直接关闭 TCP 连接”中的讲解。

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

TcpConnection 必须要有 input buffer TCP 是一个无边界的字节流协议，接收方必须要处理“收到的数据尚不构成一条完整的消息”和“一次收到两条消息的数据”等情况。一个常见的场景是，发送方 `send()` 了两条 1kB 的消息（共 2kB），接收方收到数据的情况可能是：

- 一次性收到 2kB 数据；
- 分两次收到，第一次 600B，第二次 1400B；
- 分两次收到，第一次 1400B，第二次 600B；
- 分两次收到，第一次 1kB，第二次 1kB；
- 分三次收到，第一次 600B，第二次 800B，第三次 600B；
- 其他任何可能。一般而言，长度为 n 字节的消息分块到达的可能性有 2^{n-1} 种。

网络库在处理“socket 可读”事件的时候，必须一次性把 socket 里的数据读完（从操作系统 buffer 搬到应用层 buffer），否则会反复触发 `POLLIN` 事件，造成 `busy-loop`。那么网络库必然要应对“数据不完整”的情况，收到的数据先放到 `input buffer` 里，等构成一条完整的消息再通知程序的业务逻辑。这通常是 `codec` 的职责，见 §7.3 “Boost.Asio 的聊天服务器”中的“TCP 分包”的论述与代码。所以，在 TCP 网络编程中，网络库必须要给每个 TCP connection 配置 `input buffer`。

`muduo EventLoop` 采用的是 `epoll(4) level trigger`，而不是 `edge trigger`。一是为了与传统的 `poll(2)` 兼容，因为在文件描述符数目较少，活动文件描述符比例较高时，`epoll(4)` 不见得比 `poll(2)` 更高效¹³，必要时可以在进程启动时切换 `Poller`。二是 `level trigger` 编程更容易，以往 `select(2)/poll(2)` 的经验都可以继续用，不可能发生漏掉事件的 bug。三是读写的时候不必等候出现 `EAGAIN`，可以节省系统调用次数，降低延迟。

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

¹³ <http://sheddingbikes.com/posts/1280829388.html>

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

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

对于网络程序来说, 一个简单的验收测试是: 输入数据每次收到一个字节 (200 字节的输入数据会分 200 次收到, 每次间隔 10ms), 程序的功能不受影响。对于 muduo 程序, 通常可以用 `codec` 来分离“消息接收”与“消息处理”, 见 §7.6 “在 muduo 中实现 Protobuf 编解码器与消息分发器”对“编解码器 `codec`”的介绍。

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

7.4.3 Buffer 的功能需求

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

muduo Buffer 的设计要点:

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

Buffer 其实像是一个 `queue`, 从末尾写入数据, 从头部读出数据。

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

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

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

图 7-2 是 `muduo::net::Buffer` 的类图。请注意, 为了后面画图方便, 这个类图跟实际代码略有出入, 但不影响我要表达的观点。代码位于 `muduo/net/Buffer.{h,cc}`。

Buffer
-data: vector<char> -readIndex: int -writeIndex: int
+readableBytes(): int +peek(): const char* +retrieve(int) +retrieveAsString(): string +append(const void*, int) +prepend(const void*, int) +swap(Buffer&) -readFd(int): int

图 7-2

本节不介绍每个成员函数的使用，而会详细讲解 readIndex 和 writeIndex 的作用。

Buffer::readFd() 我在 p. 138 写道：

在非阻塞网络编程中，如何设计并使用缓冲区？一方面我们希望减少系统调用，一次读的数据越多越划算，那么似乎应该准备一个大的缓冲区。另一方面希望减少内存占用。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。

具体做法是，在栈上准备一个 65 536 字节的 extrabuf，然后利用 readv() 来读取数据，iovec 有两块，第一块指向 muduo Buffer 中的 writable 字节，另一块指向栈上的 extrabuf。这样如果读入的数据不多，那么全部都读到 Buffer 中去了；如果长度超过 Buffer 的 writable 字节数，就会读到栈上的 extrabuf 里，然后程序再把 extrabuf 里的数据 append() 到 Buffer 中，代码见 §8.7.2。

这么做利用了临时栈上空间¹⁴，避免每个连接的初始 Buffer 过大造成的内存浪费，也避免反复调用 read() 的系统开销（由于缓冲区足够大，通常一次 readv() 系统调用就能读全部数据）。由于 muduo 的事件触发采用 level trigger，因此这个函数并不会反复调用 read() 直到其返回 EAGAIN，从而可以降低消息处理的延迟。

这算是一个小小的创新吧。

¹⁴ readFd() 是最内层函数，其在每个 IO 线程的最大 stack 空间开销是固定的 64KiB，与连接数目无关。如果 stack 空间紧张，也可以改用 thread local 的 extrabuf，但是不能全局共享一个 extrabuf。（为什么？）

线程安全？ `muduo::net::Buffer` 不是线程安全的（其安全性跟 `std::vector` 相同），这么设计的理由如下：

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

代码中用 `EventLoop::assertInLoopThread()` 保证以上假设成立。

如果 `TcpConnection::send()` 调用发生在该 `TcpConnection` 所属的那个 IO 线程，那么它会转而调用 `TcpConnection::sendInLoop()`，`sendInLoop()` 会在当前线程（也就是 IO 线程）操作 output buffer；如果 `TcpConnection::send()` 调用发生在别的线程，它不会在当前线程调用 `sendInLoop()`，而是通过 `EventLoop::runInLoop()` 把 `sendInLoop()` 函数调用转移到 IO 线程（听上去颇为神奇？），这样 `sendInLoop()` 还是会在 IO 线程操作 output buffer，不会有线程安全问题。当然，跨线程的函数转移调用涉及函数参数的跨线程传递，一种简单的做法是把数据拷贝一份，绝对安全。

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

7.4.4 Buffer 的数据结构

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

在介绍 `Buffer` 的数据结构之前，先简单说一下后面示意图中表示指针或下标的箭头所指位置的具体含义。对于长度为 10 的字符串 `"Chen Shuo\n"`，如果 `p0` 指向第 0 个字符（白色区域的开始），`p1` 指向第 5 个字符（灰色区域的开始），`p2` 指向 `'\n'` 之后的那个位置（通常是 `end()` 迭代器所指的位置），那么精确的画法如图 7-3 的左图所示，简略的画法如图 7-3 的右图所示，后文都采用这种简略画法。

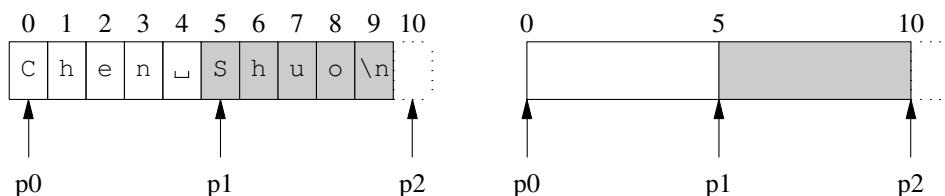


图 7-3

muduo Buffer 的数据结构如图 7-4 所示。

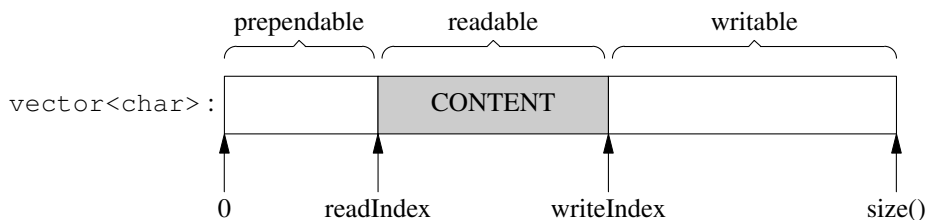


图 7-4

两个 index 把 vector 的内容分为三块：prependable、readable、writable，各块的大小见式 7-1。灰色部分是 Buffer 的有效载荷（payload），prependable 的作用留到后面讨论。

$$\begin{aligned}
 \text{prependable} &= \text{readIndex} \\
 \text{readable} &= \text{writeIndex} - \text{readIndex} \\
 \text{writable} &= \text{size}() - \text{writeIndex}
 \end{aligned} \tag{7-1}$$

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

$$0 \leq \text{readIndex} \leq \text{writeIndex} \leq \text{size}()$$

muduo Buffer 里有两个常数 kCheapPrepend 和 kInitialSize，定义了 prependable 的初始大小和 writable 的初始大小，readable 的初始大小为 0。在初始化之后，Buffer 的数据结构如图 7-5 所示，其中括号里的数字是该变量或常量的值。

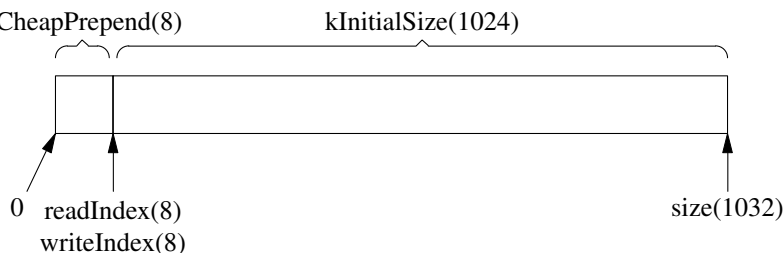


图 7-5

根据以上公式（见式 7-1）可算出各块的大小，刚刚初始化的 Buffer 里没有 payload 数据，所以 readable == 0。

7.4.5 Buffer 的操作

基本的 read-write cycle

Buffer 初始化后的情况见图 7-4。如果向 Buffer 写入了 200 字节，那么其布局如图 7-6 所示。

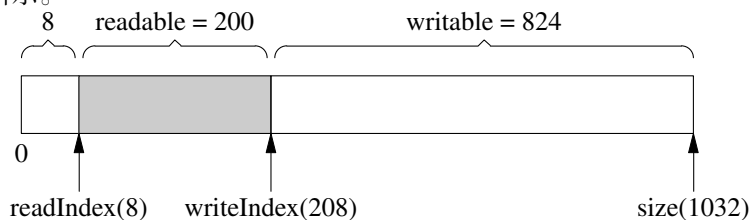


图 7-6

图 7-6 中 writeIndex 向后移动了 200 字节，readIndex 保持不变，readable 和 writable 的值也有变化。

如果从 Buffer read() & retrieve()（下称“读入”）了 50 字节，结果如图 7-7 所示。与图 7-6 相比，readIndex 向后移动 50 字节，writeIndex 保持不变，readable 和 writable 的值也有变化（这句话往后从略）。

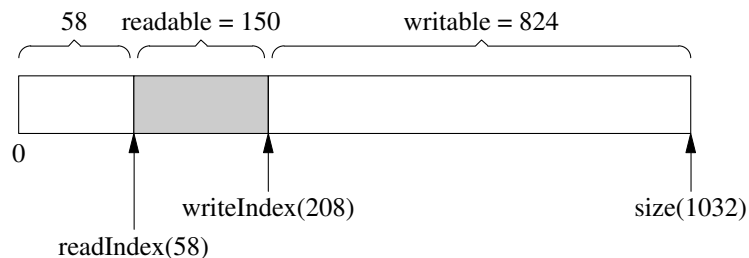


图 7-7

然后又写入了 200 字节，writeIndex 向后移动了 200 字节，readIndex 保持不变，如图 7-8 所示。

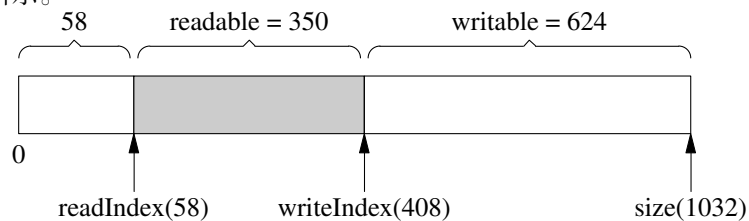


图 7-8

接下来，一次性读入 350 字节，请注意，由于全部数据读完了，readIndex 和

writeIndex 返回原位以备新一轮使用（见图 7-9），这和图 7-5 是一样的。

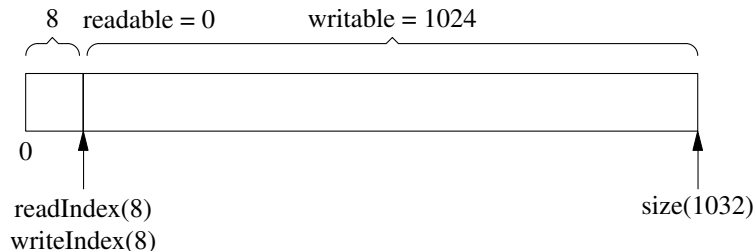


图 7-9

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

自动增长

muduo Buffer 不是固定长度的，它可以自动增长，这是使用 vector 的直接好处。假设当前的状态如图 7-10 所示。（这和前面的图 7-8 是一样的。）

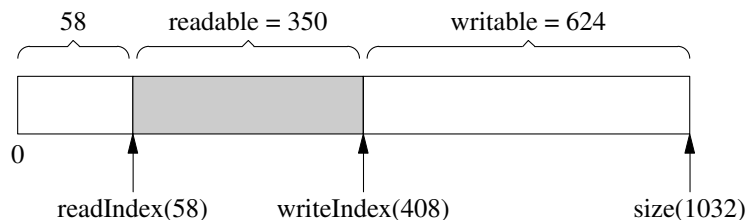


图 7-10

客户代码一次性写入 1000 字节，而当前可写的字节数只有 624，那么 buffer 会自动增长以容纳全部数据，得到的结果如图 7-11 所示。注意 readIndex 返回到了前面，以保持 prependable 等于 kCheapPrependable。由于 vector 重新分配了内存，原来指向其元素的指针会失效，这就是为什么 readIndex 和 writeIndex 是整数下标而不是指针。（注意：在目前的实现中 prependable 会保持 58 字节，留待将来修正。）

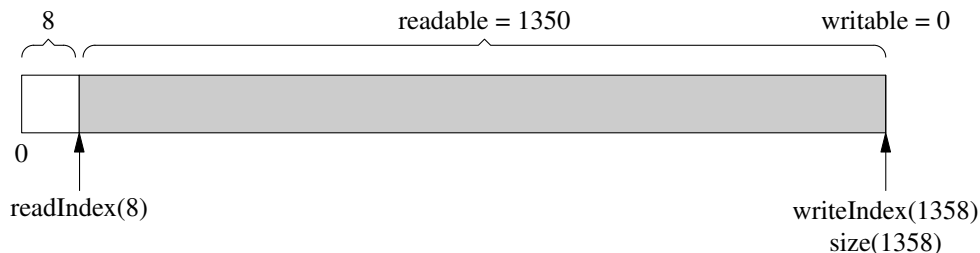


图 7-11

然后读入 350 字节，readIndex 前移，如图 7-12 所示。

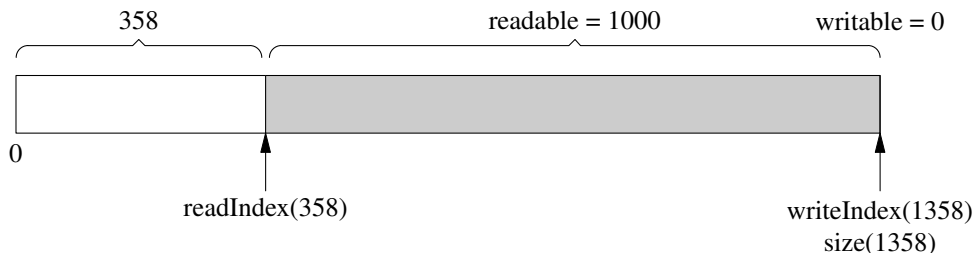


图 7-12

最后，读完剩下的 1000 字节，readIndex 和 writeIndex 返回 kCheapPrependable，如图 7-13 所示。

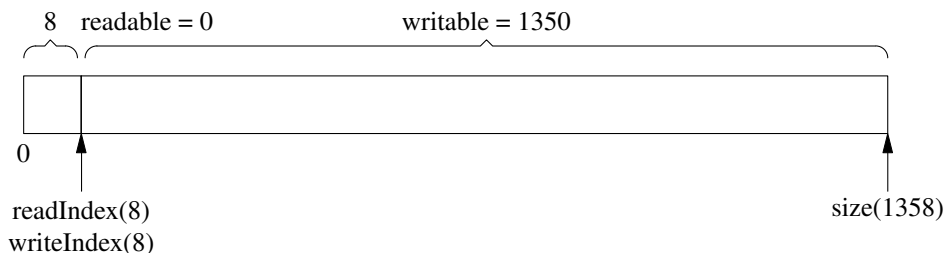


图 7-13

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

size() 与 capacity()

使用 vector 的另一个好处是它的 capacity() 机制减少了内存分配的次数。比方说程序反复写入 1 字节，muduo Buffer 不会每次都分配内存，vector 的 capacity() 以指数方式增长，让 push_back() 的平均复杂度是常数。比方说经过第一次增长，size() 刚好满足写入的需求，如图 7-14 所示。但这个时候 vector 的 capacity() 已经大于 size()，在接下来写入 capacity() - size() 字节的数据时，都不会重新分配内存，如图 7-15 所示。

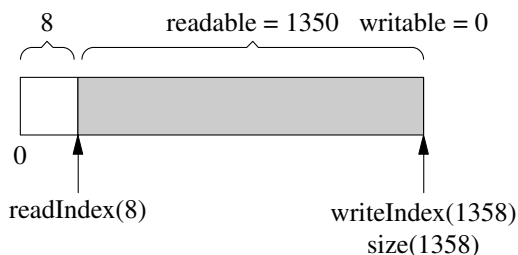


图 7-14

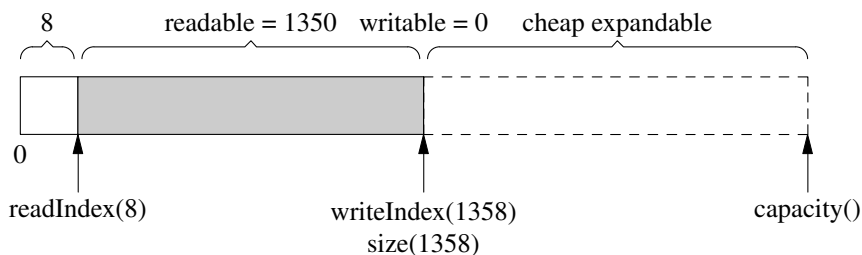


图 7-15

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

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

运行结果：

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

细心的读者可能会发现用 `capacity()` 也不是完美的，它有优化的余地。具体来说，`vector::resize()` 会初始化（`memset()`/`bzero()`）内存，而我们不需要它初始化，因为反正立刻就要填入数据。比如，在图 7-15 的基础上写入 200 字节，由于 `capacity()` 足够大，不会重新分配内存，这是好事；但是 `vector::resize()` 会先把那 200 字节 `memset()` 为 0（见图 7-16），然后 `muduo Buffer` 再填入数据（见图 7-17）。这么做稍微有点浪费，不过我不打算优化它，除非它确实造成了性能瓶

颈。(精通 STL 的读者可能会说用 `vec.insert(vec.end(), ...)` 以避免浪费, 但是 `writeIndex` 和 `size()` 不一定是对齐的, 会有别的麻烦。)

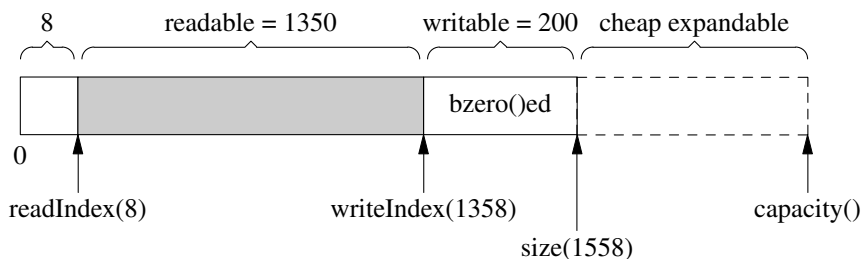


图 7-16

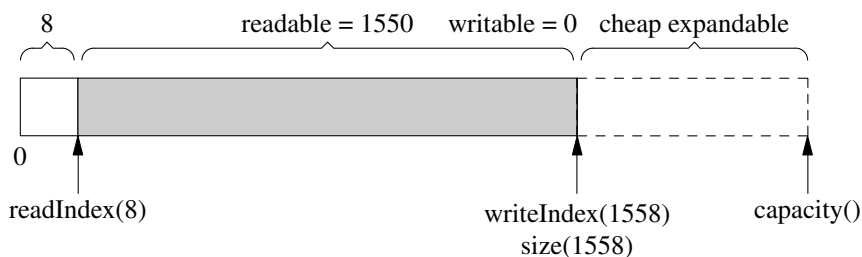


图 7-17

Google Protobuf 中有一个 `STLStringResizeUninitialized` 函数¹⁵, 干的就是这个事情。

内部腾挪

有时候, 经过若干次读写, `readIndex` 移到了比较靠后的位置, 留下了巨大的 `prependable` 空间, 如图 7-18 所示。

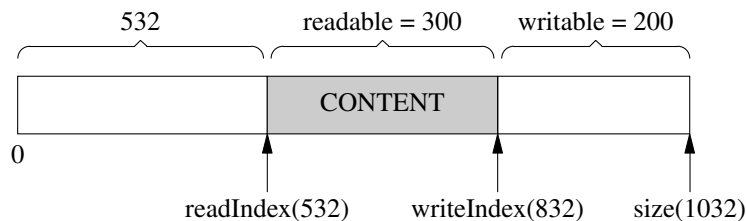


图 7-18

这时候, 如果我们想写入 300 字节, 而 `writable` 只有 200 字节, 怎么办? `muduo Buffer` 在这种情况下不会重新分配内存, 而是先把已有的数据移到前面去, 腾出 `writable` 空间, 如图 7-19 所示。

¹⁵ http://code.google.com/p/protobuf/source/browse/tags/2.4.0a/src/google/protobuf/stubs/stl_util-inLh#60

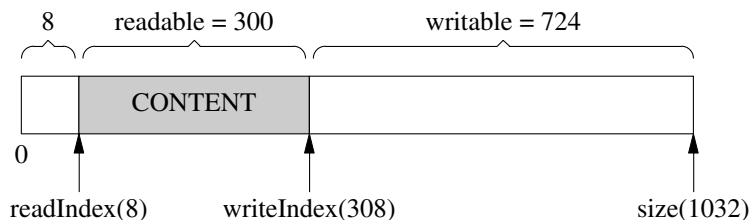


图 7-19

然后，就可以写入 300 字节了，如图 7-20 所示。

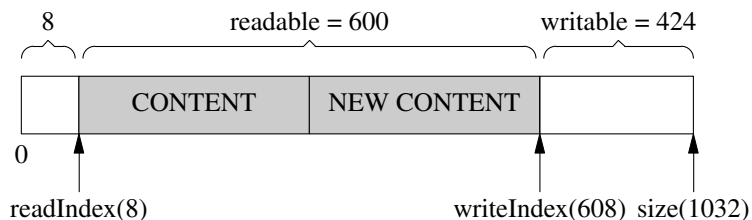


图 7-20

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

前方添加（prepend）

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

比方说，程序以固定的 4 个字节表示消息的长度（§7.3 “Boost.Asio 的聊天服务器”中的 LengthHeaderCode），我要序列化一个消息，但是不知道它有多长，那么我可以一直 append() 直到序列化完成（图 7-21，写入了 200 字节），然后再在序列化数据的前面添加消息的长度（图 7-22，把 200 这个数 prepend 到首部）。

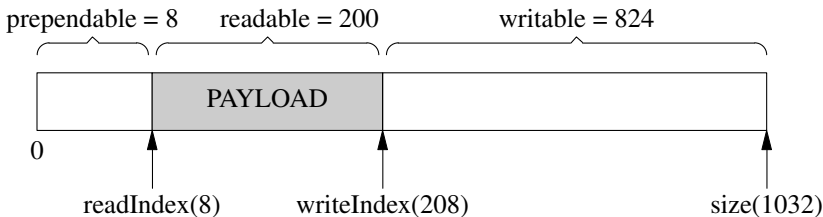


图 7-21

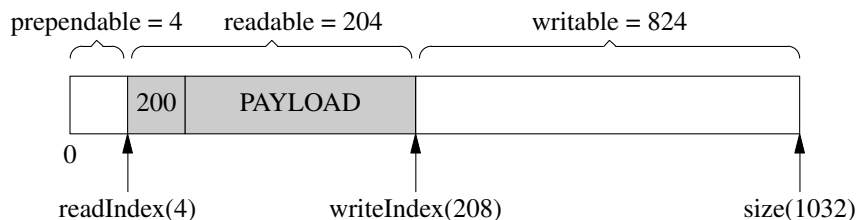


图 7-22

通过预留 `kCheapPrependable` 空间，可以简化客户代码，以空间换时间。

以上各种 use case 的单元测试见 `muduo/net/tests/Buffer_unittest.cc`。

7.4.6 其他设计方案

这里简单谈谈其他可能的应用层 buffer 设计方案。

不用 `vector<char>`

如果有 STL 洁癖，那么可以自己管理内存，以 4 个指针为 buffer 的成员，数据结构如图 7-23 所示。

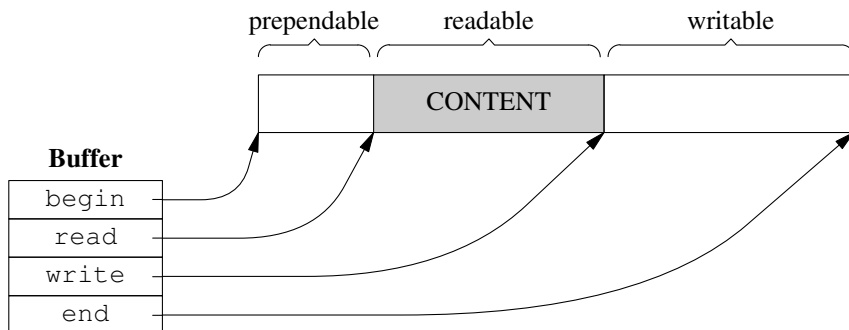


图 7-23

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

zero copy

如果对性能有极高的要求，受不了 `copy()` 与 `resize()`，那么可以考虑实现分段连续的 zero copy buffer 再配合 gather scatter IO，数据结构如图 7-24 所示，这是

libevent 2.0.x 的设计方案。TCPv2 介绍的 BSD TCP/IP 实现中的 mbuf 也是类似的方案，Linux 的 sk_buff 估计也差不多。细节有出入，但基本思路都是不要求数据在内存中连续，而是用链表把数据块链接到一起。

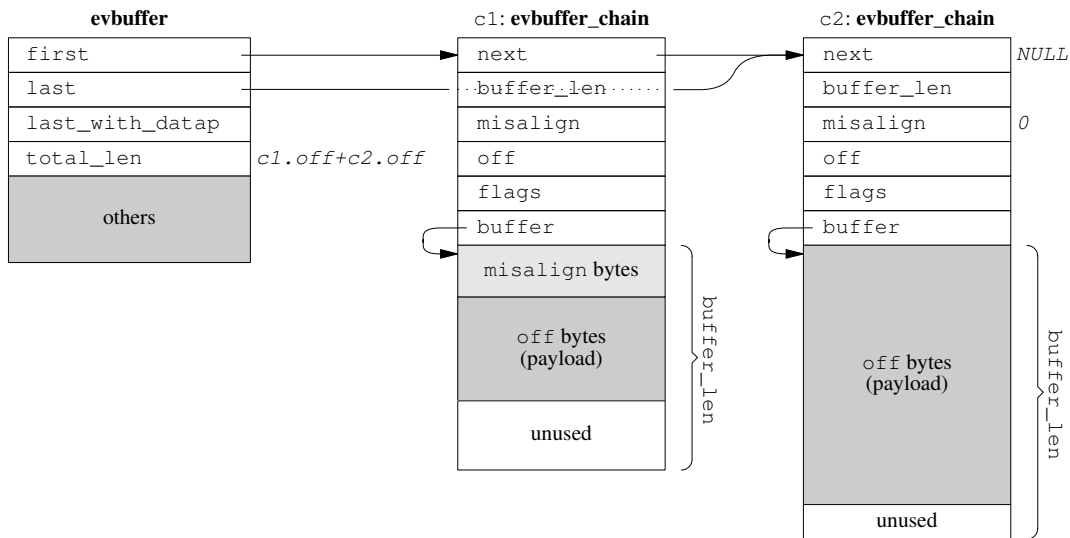


图 7-24

图 7-24 绘制的是由两个 evbuffer_chain 构成的 evbuffer，右边两个 evbuffer_chain 结构体中深灰色的部分是 payload，可见 evbuffer 的缓冲区不是连续的，而是分块的。

当然，高性能的代价是代码变得晦涩难读，buffer 不再是连续的，parse 消息会稍微麻烦一些。如果你的程序只处理 Protobuf Message，这不是问题，因为 Protobuf 有 ZeroCopyInputStream 接口，只要实现这个接口，parsing 的事情就交给 Protobuf Message 去操心了。

7.4.7 性能是不是问题

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

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

以 Sudoku 为例，假设求解一个 Sudoku 问题需要 0.2ms，服务器有 8 个核，那么理想情况下每秒最多能求解 40 000 个问题。每次 Sudoku 请求的数据大小低于 100

字节（一个 9×9 的数独只要 81 字节，加上 header 也可以控制在 100 字节以下），也就是说 $100 \times 40000 = 4\text{MB/s}$ 的吞吐量就足以让服务器的 CPU 饱和。在这种情况下，去优化 Buffer 的内存拷贝次数似乎没有意义。

再举一个例子，目前最常用的千兆以太网的裸吞吐量是 125MB/s，扣除以太网 header、IP header、TCP header 之后，应用层的吞吐率大约在 117MB/s 上下¹⁶。而现在服务器上最常用的 DDR2/DDR3 内存的带宽至少是 4GB/s，比千兆以太网高 40 倍以上。也就是说，对于几 kB 或几十 kB 大小的数据，在内存中复制几次根本不是问题，因为受千兆以太网延迟和带宽的限制，跟这个程序通信的其他机器上的程序不会觉察到性能差异。

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

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

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

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

¹⁶ 在不考虑 jumbo frame 的情况下，计算过程是：对于千兆以太网，每秒能传输 1000Mbit 数据，即 125 000 000B/s，每个以太网 frame 的固定开销有：preamble（8B）、MAC（12B）、type（2B）、payload（46B ~ 1500B）、CRC（4B）、gap（12B），因此最小的以太网帧是 84B，每秒可发送约 1 488 000 帧（换言之，对于一问一答的 RPC、其 qps 上限约是 700k/s），最大的以太网帧是 1538B，每秒可发送 81 274 帧。再来算 TCP 有效载荷：一个 TCP segment 包含 IP header（20B）和 TCP header（20B），还有 Timestamp option（12B），因此 TCP 的最大吞吐量是 $81\,274 \times (1500 - 52) = 117\text{MB/s}$ ，合 112MiB/s。实测见 §7.8.5。

7.5 一种自动反射消息类型的 Protobuf 网络传输方案

本节假定读者了解 Google Protocol Buffers 是什么，这不是一篇 Protobuf 入门教程。本节的示例代码位于 `examples/protobuf/codecs`。

本节要解决的问题是：通信双方在编译时就共享 proto 文件的情况下，接收方在收到 Protobuf 二进制数据流之后，如何自动创建具体类型的 Protobuf Message 对象，并用收到的数据填充该 Message 对象（即反序列化）。“自动”的意思是：当程序中新增一个 Protobuf Message 类型时，这部分代码不需要修改，不需要自己去注册消息类型。其实，Google Protobuf 本身具有很强的反射（reflection）功能，可以根据 type name 创建具体类型的 Message 对象，我们直接利用即可。¹⁷

7.5.1 网络编程中使用 Protobuf 的两个先决条件

Google Protocol Buffers（简称 Protobuf）是一款非常优秀的库，它定义了一种紧凑（compact，相对 XML 和 JSON 而言）的可扩展二进制消息格式，特别适合网络数据传输。

它为多种语言提供 binding，大大方便了分布式程序的开发，让系统不再局限于用某一种语言来编写。

在网络编程中使用 Protobuf 需要解决以下两个问题。

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

Protobuf 这么设计的原因见下一节。这里第一个问题很好解决，通常的做法是在每个消息前面加个固定长度的 length header，例如 §7.3 中实现的 LengthHeaderCode。第二个问题其实也很好解决，Protobuf 对此有内建的支持。但是奇怪的是，从网上简单搜索的情况看，我发现了很多“山寨”的做法。

“山寨”做法

以下均为在 Protobuf data 之前加上 header，header 中包含消息长度和类型信息。类型信息的“山寨”做法主要有两种：

¹⁷ Protobuf C++ 库的反射能力不止于此，它可以在运行时读入并解析任意 proto 文件，然后分析其对应的二进制数据。有兴趣的读者请参考王益的博客 <http://cxwangyi.blogspot.com/2010/06/google-protocol-buffers-protobuf.html>。

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

这两种做法都有问题。

第一种做法要求保持 `typeId` 的唯一性，它和 Protobuf message type 一一对应。如果 Protobuf message 的使用范围不广，比如接收方和发送方都是自己维护的程序，那么 `typeId` 的唯一性不难保证，用版本管理工具即可。如果 Protobuf message 的使用范围很大，比如全公司都在用，而且不同部门开发的分布式程序可能相互通信，那么就需要一个公司内部的全局机构来分配 `typeId`，每次增加新 message type 都要去注册一下，比较麻烦。

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

其实，不需要自己重新发明轮子，Protobuf 本身已经自带了解决方案。

7.5.2 根据 type name 反射自动创建 Message 对象

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

以下是笔者绘制的 Protobuf class diagram（见图 7-25）。我估计大家通常关心和使用的是这个类图的左半部分：`MessageLite`、`Message`、`Generated Message Types`（`Person`、`AddressBook`）等，而较少注意到图 7-25 的右半部分：`Descriptor`、`DescriptorPool`、`MessageFactory`。

在图 7-25 中，起关键作用的是 `Descriptor class`，每个具体 Message type 对应一个 `Descriptor` 对象。尽管我们没有直接调用它的函数，但是 `Descriptor` 在“根据 type name 创建具体类型的 Message 对象”中扮演了重要的角色，起了桥梁作用。图 7-25 中的 $\leftarrow$ 箭头描述了根据 type name 创建具体 Message 对象的过程，后文会详细介绍。

原理简述

Protobuf Message class 采用了 Prototype pattern¹⁸, Message class 定义了 New() 虚函数, 用以返回本对象的一份新实体, 类型与本对象的真实类型相同。也就是说, 拿到 Message* 指针, 不用知道它的具体类型, 就能创建和其类型一样的具体 Message type 的对象。

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

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

我们的英雄是 DescriptorPool, 它可以根据 type name 查到 Descriptor*, 只要找到合适的 DescriptorPool, 再调用 DescriptorPool::FindMessageTypeByName(const string& type_name) 即可。看到图 7-25 是不是眼前一亮?

在最终解决问题之前, 先简单测试一下, 看看我上面说得对不对。

验证思路

本文用于举例的 proto 文件:

```
package muduo;

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

    repeated string question = 3;
}

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

    repeated string solution = 4;
}
```

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


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

examples/protobuf/codec/query.proto

其中的 `Query.questioner` 和 `Answer.answerer` 是 §9.4 提到的“分布式系统中的进程标识”。

以下代码¹⁹ 验证 `ConcreteMessage::default_instance()`、`ConcreteMessage::descriptor()`、`MessageFactory::GetPrototype()`、`DescriptorPool::FindMessageTypeByName()` 之间的不变式 (invariant)，注意其中的 `assert`：

```
typedef muduo::Query T;

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

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

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

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

程序运行结果如下：

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

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

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

¹⁹ recipes/protobuf/descriptor_test.cc

根据 type name 自动创建 Message 的关键代码

好了，万事俱备，开始行动：

1. 用 `DescriptorPool::generated_pool()` 找到一个 `DescriptorPool` 对象，它包含了程序编译的时候所链接的全部 Protobuf Message types。
2. 根据 type name 用 `DescriptorPool::FindMessageTypeByName()` 查找 `Descriptor`。
3. 再用 `MessageFactory::generated_factory()` 找到 `MessageFactory` 对象，它能创建程序编译的时候所链接的全部 Protobuf Message types。
4. 然后，用 `MessageFactory::GetPrototype()` 找到具体 Message type 的 default instance。
5. 最后，用 `prototype->New()` 创建对象。

示例代码如下。

```

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

```

examples/protobuf/codec/codec.cc

examples/protobuf/codec/codec.cc

调用方式：

```

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

```

确实能从消息名称创建消息对象，古之人不余欺也:-)

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

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

线程安全性

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

7.5.3 Protobuf 传输格式

笔者设计了一个简单的格式，包含 Protobuf data 和其对应的长度与类型信息，消息的末尾还有一个 check sum。格式如图 7-26 所示，图中方块的宽度是 32-bit。

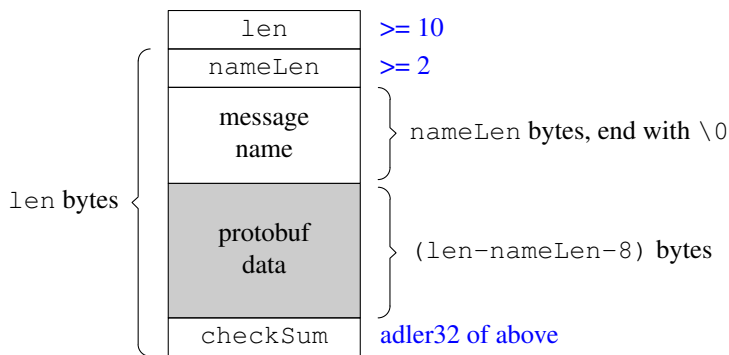


图 7-26

用 C struct 伪代码描述：

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

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

例子

用这个格式打包一个 muduo.Query 对象的结果如图 7-27 所示。

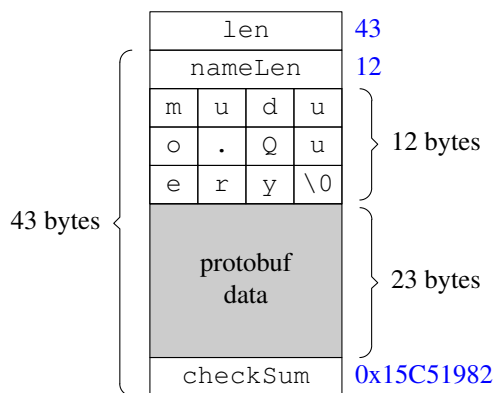


图 7-27

设计决策

以下是我在设计这个传输格式时的考虑：

- **signed int**。消息中的长度字段只使用了 signed 32-bit int，而没有使用 unsigned int，这是为了跨语言移植性，因为 Java 语言没有 unsigned 类型。另外，Protobuf 一般用于打包小于 1MB 的数据，unsigned int 也没用。
- **check sum**。虽然 TCP 是可靠传输协议，虽然 Ethernet 有 CRC-32 校验，但是网络传输必须要考虑数据损坏的情况，对于关键的网络应用，check sum 是必不可少的。见 §A.1.13 “TCP 的可靠性有多高”。对于 Protobuf 这种紧凑的二进制格式而言，肉眼看不出数据有没有问题，需要用 check sum。
- **adler32 算法**。我没有选用常见的 CRC-32，而是选用了 adler32，因为它的计算量小、速度比较快，强度和 CRC-32 差不多。另外，zlib 和 java.util.zip 都直接支持这个算法，不用我们自己实现。
- **type name 以 '\0' 结束**。这是为了方便 troubleshooting，比如通过 tcpdump 抓下来的包可以用肉眼很容易看出 type name，而不用根据 nameLen 去一个个数字节。同时，为了方便接收方处理，加入了 nameLen，节省了 strlen()，这是以空间换时间的做法。
- **没有版本号**。Protobuf Message 的一个突出优点是用 optional fields 来避免协议的版本号（凡是在 Protobuf Message 里放版本号的人都没有理解 Protobuf 的设计，甚至可能没有仔细阅读 Protobuf 的文档^{20 21 22}），让通信双方的程序能

²⁰ <http://code.google.com/apis/protocolbuffers/docs/overview.html> “A bit of history”

²¹ <http://code.google.com/apis/protocolbuffers/docs/proto.html#updating>

²² <http://code.google.com/apis/protocolbuffers/docs/cpptutorial.html> “Extending a Protocol Buffer”

各自升级，便于系统演化。如果我设计的这个传输格式又把版本号加进去，那就画蛇添足了。

Protobuf 可谓是网络协议格式的典范，值得我单独花一节篇幅讲述其思想，见 §9.6.1 “可扩展的消息格式”。

7.6 在 muduo 中实现 Protobuf 编解码器与消息分发器

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

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

为什么 Protobuf 的默认序列化格式没有包含消息的长度与类型

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

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

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

对于以上最后一点，比方说 `sudoku.proto` 的定义是：

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

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

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

只有在使用 TCP 长连接，且在一个连接上传递不止一种消息的情况下（比方同时发 `Heartbeat` 和 `Request/Response`），才需要我前文提到的那种打包方案²³。这时候我们需要一个分发器 `dispatcher`，把不同类型的消息分给各个消息处理函数，这正是本节的主题之一。

以下均只考虑 TCP 长连接这一应用场景。先谈谈编解码器。

7.6.1 什么是编解码器（codec）

编解码器（`codec`）²⁴ 是 `encoder` 和 `decoder` 的缩写，这是一个软硬件领域都在使用的术语，这里我借指“把网络数据和业务消息之间互相转换”的代码。

在最简单的网络编程中，没有消息（`message`），只有字节流数据，这时候是用不到 `codec` 的。比如我们前面讲过的 `echo server`，它只需要把收到的数据原封不动地发送回去，而不必关心消息的边界（也没有“消息”的概念），收多少就发多少，这种情况下它干脆直接使用 `muduo::net::Buffer`，取到数据再交给 `TcpConnection` 发送回去，如图 7-28 所示。

²³ 为什么要在一个连接上同时发 `Heartbeat` 和业务消息？见 §9.3。

²⁴ <http://en.wikipedia.org/wiki/Codec>

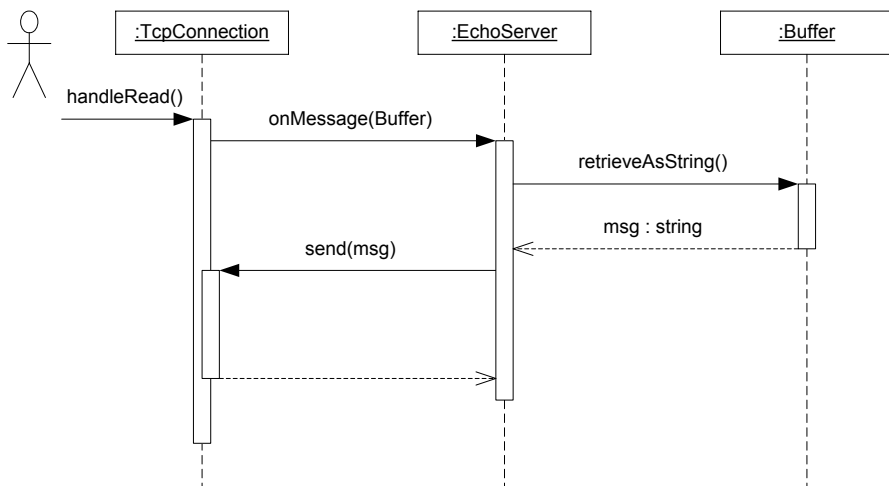


图 7-28

non-trivial 的网络服务程序通常会以消息为单位来通信，每条消息有明确的长度与界限。程序每次收到一个完整的消息的时候才开始处理，发送的时候也是把一个完整的消息交给网络库。比如我们前面讲过的 asio chat 服务，它的一条聊天记录就是一条消息。为此我们设计了一个简单的消息格式，即在聊天记录前面加上 4 字节的 length header，LengthHeaderCode 代码及解说见 §7.3。

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

codec 是一层间接性，它位于 TcpConnection 和 ChatServer 之间，拦截处理收到的数据（Buffer），在收到完整的消息之后，解出消息对象（std::string），再调用 ChatServer 对应的处理函数。注意 ChatServer::onStringMessage() 的参数是 std::string，不再是 muduo::net::Buffer，也就是说 LengthHeaderCode 把 Buffer 解码成了 string。另外，在发送消息的时候，ChatServer 通过 LengthHeaderCode::send() 来发送 string，LengthHeaderCode 负责把它编码成 Buffer。这正是

“编解码器”名字的由来。消息流程如图 7-29 所示。

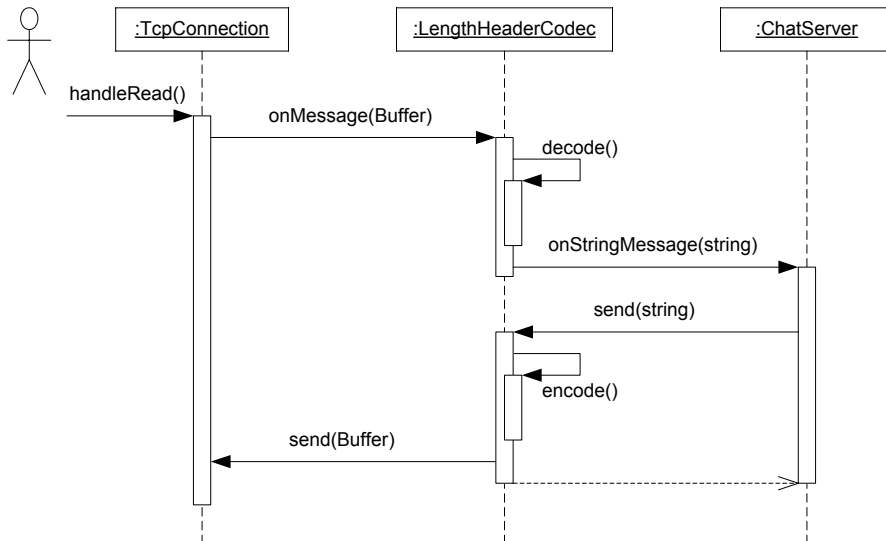


图 7-29

Protobuf codec 与此非常类似，只不过消息类型从 `std::string` 变成了 `protobuf::Message`。对于只接收处理 Query 消息的 `QueryServer` 来说，用 `ProtobufCodec` 非常方便，收到 `protobuf::Message` 之后向下转型成 Query 来用就行（见图 7-30）。

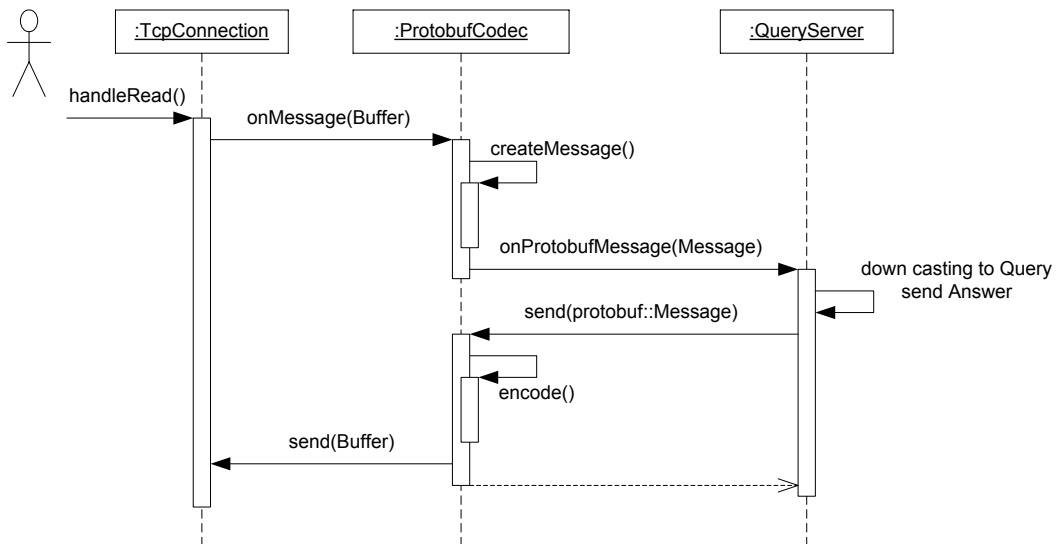


图 7-30

如果要接收处理不止一种消息，ProtobufCodec 恐怕还不能单独完成工作，请继续阅读下文。

7.6.2 实现 ProtobufCodec

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

解码算法有几个要点：

- protobuf::Message 是 new 出来的对象，它的生命期如何管理？muduo 采用 shared_ptr<Message> 来自动管理对象生命期，与整体风格保持一致。
- 出错如何处理？比方说长度超出范围、check sum 不正确、message type name 不能识别、message parse 出错等等。ProtobufCodec 定义了 ErrorCallback，用户代码可以注册这个回调。如果不注册，默认的处理是断开连接，让客户重连重试。codec 的单元测试里模拟了各种出错情况。
- 如何处理一次收到半条消息、一条消息、一条半消息、两条消息等情况？这是每个 non-blocking 网络程序中的 codec 都要面对的问题。在 p. 196 的示例代码中我们已经解决了这个问题。

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

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

7.6.3 消息分发器（dispatcher）有什么用

前面提到，在使用 TCP 长连接，且在一个连接上传递不止一种 Protobuf 消息的情况下，客户代码需要对收到的消息按类型做分发。比方说，收到 Logon 消息就交给

QueryServer::onLogon() 去处理, 收到 Query 消息就交给 QueryServer::onQuery() 去处理。这个消息分派机制可以做得稍微有点通用性, 让所有 muduo+Protobuf 程序受益, 而且不增加复杂性。

换句话说, 又是一层间接性, ProtobufCodec 拦截了 TcpConnection 的数据, 把它转换为 Message, ProtobufDispatcher 拦截了 ProtobufCodec 的 callback, 按消息具体类型把它分派给多个 callbacks, 如图 7-31 所示。

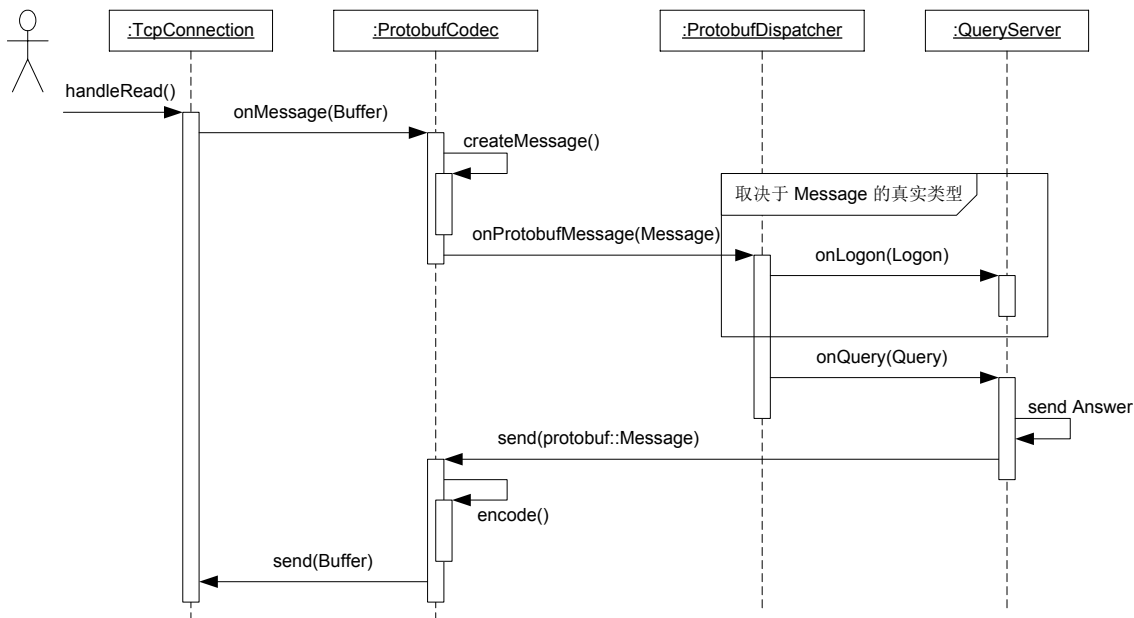


图 7-31

7.6.4 ProtobufCodec 与 ProtobufDispatcher 的综合运用

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

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

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

7.6.5 ProtobufDispatcher 的两种实现

要完成消息分发，其实就是对消息做 type-switch，这似乎是一个 bad smell，但是 Protobuf Message 的 Descriptor 没有留下定制点（比如暴露一个 boost::any 成员），我们只好硬来了。

先定义 ProtobufMessageCallback 回调：

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

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

ProtobufDispatcherLite 的结构非常简单（见图 7-32），它有一个 map<Descriptor*, ProtobufMessageCallback> 成员，客户代码可以以 Descriptor* 为 key 注册回调（回想：每个具体消息类型都有一个全局的 Descriptor 对象，其地址是不变的，可以用来当 key）。在收到 Protobuf Message 之后，在 map 中找到对应的 ProtobufMessageCallback，然后调用之。如果找不到，就调用 defaultCallback。

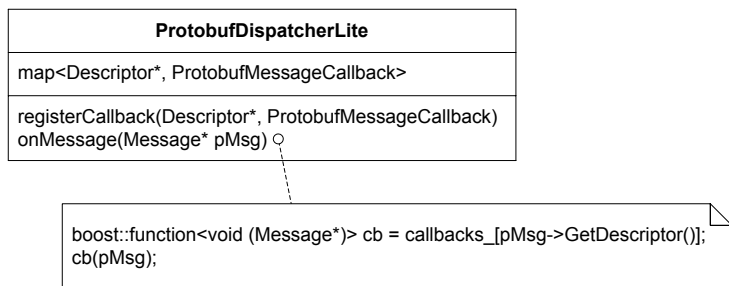


图 7-32

不过，它的设计也有小小的缺陷，那就是 ProtobufMessageCallback 限制了客户代码只能接受基类 Message，客户代码需要自己做向下转型（down cast），如图 7-33 所示。

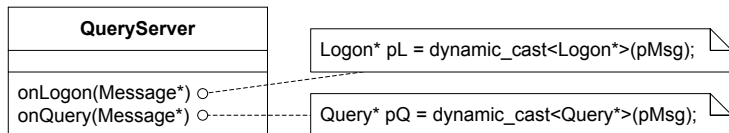


图 7-33

如果我希望 QueryServer 这么设计：不想每个消息处理函数自己做 down cast，而是交给 dispatcher 去处理，客户代码拿到的就已经是想要的类型。接口如图 7-34 所示。

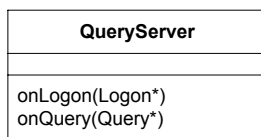


图 7-34

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

有一个办法，把多态与模板结合，利用 templated derived class 来提供类型上的灵活性。设计如图 7-35 所示²⁵。

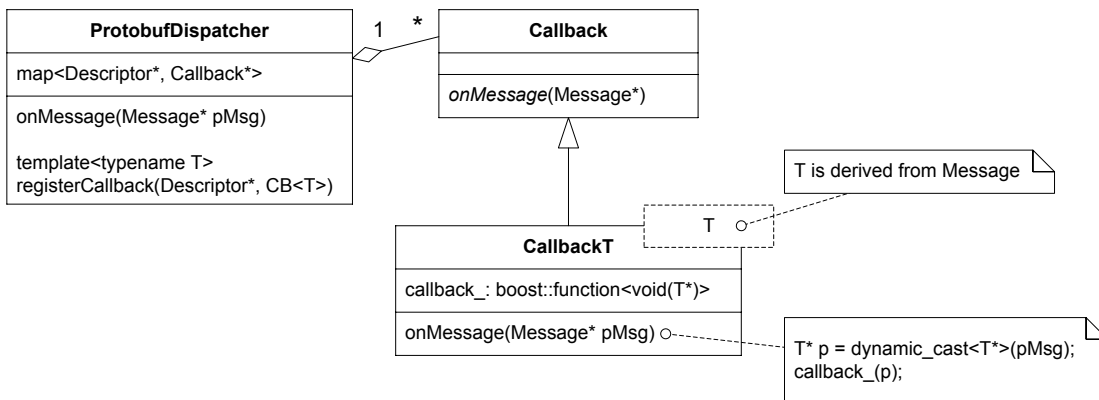


图 7-35

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

比方说，我们有两个具体消息类型 Query 和 Answer（见图 7-36）。

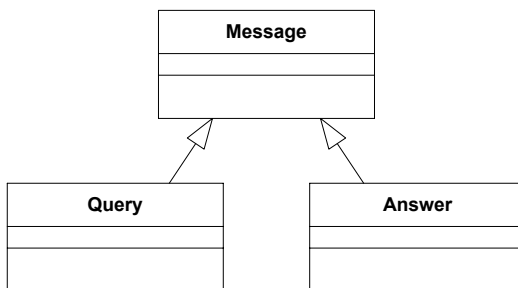


图 7-36

²⁵ 图中画的是 dynamic_cast，代码实际上自定义了 down_cast 转换操作，在 Debug 编译时会检查动态类型，而在 NDEBUG 编译时会退化为 static_cast，没有 RTTI 开销。

然后我们这样注册回调：

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

这样会具现化（instantiation）出两个 CallbackT 实体，如图 7-37 所示。

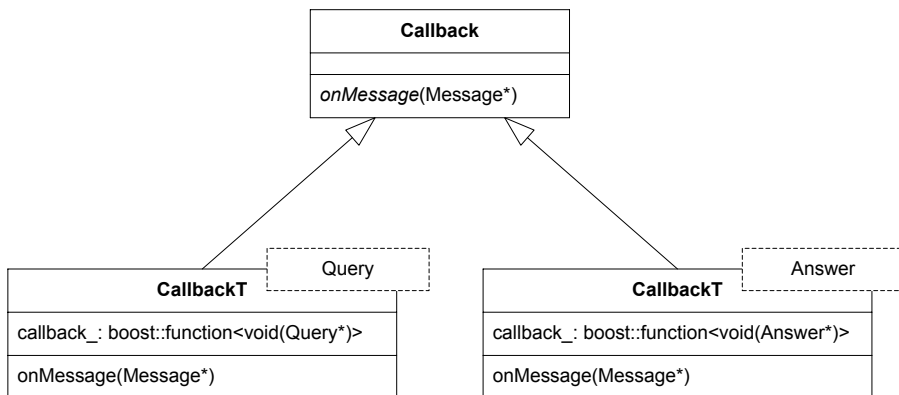


图 7-37

以上设计参考了 shared_ptr 的 deleter，Scott Meyers 也谈到过²⁶。

7.6.6 ProtobufCodec 和 ProtobufDispatcher 有何意义

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

§9.7 “分布式程序的自动化回归测试”会介绍利用 Protobuf 的跨语言特性，采用 Java 为 C++ 服务程序编写 test harness。

这种编码方案的 Java Netty 示例代码见 <http://github.com/chenshuo/muduo-protorpc> 中的 com.chenshuo.muduo.codec package。

²⁶ http://www.artima.com/cppsource/top_cpp_aha_moments.html

7.7 限制服务器的最大并发连接数

本节以大家都熟悉的 EchoServer 为例，介绍如何限制 TCP 服务器的并发连接数。代码见 examples/maxconnection/。

本节中的“并发连接数”是指一个服务端程序能同时支持的客户端连接数，连接由客户端主动发起，服务端被动接受（accept(2)）连接。（如果要限制应用程序主动发起的连接，则问题要简单得多，毕竟主动权和决定权都在程序本身。）

7.7.1 为什么要限制并发连接数

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

我 2010 年 10 月在《分布式系统的工程化开发方法》演讲²⁷中曾谈到 libev 的作者 Marc Lehmann 建议的一种应对“accept() 时 file descriptor 耗尽”的办法²⁸。

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

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

²⁷ <http://blog.csdn.net/Solstice/article/details/5950190> http://www.youku.com/playlist_show/id_5238686.html

²⁸ http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#The_special_problem_of_accepting_wh

假如 L14 的 `accept(2)` 返回 `EMFILE` 该如何应对？这意味着本进程的文件描述符已经达到上限，无法为新连接创建 `socket` 文件描述符。但是，既然没有 `socket` 文件描述符来表示这个连接，我们就无法 `close(2)` 它。程序继续运行，回到 L11 再一次调用 `epoll_wait`。这时候 `epoll_wait` 会立刻返回，因为新连接还等待处理，`listening fd` 还是可读的。这样程序立刻就陷入了 `busy loop`，CPU 占用率接近 100%。这既影响同一 `event loop` 上的连接，也影响同一机器上的其他服务。

该怎么办呢？Marc Lehmann 提到了几种做法：

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

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

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

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

7.7.2 在 `muduo` 中限制并发连接数

在 `muduo` 中限制并发连接数的做法简单得出奇。以在 §6.4.2 的 `EchoServer` 为例，只需要为它增加一个 `int` 成员，表示当前的活动连接数。（如果是多线程程序，应该用 `muduo::AtomicInt32`。）

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

         void start();

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

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

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

```
----- examples/maxconnection/echo.cc
void EchoServer::onConnection(const TcpConnectionPtr& conn)
{
    LOG_INFO << "EchoServer - " << conn->peerAddress().toIpPort() << " -> "
              << conn->localAddress().toIpPort() << " is "
              << (conn->connected() ? "UP" : "DOWN");

+
+   if (conn->connected())
+   {
+       ++numConnected_;
+       if (numConnected_ > kMaxConnections_) // 如果超过最大允许数，则踢掉连接
+       {
+           conn->shutdown();
+       }
+   }
+   else
+   {
+       --numConnected_;
+   }
+   LOG_INFO << "numConnected = " << numConnected_;
}
```

----- examples/maxconnection/echo.cc

这种做法可以积极地防止耗尽 file descriptor。

另外，如果有业务逻辑的服务，则可以在 `shutdown()` 之前发送一个简单的响应，表明本服务程序的负载能力已经饱和，提示客户端尝试下一个可用的 server（当

然，下一个可用的 server 地址不一定要在这个响应里给出，客户端可以自己 go name service 查询），这样方便客户端快速 failover。

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

7.8 定时器

从本节开始的三节内容都与非阻塞网络编程中的定时任务有关。

7.8.1 程序中的时间

程序中对时间的处理是个大问题，在这一节中我先简要谈谈与编程直接相关的内容，把更深入的内容留给日后日期与时间专题文章²⁹，本书不再细述。

在一般的服务端程序设计中，与时间有关的常见任务有：

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

其中第 2 项看起来比较复杂，但其实最简单。日期计算用 Julian Day Number³⁰，时区转换用 tz database³¹；唯一麻烦一点的是夏令时，但也可以用 tz database 解决。这些操作都是纯函数，很容易用一套单元测试来验证代码的正确性。需要特别注意的是，用 tzset/localtime_r 来做时区转换在多线程环境下可能会有问题；对此，我的解决办法是写一个 TimeZone class，以避免影响全局，日后在日期与时间专题文章中会讲到，本书不再细述。下文不考虑时区，均为 UTC 时间。

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

²⁹ <http://blog.csdn.net/solstice/article/category/790732>

³⁰ <http://blog.csdn.net/solstice/article/details/5814486>

³¹ <http://cs.ucla.edu/~eggert/tz/tz-link.htm> <http://www.iana.org/time-zones>

7.8.2 Linux 时间函数

Linux 的计时函数，用于获得当前时间：

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

还有 `gmtime` / `localtime` / `timegm` / `mktime` / `strftime` / `struct tm` 等与当前时间无关的时间格式转换函数。

定时函数，用于让程序等待一段时间或安排计划任务：

- `sleep(3)`
- `alarm(2)`
- `usleep(3)`
- `nanosleep(2)`
- `clock_nanosleep(2)`
- `getitimer(2)` / `setitimer(2)`
- `timer_create(2)` / `timer_settime(2)` / `timer_gettime(2)` / `timer_delete(2)`
- `timerfd_create(2)` / `timerfd_gettime(2)` / `timerfd_settime(2)`

我的取舍如下：

- （计时）只使用 `gettimeofday(2)` 来获取当前时间。
- （定时）只使用 `timerfd_*` 系列函数来处理定时任务。

`gettimeofday(2)` 入选原因（这也是 `muduo::Timestamp` class 的主要设计考虑）：

1. `time(2)` 的精度太低，`ftime(3)` 已被废弃；`clock_gettime(2)` 精度最高，但是其系统调用的开销比 `gettimeofday(2)` 大。
2. 在 x86-64 平台上，`gettimeofday(2)` 不是系统调用，而是在用户态实现的，没有上下文切换和陷入内核的开销³²。
3. `gettimeofday(2)` 的分辨率（resolution）是 1 微秒，现在的实现确实能达到这个计时精度，足以满足日常计时的需要。`muduo::Timestamp` 用一个 `int64_t` 来表示从 Unix Epoch 到现在的微秒数，其范围可达上下 30 万年。

³² <http://lwn.net/Articles/446528/>

timerfd_* 入选的原因:

1. sleep(3) / alarm(2) / usleep(3) 在实现时有可能用了 SIGALRM 信号, 在多线程程序中处理信号是个相当麻烦的事情, 应当尽量避免, 见 §4.10。再说, 如果主程序和程序库都使用 SIGALRM, 就糟糕了。(为什么?)
2. nanosleep(2) 和 clock_nanosleep(2) 是线程安全的, 但是在非阻塞网络编程中, 绝对不能用让线程挂起的方式来等待一段时间, 这样一来程序会失去响应。正确的做法是注册一个时间回调函数。
3. getitimer(2) 和 timer_create(2) 也是用信号来 deliver 超时, 在多线程程序中也会有麻烦。timer_create(2) 可以指定信号的接收方是进程还是线程, 算是一个进步, 不过信号处理函数 (signal handler) 能做的事情实在很受限。
4. timerfd_create(2) 把时间变成了一个文件描述符, 该“文件”在定时器超时的那一刻变得可读, 这样就能很方便地融入 select(2)/poll(2) 框架中, 用统一的方式来处理 IO 事件和超时事件, 这也正是 Reactor 模式的长处。我在以前发表的《Linux 新增系统调用的启示》³³ 中也谈到了这个想法, 现在我把这个想法在 muduo 网络库中实现了。
5. 传统的 Reactor 利用 select(2)/poll(2)/epoll(4) 的 timeout 来实现定时功能, 但 poll(2) 和 epoll_wait(2) 的定时精度只有毫秒, 远低于 timerfd_settime(2) 的定时精度。

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

关于时间的精度 (accuracy) 问题我留到日期与时间专题文章中讨论, 本书不再细述, 它与分辨率 (resolution) 不完全是一回事儿。时间跳变和闰秒的影响与应对也不在此处展开讨论了。

7.8.3 muduo 的定时器接口

muduo EventLoop 有三个定时器函数:

³³ <http://blog.csdn.net/Solstice/article/details/5327881>

```

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

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

    // timers

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

    /// Runs callback after @c delay seconds.
    TimerId runAfter(double delay, const TimerCallback& cb);

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

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

    // ...
};

```

函数名称很好地反映了其用途：

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

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

7.8.4 Boost.Asio Timer 示例

Boost.Asio 教程³⁴里以 Timer 和 Daytime 为例介绍 Asio 的基本使用，daytime 已经在 §7.1 中介绍过，这里着重谈谈 Timer。Asio 有 5 个 Timer 示例，muduo 把其中四个重新实现了一遍，并扩充了第 5 个示例。

³⁴ http://www.boost.org/doc/libs/release/doc/html/boost_asio/tutorial.html

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

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

```

examples/asio/tutorial/timer4/timer.cc
7 class Printer : boost::noncopyable
8 {
9     public:
10     Printer(muduo::net::EventLoop* loop)
11         : loop_(loop),
12           count_(0)
13     {
14         loop_->runAfter(1, boost::bind(&Printer::print, this));
15     }
16
17     ~Printer()
18     {
19         std::cout << "Final count is " << count_ << "\n";
20     }
21
22     void print()
23     {
24         if (count_ < 5)
25         {
26             std::cout << count_ << "\n";
27             ++count_;
28
29             loop_->runAfter(1, boost::bind(&Printer::print, this));
30         }
31         else
32         {
33             loop_->quit();
34         }
35     }
36
37     private:
38     muduo::net::EventLoop* loop_;
39     int count_;
40 };
41

```

```

42 int main()
43 {
44     muduo::net::EventLoop loop;
45     Printer printer(&loop);
46     loop.loop();
47 }

```

examples/asio/tutorial/timer4/timer.cc

最后我再强调一遍，在非阻塞服务端编程中，绝对不能用 `sleep()` 或类似的办法来让程序原地停留等待，这会让程序失去响应，因为主事件循环被挂起了，无法处理 IO 事件。这就像在 Windows 编程中绝对不能在消息循环里执行耗时的代码是一个道理，这会让程序界面失去响应。Reactor 模式的网络编程确实有些类似传统的消息驱动的 Windows 编程。对于“定时”任务，把它变成一个特定的消息，到时候触发相应的消息处理函数就行了。

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

7.8.5 Java Netty 示例

Netty 是一个非常好的 Java NIO 网络库，它附带的示例程序有 `echo` 和 `discard` 两个简单网络协议。与 §7.1 不同，Netty 版的 `echo` 和 `discard` 服务端有流量统计功能，这需要用到固定间隔的定时器（`EventLoop::runEvery`）。

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

这里列出 `discard server` 的完整代码。代码整体结构上与 §6.4.2 的 `EchoServer` 差别不大，这算是简单网络服务器的典型模式了。

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

```

19 int numThreads = 0;
20
21 class DiscardServer
22 {
23 public:
24     DiscardServer(EventLoop* loop, const InetAddress& listenAddr)
25         : loop_(loop),
26           server_(loop, listenAddr, "DiscardServer"),
27           oldCounter_(0),
28           startTime_(Timestamp::now())

```

muduo/examples/netty/discard/server.cc

```

29 {
30     server_.setConnectionCallback(
31         boost::bind(&DiscardServer::onConnection, this, _1));
32     server_.setMessageCallback(
33         boost::bind(&DiscardServer::onMessage, this, _1, _2, _3));
34     server_.setThreadNum(numThreads);
35     loop->runEvery(3.0, boost::bind(&DiscardServer::printThroughput, this));
36 }

```

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

消息回调只比 p. 178 的代码多两行，用于统计收到的数据长度和消息次数。

```

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

```

在每一个统计周期，打印数据吞吐量。

```

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

```

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

```

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

```

`main()` 函数，有一个可选的命令行参数，用于指定线程数目。

```
85 int main(int argc, char* argv[])
86 {
87     LOG_INFO << "pid = " << getpid() << ", tid = " << CurrentThread::tid();
88     if (argc > 1)
89     {
90         numThreads = atoi(argv[1]);
91     }
92     EventLoop loop;
93     InetAddress listenAddr(2009);
94     DiscardServer server(&loop, listenAddr);
95
96     server.start();
97
98     loop.loop();
99 }
```

muduo/examples/netty/discard/server.cc

运行方法，在同一台机器的两个命令行窗口分别运行：

```
# 窗口 1
$ bin/netty_discard_server

# 窗口 2
$ bin/netty_discard_client 127.0.0.1 256
```

第一个窗口显示吞吐量：

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

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

练习 1：把二者的关系绘制成函数曲线，看看有什么规律，想想为什么。

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

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

`discard client/server` 测试的是单向吞吐量，`echo client/server` 测试的是双向吞吐量。这两个服务端都支持多个并发连接，两个客户端都是单连接的。前文 §6.5 实现了一个 pingpong 协议，客户端和服务端都是多连接，用来测试 muduo 在多线程大量连接情况下的性能表现。

7.9 测量两台机器的网络延迟和时间差

本节介绍一个简单的网络程序 `roundtrip`，用于测量两台机器之间的网络延迟，即“往返时间（round trip time, RTT）”。其主要考察定长 TCP 消息的分包与 `TCP_NODELAY` 的作用。本节的代码见 `examples/roundtrip/roundtrip.cc`。

测量 round trip time 的办法很简单：

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

NTP 协议的工作原理与之类似³⁵，不过，除了测量 round trip time，NTP 还需要知道两台机器之间的时间差（clock offset），这样才能校准时间。

图 7-38 是 NTP 协议收发消息的协议， $\text{round trip time} = (T_4 - T_1) - (T_3 - T_2)$ ， $\text{clock offset} = \frac{(T_4 + T_1) - (T_2 + T_3)}{2}$ 。NTP 的要求是往返路径上的单程延迟要尽量相等，这样才能减少系统误差。偶然误差由单程延迟的不确定性决定。

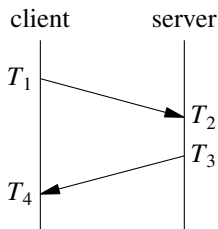


图 7-38

在我设计的 `roundtrip` 示例程序中，协议有所简化，如图 7-39 所示。

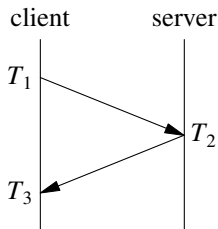


图 7-39

³⁵ NTP 的原理是持续检测本机与时间服务器的时差，调整本机的时钟频率和时间 offset，让修正后的本机时间与服务器时间尽量接近。NTP 的核心是一个数字锁相环，这里的“相位”就是时间，频率是时钟快慢。NTP 对时除了要拨表盘指针，还要调钟摆长短以控制钟的快慢。

计算公式如下。

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

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

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

我设计的消息格式是 16 字节定长消息，如图 7-40 所示。

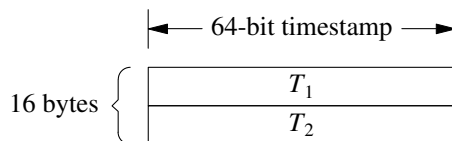


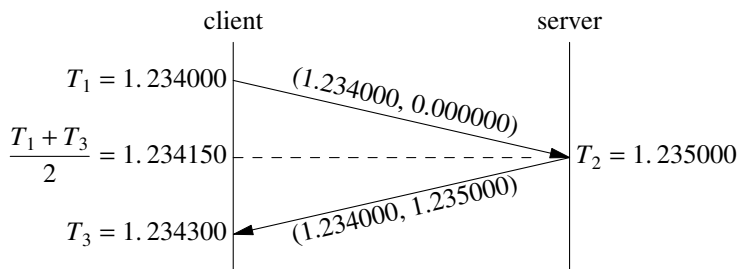
图 7-40

T_1 和 T_2 都是 `muduo::Timestamp`，成员是一个 `int64_t`，表示从 Unix Epoch 到现在的微秒数。为了让消息的单程往返时间接近，server 和 client 发送的消息都是 16 bytes，这样做到对称。由于是定长消息，可以不必使用 codec，在 message callback 中直接用

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

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

client 程序以 200ms 为间隔发送消息，在收到消息之后打印 round trip time 和 clock offset。一次运作实例如图 7-41 所示。



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

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

图 7-41

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

比方说以图 7-41 为例，server 在它本地 1.235000s 时刻发送了一条消息，client 在它本地 1.234300s 收到这条消息，若直接计算的话延迟是 -700 μ s。这个结果肯定是错的，因为 server 和 client 不在一个时钟域（clock domain，这是数字电路中的概念），它们的时间直接相减无意义。如果我们已经测量得到 server 比 client 快 850 μ s，那么用这个数据做一次校正： $-700 + 850 = 150\mu$ s，这个结果就比较符合实际了。当然，在实际应用中，clock offset 要经过一个低通滤波才能使用，不然偶然性太大。

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

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

7.10 用 timing wheel 踢掉空闲连接

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

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

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

- 每个连接保存“最后收到数据的时间 lastReceiveTime”，然后用一个定时器，每秒遍历一遍所有连接，断开那些 $(\text{now} - \text{connection.lastReceiveTime}) > 8\text{s}$ 的 connection。这种做法全局只有一个 repeated timer，不过每次 timeout 都要检查全部连接，如果连接数目比较大（几千上万），这一步可能会比较费时。

- 每个连接设置一个 one-shot timer，超时定为 8s，在超时的时候就断开本连接。当然，每次收到数据要去更新 timer。这种做法需要很多个 one-shot timer，会频繁地更新 timers。如果连接数目比较大，可能对 EventLoop 的 TimerQueue 造成压力。

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

连接超时不需要精确定时，只要大致 8 秒超时断开就行，多一秒、少一秒关系不大。处理连接超时可用一个简单的数据结构：8 个桶组成的循环队列。第 1 个桶放 1 秒之后将要超时的连接，第 2 个桶放 2 秒之后将要超时的连接。每个连接一收到数据就把自己放到第 8 个桶，然后在每秒的 timer 里把第一个桶里的连接断开，把这个空桶挪到队尾。这样大致可以做到 8 秒没有数据就超时断开连接。更重要的是，每次不用检查全部的连接，只要检查第一个桶里的连接，相当于把任务分散了。

7.10.1 timing wheel 原理

《Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility》³⁶ 这篇论文详细比较了实现定时器的各种数据结构，并提出了层次化的 timing wheel 与 hash timing wheel 等新结构。针对本节要解决的问题的特点，我们不需要实现一个通用的定时器，只用实现 simple timing wheel 即可。

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

以下是某一时刻 timing wheel 的状态（见图 7-42 的左图），格子里的数字是倒计时（与通常的 timing wheel 相反），表示这个格子（桶子）中连接的剩余寿命。

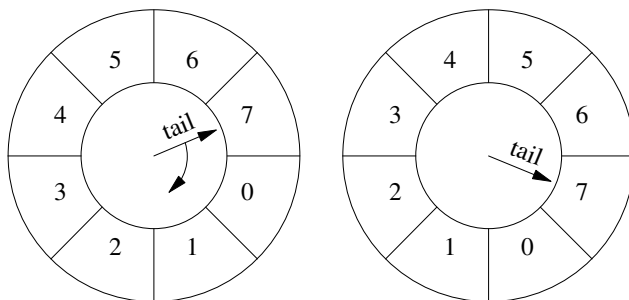


图 7-42

³⁶ <http://www.cs.columbia.edu/~nahum/w6998/papers/sosp87-timing-wheels.pdf>

1 秒以后（见图 7-42 的右图），tail 指针移动一格，原来四点钟方向的格子被清空，其中的连接已被断开。

连接超时被踢掉的过程

假设在某个时刻，conn 1 到达，把它放到当前格子中，它的剩余寿命是 7 秒（见图 7-43 的左图）。此后 conn 1 上没有收到数据。1 秒之后（见图 7-43 的右图），tail 指向下一个格子，conn 1 的剩余寿命是 6 秒。

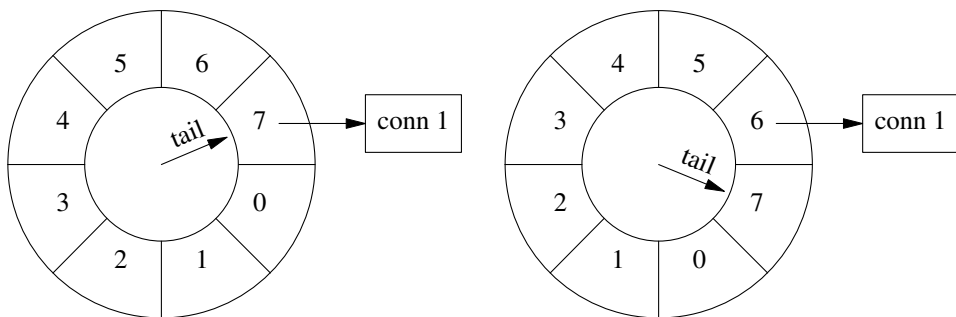


图 7-43

又过了几秒，tail 指向 conn 1 之前的那个格子，conn 1 即将被断开（见图 7-44 的左图）。下一秒（见图 7-44 的右图），tail 重新指向 conn 1 原来所在的格子，清空其中的数据，断开 conn 1 连接。

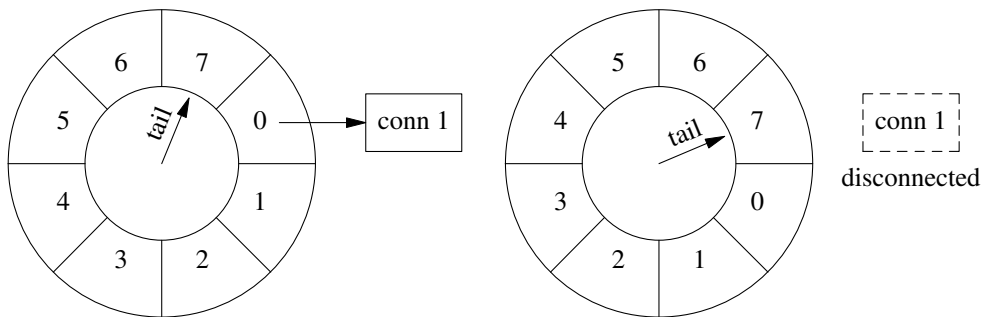


图 7-44

连接刷新

如果在断开 conn 1 之前收到数据，就把它移到当前的格子里。conn 1 的剩余寿命是 3 秒（见图 7-45 的左图），此时 conn 1 收到数据，它的寿命恢复为 7 秒（见图 7-45 的右图）。

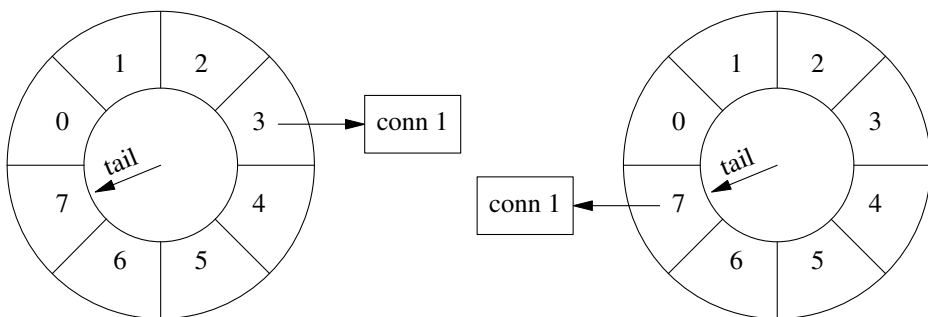


图 7-45

时间继续前进, conn 1 寿命递减, 不过它已经比第一种情况长寿了 (见图 7-46)。

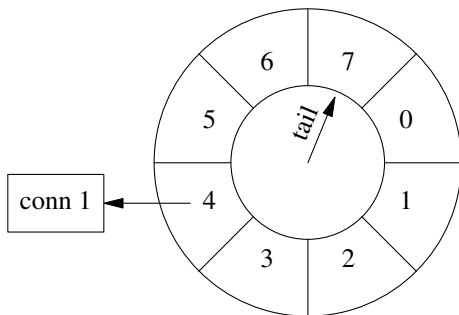


图 7-46

多个连接

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

比如一开始, conn 1 到达。随后, conn 2 到达 (见图 7-47), 这时候 tail 还没有移动, 两个连接位于同一个格子中, 具有相同的剩余寿命。(在图 7-47 中画成链表, 代码中是哈希表。)

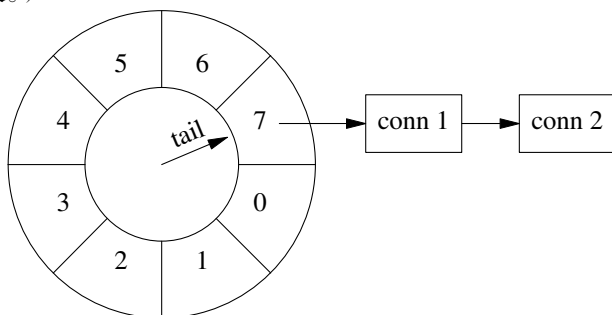


图 7-47

几秒之后，conn 1 收到数据，而 conn 2 一直没有收到数据，那么 conn 1 被移到当前的格子中。这时 conn 1 的预期寿命比 conn 2 长（见图 7-48）。

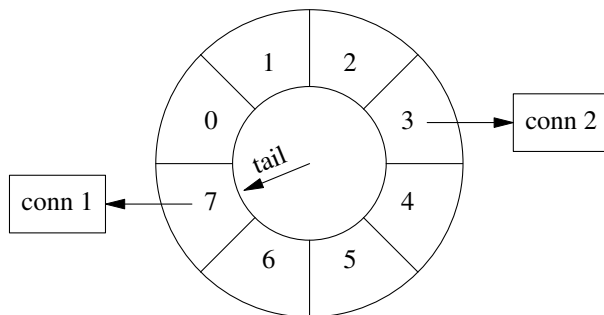


图 7-48

7.10.2 代码实现与改进

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

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

数据结构：（本节的代码压缩了单行缩进）

```

45 struct Entry : public muduo::copyable // 这是一个 type tag
46 {
47     explicit Entry(const WeakTcpConnectionPtr& weakConn)
48         : weakConn_(weakConn)
49     { }
50
51     ~Entry()
52     {
53         muduo::net::TcpConnectionPtr conn = weakConn_.lock();
54         if (conn)
55             conn->shutdown();
56     }
57
58     WeakTcpConnectionPtr weakConn_;
59 };
60
61 typedef boost::shared_ptr<Entry> EntryPtr;
62 typedef boost::weak_ptr<Entry> WeakEntryPtr;
63 typedef boost::unordered_set<EntryPtr> Bucket;
64 typedef boost::circular_buffer<Bucket> WeakConnectionList;

```

examples/idleconnection/echo.h

在实现中，为了简单起见，我们不会真的把一个连接从一个格子移到另一个格子，而是采用引用计数的办法，用 `shared_ptr` 来管理 `Entry`。如果从连接收到数据，就把对应的 `EntryPtr` 放到这个格子里，这样它的引用计数就递增了。当 `Entry` 的引用计数递减到零时，说明它没有在任何格子里出现，那么连接超时，`Entry` 的析构函数会断开连接。

注意在头文件中我们自己定义了 `shared_ptr<T>` 的 `hash` 函数，原因是直到 Boost 1.47.0 之前，`unordered_set<shared_ptr<T>>` 虽然可以编译通过，但是其 `hash_value` 是 `shared_ptr` 隐式转换为 `bool` 的结果。也就是说，如果不自定义 `hash` 函数，那么 `unordered_{set/map}` 会退化为链表。

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

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

```

15  EchoServer::EchoServer(EventLoop* loop,
16                          const InetAddress& listenAddr,
17                          int idleSeconds)
18      : loop_(loop),
19        server_(loop, listenAddr, "EchoServer"),
20        connectionBuckets_(idleSeconds)
21  {
22      server_.setConnectionCallback(
23          boost::bind(&EchoServer::onConnection, this, _1));
24      server_.setMessageCallback(
25          boost::bind(&EchoServer::onMessage, this, _1, _2, _3));
26      loop->runEvery(1.0, boost::bind(&EchoServer::onTimer, this));
27      connectionBuckets_.resize(idleSeconds);
28  }

```

examples/idleconnection/echo.cc

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

```

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

```

在连接建立时，创建一个 `Entry` 对象，把它放到 `timing wheel` 的队尾。另外，我们还需要把 `Entry` 的弱引用保存到 `TcpConnection` 的 `context` 里，因为在收到数据的

时候还要用到 Entry。（思考题：如果 TcpConnection::setContext 保存的是强引用 EntryPtr，会出现什么情况？）

```

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

```

examples/idleconnection/echo.cc

在收到消息时，从 TcpConnection 的 context 中取出 Entry 的弱引用，把它提升为强引用 EntryPtr，然后放到当前的 timing wheel 队尾。（思考题：为什么要把 Entry 作为 TcpConnection 的 context 保存，如果这里再创建一个新的 Entry 会有什么后果？）

```

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

```

examples/idleconnection/echo.cc

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

希望本节内容有助于你理解 `shared_ptr` 和 `weak_ptr` 的引用计数。

改进

在现在的实现中，每次收到消息都会往队尾添加 `EntryPtr`（当然，`hash set` 会帮我们去重（deduplication））。一个简单的改进措施是，在 `TcpConnection` 里保存“最后一次往队尾添加引用时的 `tail` 位置”，收到消息时先检查 `tail` 是否变化，若无变化则不重复添加 `EntryPtr`，若有变化则把 `EntryPtr` 从旧的 `Bucket` 移到当前队尾 `Bucket`。这样或许能提高空间和时间效率。以上改进留作练习。

另外一个思路是“选择排序”：使用链表将 `TcpConnection` 串起来，`TcpConnection` 每次收到消息就把自己移到链表末尾，这样链表是按接收时间先后排序的。再用一个定时器定期从链表前端查找并踢掉超时的连接。代码示例位于同一目录。

7.11 简单的消息广播服务

本节介绍用 `muduo` 实现一个简单的 topic-based 消息广播服务，这其实是“聊天室”的一个简单扩展，不过聊天的不是人，而是分布式系统中的程序。本节的代码见 `examples/hub`。

在分布式系统中，除了常用的 `end-to-end` 通信，还有一对多的广播通信。一提到“广播”，或许会让人联想到 IP 多播或 IP 组播，这不是本节的主题。本节将要谈的是基于 TCP 协议的应用层广播。示意图如图 7-49 所示。

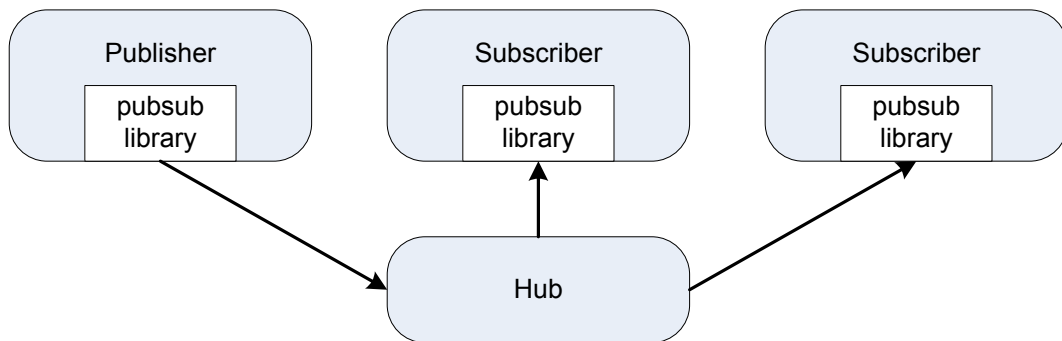


图 7-49

图 7-49 中的圆角矩形代表程序，“Hub”是一个服务程序，不是网络集线器，它起到类似集线器的作用，故而得名。Publisher 和 Subscriber 通过 TCP 协议与 Hub 程

序通信。Publisher 把消息发到某个 topic 上，Subscriber 订阅该 topic，然后就能收到消息。即 Publisher 借助 Hub 把消息广播给了一个或多个 Subscriber。这种 pub/sub 结构的好处在于可以增加多个 Subscriber 而不用修改 Publisher，一定程度上实现了“解耦”（也可以看成分布式的 Observer pattern）。由于走的是 TCP 协议，广播是基本可靠的，这里的“可靠”指的是“比 UDP 可靠”，不是“完全可靠”。³⁷（思考：如何避免 Hub 成为 single point of failure？）

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

应用层广播在分布式系统中用处很大，这里略举几例。

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

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

协议

为了简单起见，muduo 的 Hub 示例采用以“\r\n”分界的文本协议，这样用 telnet 就能测试 Hub。协议只有以下三个命令：

- sub <topic>\r\n
该命令表示订阅 <topic>，以后该 topic 有任何更新都会发给这个 TCP 连接。在 sub 的时候，Hub 会把该 <topic> 上最近的消息发给此 Subscriber。
- unsub <topic>\r\n
该命令表示退订 <topic>。

³⁷ “可靠广播、原子广播”在分布式系统中有重大意义，是以 replicated state machine 方式实现可靠的分布式服务的基础。“可靠广播”涉及 consensus 算法，超出了本书的范围。

- `pub <topic>\r\n<content>\r\n`
往 `<topic>` 发送消息，内容为 `<content>`。所有订阅了此 `<topic>` 的 Subscriber 会收到同样的消息 “`pub <topic>\r\n<content>\r\n`”。

代码

muduo 示例中的 Hub 分为几个部分：

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

一个程序可以既是 Publisher 又是 Subscriber，而且 pubsub 库只用一个 TCP 连接（这样 failover 比较简便）。使用范例如下所示。

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

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

多线程的高效广播

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

7.12 “串并转换”连接服务器及其自动化测试

本节介绍如何使用 test harness 来测试一个具有内部逻辑的网络服务程序。这是一个既扮演服务端，又扮演客户端的网络程序。代码见 `examples/multiplexer`。

云风在他的博客中提到了网游连接服务器的功能需求³⁸，我用 C++ 初步实现了这些需求，并为之编写了配套的自动化 test harness，作为 muduo 网络库的示例。

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

功能需求

这个连接服务器把多个客户连接汇聚为一个内部 TCP 连接，起到“数据串并转换”的作用，让 backend 的逻辑服务器专心处理业务，而无须顾及多连接的并发性。系统的框图如图 7-50 所示。

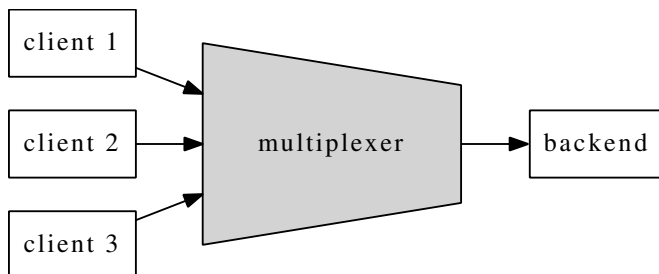


图 7-50

³⁸ 在 http://blog.codingnow.com/2010/11/go_prime.html 搜“练手项目”。

这个连接服务器的作用与数字电路中的数据选择器（multiplexer）类似（见图 7-51），所以我把它命名为 multiplexer。（其实 IO multiplexing 也是取的这个意思，让一个 thread-of-control 能有选择地处理多个 IO 文件描述符。）

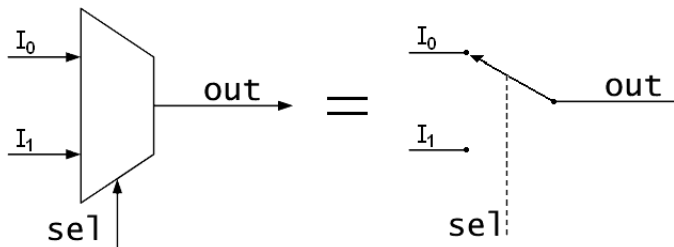


图 7-51（本图取自 wikipedia，是 public domain 版权）

实现

multiplexer 的功能需求不复杂，无非是在 backend connection 和 client connections 之间倒腾数据。对每个新 client connection 分配一个新的整数 id，如果 id 用完了，则断开新连接（这样通过控制 id 的数目就能控制最大连接数）。另外，为了避免 id 过快地被复用（有可能造成 backend 串话），multiplexer 采用 queue 来管理 free id，每次从队列的头部取 id，用完之后放回 queue 的尾部。具体来说，主要是处理四种事件：

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

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

在实现的时候有以下两点值得注意。

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

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

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

```

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

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

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

    boost::any* getMutableContext()
    { return &context_; }

    // ...

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

typedef boost::shared_ptr<TcpConnection> TcpConnectionPtr;

```

muduo/net/TcpConnection.h

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

```

117 void onClientMessage(const TcpConnectionPtr& conn, Buffer* buf, Timestamp)
118 {

```

examples/multiplexer/multiplexer_simple.cc

```

119     if (!conn->getContext().empty())
120     {
121         int id = boost::any_cast<int>(conn->getContext());
122         sendBackendBuffer(id, buf);
123     }
124     else
125     {
126         buf->retrieveAll();
127         // error handling
128     }
129 }

```

examples/multiplexer/multiplexer_simple.cc

TcpConnection 的生命期如何管理？ 由于 client connection 是动态创建并销毁的，其生与灭完全由客户决定，如何保证 backend 想向它发送数据的时候，这个 TcpConnection 对象还活着？解决思路是用 reference counting。当然，不用自己写，用 boost::shared_ptr 即可。TcpConnection 是 muduo 中唯一默认采用 shared_ptr 来管理生命期的对象，盖由其动态生命期的本质决定。更多内容请参考第 1 章。

multiplexer 采用二进制协议，如何测试呢？

自动化测试

multiplexer 是 muduo 网络编程示例中第一个具有 non-trivial 业务逻辑的网络程序，根据 §9.7 “分布式程序的自动化回归测试”的思路，我为它编写了测试夹具 (test harness)。代码见 examples/multiplexer/harness/。

这个 test harness 采用 Java 编写，用的是 Netty 网络库。这个 test harness 要同时扮演 clients 和 backend，也就是既要主动发起连接，也要被动接受连接。而且，test harness 与 multiplexer 的启动顺序是任意的，如何做到这一点请阅读代码。结构如图 7-52 所示。

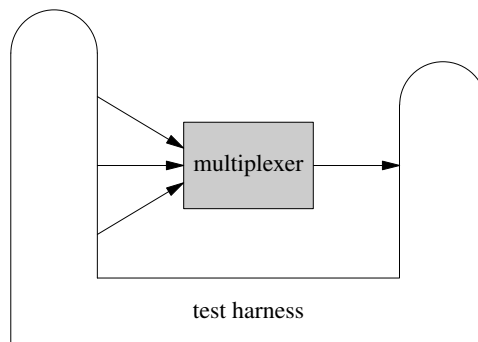


图 7-52

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

这里的几个 test cases 都是用 Java 直接写的，如果有必要，也可以采用 Groovy 来编写，这样可以在不重启 test harness 的情况下随时修改、添加 test cases。具体做法见笔者的博客《“过家家”版的移动离线计费系统实现》³⁹。

将来的改进

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

- 增加“backend 发送指令断开 client connection”的功能。有了自动化测试，这个新功能可以被单独测试（开发者测试），而不需要真正的 backend 参与进来。
- 将 multiplexer 改用多线程重写。有了自动化回归测试，我们不用担心破坏原有的功能，可以放心大胆地重写。而且由于 test harness 是从外部测试，不是单元测试，重写 multiplexer 的时候不用动 test cases，这样保证了测试的稳定性。另外，这个 test harness 稍加改进还可以进行 stress testing，既可用于验证多线程 multiplexer 的正确性，亦可对比其相对单线程版的效率提升。

7.13 socks4a 代理服务器

本节介绍用 muduo 实现一个简单的 socks4a 代理服务器（examples/socks4a/）。

7.13.1 TCP 中继器

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

一般情况下，客户端程序直接连接服务端，如图 7-53 所示。



图 7-53

³⁹ <http://www.cnblogs.com/Solstice/archive/2011/04/22/2024791.html>

有时候,我们想在 client 和 server 之间放一个中继器 (relay),把 client 与 server 之间的通信内容记录下来。这时用 tcpdump 是最方便省事的,但是 tcpdump 需要 root 权限,万一拿不到权限呢? 穷人有穷人的办法,自己写一个 TcpRelay,让 client 连接 TcpRelay,再让 TcpRelay 连接 server,如图 7-54 中的 T 型结构, TcpRelay 扮演了类似 proxy 的角色。

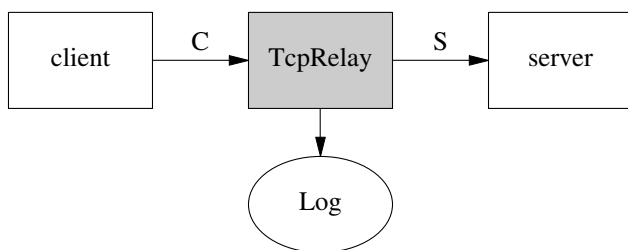


图 7-54

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

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

1. 建立连接。为了真实模拟 client, TcpRelay 在 accept 连接 C 之后才向 server 发起连接 S,那么在 S 建立起来之前,从 C 收到数据怎么办? 要不要暂存起来?
2. 并发连接的管理。图 7-54 中只画出了一个 client,实际上 TcpRelay 可以服务多个 client,左右两边这些并发连接如何管理,如何防止串话 (cross talk)?
3. 连接断开。client 和 server 都可能主动断开连接。当 client 主动断开连接 C 时, TcpRelay 应该立刻断开 S。当 server 主动断开连接 S 时, TcpRelay 应立刻断开 C。这样才能比较精确地模拟 client 和 server 的行为。在关闭连接的一刹那,又有新的 client 连接进来,复用了刚刚 close 的 fd 号码,会不会造成串话? 万一 client 和 server 几乎同时主动断开连接, TcpRelay 如何应对?
4. 速度不匹配。如果连接 C 的带宽是 100kB/s,而连接 S 的带宽是 10MB/s,不巧 server 是个 chargen 服务,会全速发送数据,那么会不会撑爆 TcpRelay 的 buffer? 如何限速? 特别是在使用 non-blocking IO 和 level-trigger polling 的时候如何限制读取数据的速度?

在看 muduo 的实现之前,请读者思考:如果用 Sockets API 来实现 TcpRelay,如何解决以上这些问题。(如果真要实现这么一个功能,可以试试 splice(2) 系统调用。)

如果用传统多线程阻塞 IO 的方式来实现 TcpRelay 一点也不难，好处是自动解决了速度不匹配的问题，Python 代码如下。这个实现功能上没有问题，但是并发度就高不到哪儿去了。注意以下代码会一个字节一个字节地转发数据，每两个字节之间间隔 1ms，可以用于测试网络程序的消息解码功能（codec）是否完善。

```

1  #!/usr/bin/python
2
3  import socket, thread, time
4
5  listen_port = 3007
6  connect_addr = ('localhost', 2007)
7  sleep_per_byte = 0.0001
8
9  def forward(source, destination):
10     source_addr = source.getpeername()
11     while True:
12         data = source.recv(4096)
13         if data:
14             for i in data:
15                 destination.sendall(i)
16                 time.sleep(sleep_per_byte)
17         else:
18             print 'disconnect', source_addr
19             destination.shutdown(socket.SHUT_WR)
20             break
21
22  serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23  serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
24  serversocket.bind(('', listen_port))
25  serversocket.listen(5)
26
27  while True:
28     (clientsocket, address) = serversocket.accept()
29     print 'accepted', address
30     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
31     sock.connect(connect_addr)
32     print 'connected', sock.getpeername()
33     thread.start_new_thread(forward, (clientsocket, sock))
34     thread.start_new_thread(forward, (sock, clientsocket))

```

TcpRelay 的实现很简单，只有几十行代码（examples/socks4a/tcprelay.cc），主要逻辑都在 Tunnel class 里（examples/socks4a/tunnel.h）。这个实现很好地解决了前三个问题，第四个问题的解法比较粗暴，用的是 HighWaterMarkCallback，如果发送缓冲区堆积的数据大于 10MiB 就断开连接（更好的办法见 §8.9.3）。TcpRelay 既是服务端，又是客户端，在阅读代码的时候要注意 onClientMessage() 处理的是从 server 发来的消息，表示它作为客户端（client）收到的消息，这与前面的 multiplexer 正好相反。

7.13.2 socks4a 代理服务器

socks4a 的功能与 TcpRelay 非常相似，也是把连接 C 上收到的数据发给连接 S，同时把连接 S 上收到的数据发给连接 C。它与 TcpRelay 的区别在于，TcpRelay 固定连到某个 server 地址，而 socks4a 允许 client 指定要连哪个 server。在 accept 连接 C 之后，socks4a server 会读几个字节，以了解 server 的地址，再发起连接 S。socks4a 的协议非常简单，请参考维基百科⁴⁰。

muduo 的 socks4a 代理服务器的实现在 examples/socks4a/socks4a.cc，它也使用了 Tunnel class。与 TcpRelay 相比，只多了解析 server 地址这一步骤。目前 DNS 地址解析这一步用的是阻塞的 gethostbyname() 函数，在真正的系统中，应该换成非阻塞的 DNS 解析，可参考 §7.15。muduo 的这个 socks4a 是个标准的网络服务，可以供 Web 浏览器使用（我正是这么测试它的）。

7.13.3 $N:1$ 与 $1:N$ 连接转发

云风在《写了一个 proxy 用途你懂的》⁴¹中写了一个 TCP 隧道 tunnel，程序由三部分组成： $N:1$ 连接转发服务， $1:N$ 连接转发服务，socks 代理服务。

我仿照他的思路，用 muduo 实现了这三个程序。不同的是，我没有做数据混淆，所以功能上有所减弱。

- $N:1$ 连接转发服务就是 §7.12 中的 multiplexer（数据选择器）。
- $1:N$ 连接转发服务是云风文中提到的 backend，一个数据分配器（demultiplexer），代码在 examples/multiplexer/demux.cc。
- socks 代理服务正是 §7.13.2 实现的 socks4a。

有兴趣的读者可以把这三个程序级联起来试一试。

7.14 短址服务

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

⁴⁰ http://en.wikipedia.org/wiki/SOCKS#SOCKS_4a

⁴¹ <http://blog.codingnow.com/2011/05/xtunnel.html>

```

std::map<string, string> redirections; // URL 转发表

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

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

    std::map<string, string>::const_iterator it = redirections.find(req.path());
    if (it != redirections.end()) // 如果找到了短址
    {
        resp->setStatusCode(HttpResponse::k301MovedPermanently);
        resp->setStatusMessage("Moved Permanently");
        resp->addHeader("Location", it->second); // 转发到 it->second 地址
        // resp->setCloseConnection(true);
    }
    // ...
}

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

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

```

muduo 并没有为短连接 TCP 服务优化，无法发挥多核优势。一种真正高效的优化手段是修改 Linux 内核，例如 Google 的 SO_REUSEPORT 内核补丁⁴²。

读者可以试试建立一个 loop 转发，例如 `"/1" → "/2" → "/3" → "/1"`，看看浏览器反应如何。

7.15 与其他库集成

前面介绍的网络应用例子都是直接用 muduo 库收发网络消息，也就是主要介绍 TcpConnection、TcpServer、TcpClient、Buffer 等 class 的使用。本节将稍微深入其内部，介绍 Channel class 的用法，通过它可以把其他一些现成的网络库融入 muduo 的 event loop 中。

⁴² http://linux.dell.com/files/presentations/Linux_Plumbers_Conf_2010/Scaling_techniques_for_servers_with_high_connection%20rates.pdf

Channel class 是 IO 事件回调的分发器 (dispatcher)，它在 handleEvent() 中根据事件的具体类型分别回调 ReadCallback、WriteCallback 等，代码见 §8.1.1。每个 Channel 对象服务于一个文件描述符，但并不拥有 fd，在析构函数中也不会 close(fd)。Channel 也使用 muduo 一贯的 boost::function 来表示函数回调，它不是基类⁴³。这样用户代码不必继承 Channel，也无须 override 虚函数。

```

class Channel : boost::noncopyable
{
public:
    typedef boost::function<void()> EventCallback;
    typedef boost::function<void(Timestamp)> ReadEventCallback;

    Channel(EventLoop* loop, int fd);
    ~Channel();

    void setReadCallback(const ReadEventCallback& cb);
    void setWriteCallback(const EventCallback& cb);
    void setCloseCallback(const EventCallback& cb);
    void setErrorCallback(const EventCallback& cb);

    void enableReading();
    // void disableReading(); // 暂时没有用到
    void enableWriting();
    void disableWriting();
    void disableAll();

    void handleEvent(Timestamp receiveTime); // 由 EventLoop::loop() 调用

    /// Tie this channel to the owner object managed by shared_ptr,
    /// prevent the owner object being destroyed in handleEvent.
    void tie(const boost::shared_ptr<void>&); // tie() 的例子见 7.15.3 节

    int fd() const; // obvious
    void remove(); // loop_->removeChannel(this);

    // .....
};

```

Channel 与 EventLoop 的内部交互有两个函数 EventLoop::updateChannel(Channel*) 和 EventLoop::removeChannel(Channel*)。客户需要在 Channel 析构前自己调用 Channel::remove()。

后面我们将通过一些实例来介绍 Channel class 的使用。

⁴³ 相关讨论见 <http://www.cppblog.com/Solstice/archive/2012/07/01/181058.aspx> 后面的评论。

7.15.1 UDNS

UDNS⁴⁴ 是一个 stub⁴⁵ DNS 解析器，它能够异步地发起 DNS 查询，再通过回调函数通知结果。UDNS 在设计的时候就考虑到了配合（融入）主程序现有的基于 select/poll/epoll 的 event loop 模型，因此它与 muduo 的配接相对较为容易。由于 License 限制，本节的代码位于单独的项目中：<https://github.com/chenshuo/muduo-udns>。

muduo-udns 由三部分组成，一是 udns-0.2 源码⁴⁶；二是 UDNS 与 muduo 的配接器（adapter），即 Resolver class，位于 Resolver.{h,cc}；三是简单的测试 dns.cc，展示 Resolver 的使用。前两部分构成了 muduo-udns 程序库。

先看 Resolver class 的接口（Resolver.h）：

```
class Resolver : boost::noncopyable
{
public:
    typedef boost::function<void(const InetAddress&)> Callback;

    Resolver(EventLoop* loop);
    Resolver(EventLoop* loop, const InetAddress& nameServer);
    ~Resolver();

    void start();

    bool resolve(const StringPiece& hostname, const Callback& cb);

    // ...
};
```

其中第一个构造函数会使用系统默认的 DNS 服务器地址，第二个构造函数由用户指明 DNS 服务器的 IP 地址（见后面的练习 1）。用户最关心的是 resolve() 函数，它会回调用户的 Callback。

在介绍 Resolver 的实现之前，先来看它的用法（dns.cc），下面这段代码同时解析三个域名，并在 stdout 输出结果。注意回调函数只提供解析后的地址，因此 resolveCallback 需要自己设法记住域名，这里我用的是 boost::bind。

```
void resolveCallback(const string& host, const InetAddress& addr)
{
    LOG_INFO << "resolved " << host << " -> " << addr.toIp();
}
```

⁴⁴ <http://www.corpit.ru/mjt/udns.html>

⁴⁵ stub 的意思是只会查询一个 DNS 服务器，而不会递归地（recursive）查询多个 DNS 服务器，因此适合在公司内网使用（http://en.wikipedia.org/wiki/Domain_Name_System#DNS_resolvers）。

⁴⁶ Ubuntu 和 Debian 都不包含 UDNS 0.2 软件包，因此必须连同上游源码一起发布。

```
void resolve(Resolver* res, const string& host)
{
    res->resolve(host, boost::bind(&resolveCallback, host, _1));
}

int main(int argc, char* argv[])
{
    EventLoop loop;
    Resolver resolver(&loop);
    resolver.start();

    resolve(&resolver, "chenshuo.com");
    resolve(&resolver, "www.example.com");
    resolve(&resolver, "www.google.com");

    loop.loop(); // 开始事件循环
}
```

由于是异步解析，因此输出结果的顺序和提交请求的顺序不一定一致，例如：

```
20120822 04:46:39.945033Z 15726 INFO resolved www.google.com -> 74.125.71.104
20120822 04:46:41.944464Z 15726 INFO resolved chenshuo.com -> 173.212.209.144
20120822 04:46:42.068084Z 15726 INFO resolved www.example.com -> 192.0.43.10
```

UDNS 与 muduo Resolver 的交互过程如下：

1. 初始化 `dns_ctx*` 之后，`Resolver::start()` 调用 `dns_open()` 获得 UDNS 使用的文件描述符，并通过 `muduo Channel` 观察其可读事件。由于 UDNS 始终只使用一个 `socket fd`，只观察一个事件，因此特别容易和现有的 `event loop` 集成。
2. 在解析域名时（`Resolver::resolve()`），调用 `dns_submit_a4()` 发起解析，并通过 `dns_timeouts()` 获得超时的秒数，使用 `EventLoop::runAfter()` 注册单次定时器回调。
3. 在 `fd` 可读时（`Resolver::onRead()`），调用 `dns_ioevent()`。如果 DNS 解析成功，会回调 `Resolver::dns_query_a4()` 通知解析的结果，继而调用 `Resolver::onQueryResult()`，后者会回调用户 `Callback`。
4. 在超时后（`Resolver::onTimer()`），调用 `dns_timeouts()`，必要时继续注册下一次定时器回调。

可见 UDNS 是一个设计良好的库，可与现有的 `event loop` 很好地结合。UDNS 使用定时器的原因是 UDP 可能丢包，因此程序必须自己处理超时重传。

`Resolve class` 不是线程安全的，客户代码只能在 `EventLoop` 所属的线程调用它的 `Resolver::resolve()` 成员函数，解析结果也是由这个线程回调客户代码。这个函数通过 `loop_->assertInLoopThread()`；来确保不被误用。

Linux 多线程服务端编程：使用 muduo C++ 网络库

C++ 程序与 C 语言函数库交互的一个难点在于资源管理，muduo-udns 不得已使用了手工 new/delete 的做法，每次解析会在堆上创建 QueryData 对象，这样在 UDNS 回调 Resolver::dns_query_a4() 时才知道该回调哪个用户 Callback。

练习 1：补充构造函数 Resolver(EventLoop* loop, const InetAddress& nameServer) 的实现。可利用文档⁴⁷介绍的 dns_add_serv_s() 函数。

练习 2：用 muduo-udns 改进 §7.13 的 socks4a 服务器，替换其中阻塞的 get-hostbyname() 函数调用，实现完全的无阻塞服务。

7.15.2 c-ares DNS

c-ares DNS⁴⁸ 是一款常用的异步 DNS 解析库，§6.2 介绍了它的安装方法，本节将简要介绍其与 muduo 的集成。示例代码位于 examples/cdns，代码结构与 §7.15.1 的 UDNS 非常相似。Resolver.{h,cc} 是 c-ares DNS 与 muduo 的配接器 (adapter)，即 udns::Resolver class；dns.cc 是简单的测试，展示 Resolver 的使用。c-ares DNS 的选项非常多⁴⁹，本节只是展示其与 muduo EventLoop 集成的基本做法，cdns::Resolver 并没有暴露其全部功能。

cdns::Resolver 的接口和用法与前面 UDNS Resolver 相同，只是少了 start() 函数，此处不再重复举例。

cdns::Resolver 的实现与前面 UDNS Resolver 很相似：

1. Resolver::resolve() 调用 ares_gethostbyname() 发起解析，并通过 ares_timeout() 获得超时的秒数，注册定时器。
2. 在 fd 可读时 (Resolver::onRead())，调用 ares_process_fd()。如果 DNS 解析成功，会回调 Resolver::ares_host_callback()⁵⁰ 通知解析的结果，继而调用 Resolver::onQueryResult()，后者会回调用户 Callback。
3. 在超时后 (Resolver::onTimer())，调用 ares_process_fd() 处理这一事件，并再次调用 dns_timeouts() 获得下一次超时的间隔，必要时继续注册下一次定时器回调。

cdns::Resolver 的线程安全性与 UDNS Resolver 相同。

⁴⁷ <http://www.corpit.ru/mjt/udns/udns.3.html>

⁴⁸ <http://c-ares.haxx.se/>

⁴⁹ 功能也比 UDNS 强大，例如可以读取 /etc/hosts。udns::Resolver 的构造函数有选项可禁用此功能。

⁵⁰ ares_host_callback() 相当于前面 UDNS 的 dns_query_a4() 回调。

与 UDNS 不同, c-ares DNS 会用到不止一个 socket 文件描述符⁵¹, 而且既会用到 fd 可读事件, 又会用到 fd 可写事件, 因此 `cdns::Resolver` 的代码比 UDNS 要复杂一些。`Resolver::ares_sock_create_callback()` 是新建 socket fd 的回调函数, 其中会调用 `Resolver::onSockCreate()` 来创建 `Channel` 对象, 这正是 `Resolver` 没有 `start()` 成员函数的原因。`Resolver::ares_sock_state_callback()` 是变更 socket fd 状态的回调函数, 会通知该观察哪些 IO 事件 (可读 and/or 可写)。

练习 3: 阅读源码并测试 c-ares DNS 什么时候需要观察 “fd 可写” 事件, 然后补充完整 `Resolver::onSockStateChange()`。

练习 4: 修改 `Callback` 的原型, 让 `Resolver` 能返回地址列表 (`std::vector<Inet-Address>`), 这个练习同样适用于 §7.15.1 的 UDNS。

练习 5: 为 `libunbound`⁵² 编写类似的 `muduo adapter`。注意它似乎没有使用 `timeout`, 很奇怪。

7.15.3 curl

`libcurl` 是一个常用的 HTTP 客户端库⁵³, 可以方便地下载 HTTP 和 HTTPS 数据。`libcurl` 有两套接口, `easy` 和 `multi`, 本节介绍的是使用其 `multi` 接口⁵⁴ 以达到单线程并发访问多个 URL 的效果。`muduo` 与 `libcurl` 搭配的例子见 `examples/curl`, 其中包含单线程多连接并发下载同一文件的示例, 即单线程实现的 “多线程下载器”。

`libcurl` 融入 `muduo EventLoop` 的复杂度比前面两个 DNS 库都更高, 一方面因为它本身的功能丰富, 另一方面也因为它的接口设计更偏重传统阻塞 IO (它原本是从 `curl(1)` 这个命令行工具剥离出来的), 在事件驱动方面的调用、回调、传参都比较烦琐。这里不去详细解释每一个函数的作用, 想必读者在读过前两节之后已经对 `Channel` 的用法有了基本的了解, 对照 `libcurl` 文档和 `muduo` 代码就能搞明白。

练习 6: 修改 `curl::Request::DataCallback` 的原型, 改为以 `muduo::net::Buffer*` 为参数, 方便用户使用。这需要在 `curl::Request` 中增加 `Buffer` 成员。

第 1 章我们探讨了多线程程序中的对象生命期管理技术。在单线程事件驱动的程序中, 对象的生命期管理有时也不简单。比方说图 7-55 展示的例子, 对方断开 TCP 连接, 这个 IO 事件会触发 `Channel::handleEvent()` 调用, 后者会回调用户提供的

⁵¹ 因为 DNS 解析时, 如果 UDP 响应发生消息截断, 会改用 TCP 重发请求。

⁵² <http://www.unbound.net/documentation/libunbound.html>

⁵³ 也可以访问 FTP 服务器。

⁵⁴ <http://curl.haxx.se/libcurl/c/libcurl-multi.html>

CloseCallback, 而用户代码在 onclose() 中有可能析构 Channel 对象, 这就造成了灾难。等于说 Channel::handleEvent() 执行到一半的时候, 其所属的 Channel 对象本身被销毁了。这时程序立刻 core dump 就是最好的结果了。

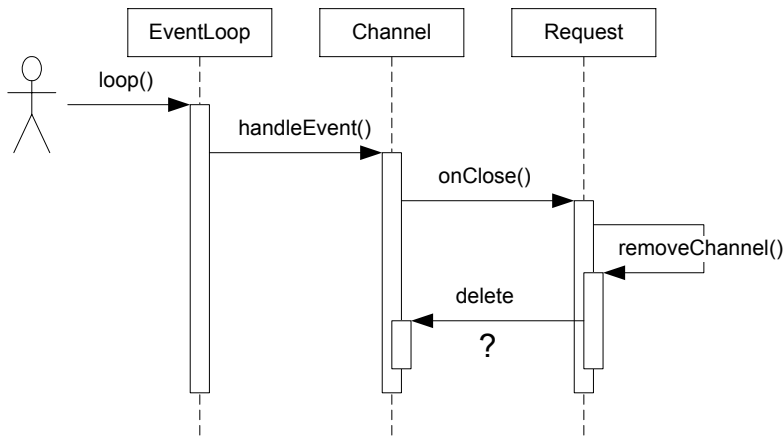


图 7-55

muduo 的解决办法是提供 Channel::tie(const boost::shared_ptr<void>&) 这个函数, 用于延长某些对象⁵⁵的生命期, 使之长过 Channel::handleEvent() 函数。这也是 muduo TcpConnection 采用 shared_ptr 管理对象生命期的原因之一; 否则的话, Channel::handleEvent() 有可能引发 TcpConnection 析构, 继而把当前 Channel 对象也析构了, 引起程序崩溃。

第三方库与 muduo 集成的另外一个问题是对 IO 事件变化的理解可能不一致。拿 libcurl 来说, 它会在某个文件描述符需要关注的 IO 事件发生变化时通知外围的 event loop 库, 比方说原来关注 POLLIN, 现在关注 (POLLIN | POLLOUT), muduo 在 Curl::socketCallback 回调函数中会相应地调用 Channel::enableWriting(), 能正确处理这种变化。

不幸的是, libcurl 在与 c-ares DNS 配合⁵⁶的时候会出现与 muduo 不兼容的现象。libcurl 在访问 URL 的时候先要解析其中的域名, 然后再对那个 Web 服务器发起 TCP 连接。在与 c-ares DNS 搭配时会出现一种情况: c-ares DNS 解析域名用到的与 DNS 服务器通信的 socket fd₁ 和 libcurl 对 Web 服务器发起 TCP 连接的 fd₂ 恰好相等, 即 fd₁ == fd₂。原因是 POSIX 操作系统总是选用当前最小可用的文件描述符, 当

⁵⁵ 可以是 Channel 对象, 也可以是其 owner 对象。

⁵⁶ 这两个库是同一个作者, libcurl 默认会用 gethostbyname() 执行同步 DNS 解析, 在有 c-ares DNS 的时候用它执行异步 DNS 解析。

DNS 解析完成后, libcurl 内部使用的 c-ares DNS 会关闭 `fd1`, libcurl 随后再立刻新建一个 TCP socket `fd2`, 它有可能恰好复用了 `fd1` 的值。

但这时 libcurl 不会认为文件描述符或其关注的 IO 事件发生了变化, 也就不会通知 muduo 去销毁并新建 Channel 对象。这种做法与传统的基于 `select(2)` 和 `poll(2)` 的 event loop 配合不会有问题, 因为 `select(2)` 和 `poll(2)` 是上下文无关的, 每次都从输入重建要关注的文件描述符列表。但是在与 `epoll(4)` 配合的时候就有问题了, 关闭 `fd1` 会使得 `epoll` 从关注列表 (watch list) 中移除 `fd1` 的条目, 新建的同名 `fd2` 却没有机会加入 IO 事件 watch list, 也就不会收到任何 IO 事件通知。这个问题无法在 muduo 内部修复, 只能修改上游的程序库。

另外一个问题是 libcurl 在通知 muduo 取消关注某个 fd 的时候已经事先关闭了它, 这将造成 muduo 调用 `::epoll_ctl(epollfd_, EPOLL_CTL_DEL, fd, NULL)` 时会返回错误, 因为关闭文件描述符已经就把它从 `epoll` watch list 中除掉了。为了应对这种情况, 我不得已更改了 `EPollPoller::update()` 的错误处理, 放宽检查。

7.15.4 更多

除了前面举的几个例子, muduo 当然还可以将其他涉及网络 IO 的库融入其 EventLoop/Channel 框架, 我能想到的有:

- libmicrohttpd —— 可嵌入的 HTTP 服务器。
- libpg —— PostgreSQL 的官方客户端库。
- libdrizzle —— MySQL 的非官方客户端库。
- QuickFIX —— 常用的 FIX 消息库。

在有具体应用场景的时候, 我多半会为之提供 muduo adapter, 也欢迎用户贡献有关补丁。

另外一个扩展思路是, 对每个 TCP 连接创建一个 lua state, 用 muduo 为 lua 提供通信机制。然后用 lua 来编写业务逻辑, 这也可以做到在线更改逻辑而不重启进程。就像 OpenResty⁵⁷ 和云风的 skynet⁵⁸ 那样。这种做法还可以利用 coroutine 来简化业务逻辑的实现。

⁵⁷ <http://openresty.org/>

⁵⁸ <https://github.com/cloudwu/skynet>

第 8 章

muduo 网络库设计与实现

本章从零开始逐步实现一个类似 muduo 的基于 Reactor 模式的 C++ 网络库，大体反映了 muduo 网络相关部分的开发过程。本章大致分为三段，为了与代码匹配，本章的小节从 0 开始编号。注意本章呈现的代码与现在 muduo 的代码略有出入。

1. §8.0 至 §8.3 介绍 Reactor 模式的现代 C++ 实现，包括 EventLoop、Poller、Channel、TimerQueue、EventLoopThread 等 class；
2. §8.4 至 §8.9 介绍基于 Reactor 的单线程、非阻塞、并发 TCP server 网络编程，主要介绍 Acceptor、Socket、TcpServer、TcpConnection、Buffer 等 class；
3. §8.10 至 §8.13 是提高篇，介绍 one loop per thread 的实现（用 EventLoopThreadPool 实现多线程 TcpServer），Connector 和 TcpClient class，还有用 epoll(4) 替换 poll(2) 作为 Poller 的 IO multiplexing 机制等。

本章的代码位于 recipes/reactor/，会直接使用 muduo/base 中的日志、线程等基础库。

8.0 什么都不做的 EventLoop

首先定义 EventLoop class 的基本接口：构造函数、析构函数、loop() 成员函数。注意 EventLoop 是不可拷贝的，因此它继承了 boost::noncopyable。muduo 中的大多数 class 都是不可拷贝的，因此以后只会强调某个 class 是可拷贝的。

```
16 class EventLoop : boost::noncopyable
17 {
18     public:
19
20         EventLoop();
21         ~EventLoop();
22
23         void loop();
```

reactor/s00/EventLoop.h

```

24
25     void assertInLoopThread()
26     {
27         if (!isInLoopThread())
28         {
29             abortNotInLoopThread();
30         }
31     }
32
33     bool isInLoopThread() const { return threadId_ == CurrentThread::tid(); }
34
35 private:
36     void abortNotInLoopThread();
37
38     bool looping_; /* atomic */
39     const pid_t threadId_;
40 };

```

reactor/s00/EventLoop.h

one loop per thread 顾名思义每个线程只能有一个 EventLoop 对象，因此 EventLoop 的构造函数会检查当前线程是否已经创建其他 EventLoop 对象，遇到错误就终止程序（LOG_FATAL）。EventLoop 的构造函数会记住本对象所属的线程（threadId_）。创建了 EventLoop 对象的线程是 **IO 线程**，其主要功能是运行事件循环 EventLoop::loop()。EventLoop 对象的生命期通常和其所属的线程一样长，它不必是 heap 对象。

```

17 __thread EventLoop* t_loopInThisThread = 0;
18
19 EventLoop::EventLoop()
20 : looping_(false),
21   threadId_(CurrentThread::tid())
22 {
23     LOG_TRACE << "EventLoop created " << this << " in thread " << threadId_;
24     if (t_loopInThisThread)
25     {
26         LOG_FATAL << "Another EventLoop " << t_loopInThisThread
27                   << " exists in this thread " << threadId_;
28     }
29     else
30     {
31         t_loopInThisThread = this;
32     }
33 }
34
35 EventLoop::~EventLoop()
36 {
37     assert(!looping_);
38     t_loopInThisThread = NULL;
39 }

```

reactor/s00/EventLoop.cc

既然每个线程至多有一个 EventLoop 对象，那么我们让 EventLoop 的 static 成员函数 `getEventLoopOfCurrentThread()` 返回这个对象。返回值可能为 NULL，如果当前线程不是 IO 线程的话。（这个函数是 muduo 后来新加的，因此前面头文件中没有它的原型。）

```
EventLoop* EventLoop::getEventLoopOfCurrentThread()
{
    return t_loopInThisThread;
}
```

muduo/net/EventLoop.cc

muduo/net/EventLoop.cc

muduo 的接口设计会明确哪些成员函数是线程安全的，可以跨线程调用；哪些成员函数只能在某个特定线程调用（主要是 IO 线程）。为了能在运行时检查这些 pre-condition，EventLoop 提供了 `isInLoopThread()` 和 `assertInLoopThread()` 等函数（EventLoop.h L25~L33），其中用到的 `EventLoop::abortNotInLoopThread()` 函数的定义从略。

事件循环必须在 IO 线程执行，因此 `EventLoop::loop()` 会检查这一 pre-condition（L44）。本节的 `loop()` 什么事都不做，等 5 秒就退出。

```
41 void EventLoop::loop()
42 {
43     assert(!looping_);
44     assertInLoopThread();
45     looping_ = true;
46
47     ::poll(NULL, 0, 5*1000);
48
49     LOG_TRACE << "EventLoop " << this << " stop looping";
50     looping_ = false;
51 }
```

reactor/s00/EventLoop.cc

reactor/s00/EventLoop.cc

为了验证现有的功能，我编写了 `s00/test1.cc` 和 `s00/test2.cc`。其中 `test1.cc` 会在主线程和子线程分别创建一个 EventLoop，程序正常运行退出。

```
5 void threadFunc()
6 {
7     printf("threadFunc(): pid = %d, tid = %d\n",
8           getpid(), muduo::CurrentThread::tid());
9
10    muduo::EventLoop loop;
11    loop.loop();
12 }
13
```

reactor/s00/test1.cc


```

14 int main()
15 {
16     printf("main(): pid = %d, tid = %d\n",
17           getpid(), muduo::CurrentThread::tid());
18
19     muduo::EventLoop loop;
20
21     muduo::Thread thread(threadFunc);
22     thread.start();
23
24     loop.loop();
25     pthread_exit(NULL);
26 }

```

 reactor/s00/test1.cc

test2.cc 是个负面测试，它在线程创建了 EventLoop 对象，却试图在另一个线程调用其 EventLoop::loop()，程序会因断言失效而异常终止。练习：写一个负面测试，在线程创建两个 EventLoop 对象，验证程序会异常终止。

```

4 muduo::EventLoop* g_loop;
5
6 void threadFunc()
7 {
8     g_loop->loop();
9 }
10
11 int main()
12 {
13     muduo::EventLoop loop;
14     g_loop = &loop;
15     muduo::Thread t(threadFunc);
16     t.start();
17     t.join();
18 }

```

 reactor/s00/test2.cc

 reactor/s00/test2.cc

8.1 Reactor 的关键结构

本节讲 Reactor 最核心的事件分发机制，即将 IO multiplexing 拿到的 IO 事件分发给各个文件描述符（fd）的事件处理函数。

8.1.1 Channel class

Channel class 的功能有一点类似 Java NIO 的 SelectableChannel 和 SelectionKey 的组合。每个 Channel 对象自始至终只属于一个 EventLoop，因此每个 Channel 对

象都只属于某一个 IO 线程。每个 Channel 对象自始至终只负责一个文件描述符 (fd) 的 IO 事件分发, 但它并不拥有这个 fd, 也不会在析构的时候关闭这个 fd。Channel 会把不同的 IO 事件分发为不同的回调, 例如 ReadCallback、WriteCallback 等, 而且“回调”用 boost::function 表示, 用户无须继承 Channel, Channel 不是基类。muduo 用户一般不直接使用 Channel, 而会使用更上层的封装, 如 TcpConnection。Channel 的生命期由其 owner class 负责管理, 它一般是其他 class 的直接或间接成员。以下是 Channel 的 public interface:

```

17 class EventLoop;
18
19 ///
20 /// A selectable I/O channel.
21 ///
22 /// This class doesn't own the file descriptor.
23 /// The file descriptor could be a socket,
24 /// an eventfd, a timerfd, or a signalfd
25 class Channel : boost::noncopyable
26 {
27 public:
28     typedef boost::function<void()> EventCallback;
29
30     Channel(EventLoop* loop, int fd);
31
32     void handleEvent();
33     void setReadCallback(const EventCallback& cb)
34     { readCallback_ = cb; }
35     void setWriteCallback(const EventCallback& cb)
36     { writeCallback_ = cb; }
37     void setErrorCallback(const EventCallback& cb)
38     { errorCallback_ = cb; }
39
40     int fd() const { return fd_; }
41     int events() const { return events_; }
42     void set_revents(int revt) { revents_ = revt; }
43     bool isNoneEvent() const { return events_ == kNoneEvent; }
44
45     void enableReading() { events_ |= kReadEvent; update(); }
46     // void enableWriting() { events_ |= kWriteEvent; update(); }
47     // void disableWriting() { events_ &= ~kWriteEvent; update(); }
48     // void disableAll() { events_ = kNoneEvent; update(); }
49
50     // for Poller
51     int index() { return index_; }
52     void set_index(int idx) { index_ = idx; }
53
54     EventLoop* ownerLoop() { return loop_; }

```

有些成员函数是内部使用的，用户一般只用 `set*Callback()` 和 `enableReading()` 这几个函数。其中有些函数目前还用不到，因此暂时注释起来。`Channel` 的成员函数都只能在 IO 线程调用，因此更新数据成员都不必加锁。

以下是 `Channel` class 的数据成员。其中 `events_` 是它关心的 IO 事件，由用户设置；`revents_` 是目前活动的事件，由 `EventLoop/Poller` 设置；这两个字段都是 bit pattern，它们的名字来自 `poll(2)` 的 `struct pollfd`。

```

56 private:
57     void update();
58
59     static const int kNoneEvent;
60     static const int kReadEvent;
61     static const int kWriteEvent;
62
63     EventLoop* loop_;
64     const int fd_;
65     int events_;
66     int revents_;
67     int index_; // used by Poller.
68
69     EventCallback readCallback_;
70     EventCallback writeCallback_;
71     EventCallback errorCallback_;
72 };

```

reactor/s01/Channel.h

注意到 `Channel.h` 没有包含任何 POSIX 头文件，因此 `kReadEvent` 和 `kWriteEvent` 等常量的定义要放到 `Channel.cc` 中。

```

18 const int Channel::kNoneEvent = 0;
19 const int Channel::kReadEvent = POLLIN | POLLPRI;
20 const int Channel::kWriteEvent = POLLOUT;
21
22 Channel::Channel(EventLoop* loop, int fdArg)
23     : loop_(loop),
24       fd_(fdArg),
25       events_(0),
26       revents_(0),
27       index_(-1)
28 {
29 }
30
31 void Channel::update()
32 {
33     loop_>updateChannel(this);
34 }

```

reactor/s01/Channel.cc

`Channel::update()` 会调用 `EventLoop::updateChannel()`，后者会转而调用 `Poller::updateChannel()`。由于 `Channel.h` 没有包含 `EventLoop.h`，因此 `Channel::update()` 必须定义在 `Channel.cc` 中。

`Channel::handleEvent()` 是 `Channel` 的核心，它由 `EventLoop::loop()` 调用，它的功能是根据 `revents_` 的值分别调用不同的用户回调。这个函数以后还会扩充。

```

36 void Channel::handleEvent()
37 {
38     if (revents_ & POLLNVAL) {
39         LOG_WARN << "Channel::handle_event() POLLNVAL";
40     }
41
42     if (revents_ & (POLLERR | POLLNVAL)) {
43         if (errorCallback_) errorCallback_();
44     }
45     if (revents_ & (POLLIN | POLLPRI | POLLRDHUP)) {
46         if (readCallback_) readCallback_();
47     }
48     if (revents_ & POLLOUT) {
49         if (writeCallback_) writeCallback_();
50     }
51 }

```

reactor/s01/Channel.cc

8.1.2 Poller class

`Poller class` 是 `IO multiplexing` 的封装。它现在是个具体类，而在 `muduo` 中是个抽象基类，因为 `muduo` 同时支持 `poll(2)` 和 `epoll(4)` 两种 `IO multiplexing` 机制。`Poller` 是 `EventLoop` 的间接成员，只供其 `owner EventLoop` 在 `IO` 线程调用，因此无须加锁。其生命期与 `EventLoop` 相等。`Poller` 并不拥有 `Channel`，`Channel` 在析构之前必须自己 `unregister` (`EventLoop::removeChannel()`)，避免空悬指针。

```

17 struct pollfd;
18
19 namespace muduo
20 {
21
22     class Channel;
23
24     ///
25     /// IO Multiplexing with poll(2).
26     ///
27     /// This class doesn't own the Channel objects.

```

reactor/s01/Poller.h

```

28 class Poller : boost::noncopyable
29 {
30 public:
31     typedef std::vector<Channel*> ChannelList;
32
33     Poller(EventLoop* loop);
34     ~Poller();
35
36     /// Polls the I/O events.
37     /// Must be called in the loop thread.
38     Timestamp poll(int timeoutMs, ChannelList* activeChannels);
39
40     /// Changes the interested I/O events.
41     /// Must be called in the loop thread.
42     void updateChannel(Channel* channel);
43
44     void assertInLoopThread() { ownerLoop_>assertInLoopThread(); }

```

注意 Poller.h 并没有 include <poll.h>, 而是自己前向声明了 struct pollfd, 这不妨碍我们定义 vector<struct pollfd> 成员。

Poller 供 EventLoop 调用的函数目前有两个, poll() 和 updateChannel(), Poller 暂时没有定义 removeChannel() 成员函数, 因为前几节还用不到它。

以下是 Poller class 的数据成员。其中 ChannelMap 是从 fd 到 Channel* 的映射。Poller::poll() 不会在每次调用 poll(2) 之前临时构造 pollfd 数组, 而是把它缓存起来 (pollfds_)。

```

46 private:
47     void fillActiveChannels(int numEvents,
48                             ChannelList* activeChannels) const;
49
50     typedef std::vector<struct pollfd> PollFdList;
51     typedef std::map<int, Channel*> ChannelMap;
52
53     EventLoop* ownerLoop_;
54     PollFdList pollfds_;
55     ChannelMap channels_;
56 };
57
58 }

```

reactor/s01/Poller.h

Poller 的构造函数和析构函数都很简单, 因其成员都是标准库容器。

```

18 Poller::Poller(EventLoop* loop)
19     : ownerLoop_(loop)
20 {
21 }
22

```

reactor/s01/Poller.cc

```

23 Poller::~Poller()
24 {
25 }

```

reactor/s01/Poller.cc

`Poller::poll()` 是 `Poller` 的核心功能，它调用 `poll(2)` 获得当前活动的 IO 事件，然后填充调用方传入的 `activeChannels`，并返回 `poll(2)` return 的时刻。这里我们直接把 `vector<struct pollfd> pollfds_` 作为参数传给 `poll(2)`，因为 C++ 标准保证 `std::vector` 的元素排列跟数组一样。`L30` 中的 `&*pollfds_.begin()` 是获得元素的首地址，这个表达式的类型为 `pollfds_*`，符合 `poll(2)` 的要求。（在 C++11 中可写为 `pollfds_.data()`，g++4.4 的 STL 也支持这种写法。）

```

27 Timestamp Poller::poll(int timeoutMs, ChannelList* activeChannels)
28 {
29     // XXX pollfds_ shouldn't change
30     int numEvents = ::poll(&*pollfds_.begin(), pollfds_.size(), timeoutMs);
31     Timestamp now(Timestamp::now());
32     if (numEvents > 0) {
33         LOG_TRACE << numEvents << " events happended";
34         fillActiveChannels(numEvents, activeChannels);
35     } else if (numEvents == 0) {
36         LOG_TRACE << " nothing happended";
37     } else {
38         LOG_SYSERR << "Poller::poll()";
39     }
40     return now;
41 }

```

reactor/s01/Poller.cc

`fillActiveChannels()` 遍历 `pollfds_`，找出有活动事件的 `fd`，把它对应的 `Channel` 填入 `activeChannels`。这个函数的复杂度是 $O(N)$ ，其中 N 是 `pollfds_` 的长度，即文件描述符数目。为了提前结束循环，每找到一个活动 `fd` 就递减 `numEvents`，这样当 `numEvents` 减为 0 时表示活动 `fd` 都找完了，不必做无用功。当前活动事件 `revents` 会保存在 `Channel` 中，供 `Channel::handleEvent()` 使用（`L56`）。

注意这里我们不能一边遍历 `pollfds_`，一边调用 `Channel::handleEvent()`，因为后者会添加或删除 `Channel`，从而造成 `pollfds_` 在遍历期间改变大小，这是非常危险的。另外一个原因是简化 `Poller` 的职责，它只负责 IO multiplexing，不负责事件分发（dispatching）。这样将来可以方便地替换为其他更高效的 IO multiplexing 机制，如 `epoll(4)`。

```

43 void Poller::fillActiveChannels(int numEvents,
44                               ChannelList* activeChannels) const
45 {
46     for (PollFdList::const_iterator pfd = pollfds_.begin();
47         pfd != pollfds_.end() && numEvents > 0; ++pfd)

```

```

48     {
49         if (pfd->revents > 0)
50         {
51             --numEvents;
52             ChannelMap::const_iterator ch = channels_.find(pfd->fd);
53             assert(ch != channels_.end());
54             Channel* channel = ch->second;
55             assert(channel->fd() == pfd->fd);
56             channel->set_revents(pfd->revents);
57             // pfd->revents = 0;
58             activeChannels->push_back(channel);
59         }
60     }
61 }

```

`Poller::updateChannel()` 的主要功能是负责维护和更新 `pollfds_` 数组。添加新 `Channel` 的复杂度是 $O(\log N)$, 更新已有的 `Channel` 的复杂度是 $O(1)$, 因为 `Channel` 记住了自己在 `pollfds_` 数组中的下标, 因此可以快速定位。`removeChannel()` 的复杂度也将会是 $O(\log N)$ 。这里用了大量的 `assert` 来检查 invariant。

```

63 void Poller::updateChannel(Channel* channel)
64 {
65     assertInLoopThread();
66     LOG_TRACE << "fd = " << channel->fd() << " events = " << channel->events();
67     if (channel->index() < 0) {
68         // a new one, add to pollfds_
69         assert(channels_.find(channel->fd()) == channels_.end());
70         struct pollfd pfd;
71         pfd.fd = channel->fd();
72         pfd.events = static_cast<short>(channel->events());
73         pfd.revents = 0;
74         pollfds_.push_back(pfd);
75         int idx = static_cast<int>(pollfds_.size())-1;
76         channel->set_index(idx);
77         channels_[pfd.fd] = channel;
78     } else {
79         // update existing one
80         assert(channels_.find(channel->fd()) != channels_.end());
81         assert(channels_[channel->fd()] == channel);
82         int idx = channel->index();
83         assert(0 <= idx && idx < static_cast<int>(pollfds_.size()));
84         struct pollfd& pfd = pollfds_[idx];
85         assert(pfd.fd == channel->fd() || pfd.fd == -1);
86         pfd.events = static_cast<short>(channel->events());
87         pfd.revents = 0;
88         if (channel->isNoneEvent()) {
89             // ignore this pollfd
90             pfd.fd = -1;
91         }
92     }
93 }

```

reactor/s01/Poller.cc

另外，如果某个 Channel 暂时不关心任何事件，就把 `pollfd.fd` 设为 `-1`，让 `poll(2)` 忽略此项（L90）¹。这里不能改为把 `pollfd.events` 设为 `0`，这样无法屏蔽 `POLLERR` 事件。改进的做法（p. 312）是把 `pollfd.fd` 设为 `channel->fd()` 的相反数减一，这样可以进一步检查 `invariant`。（思考：为什么要减一？）

8.1.3 EventLoop 的改动

`EventLoop` class 新增了 `quit()` 成员函数，还加了几个数据成员，并在构造函数里初始化它们。注意 `EventLoop` 通过 `scoped_ptr` 来间接持有 `Poller`，因此 `EventLoop.h` 不必包含 `Poller.h`，只需前向声明 `Poller` class。为此，`EventLoop` 的析构函数必须在 `EventLoop.cc` 中显式定义。

```

56     void abortNotInLoopThread();
57
58 + typedef std::vector<Channel*> ChannelList;
59 +
60     bool looping_; /* atomic */
61 + bool quit_; /* atomic */
62     const pid_t threadId_;
63 + boost::scoped_ptr<Poller> poller_;
64 + ChannelList activeChannels_;
65 };
66

```

reactor/s01/EventLoop.h

reactor/s01/EventLoop.h

`EventLoop::loop()` 有了真正的工作内容，它调用 `Poller::poll()` 获得当前活动事件的 Channel 列表，然后依次调用每个 Channel 的 `handleEvent()` 函数。

```

46 void EventLoop::loop()
47 {
48     assert(!looping_);
49     assertInLoopThread();
50     looping_ = true;
51 + quit_ = false;
52
53 + while (!quit_)
54 + {
55 +     activeChannels_.clear();
56 +     poller_->poll(kPollTimeMs, &activeChannels_);
57 +     for (ChannelList::iterator it = activeChannels_.begin();
58 +         it != activeChannels_.end(); ++it)
59 +     {
60 +         (*it)->handleEvent();
61 +     }
62 + }

```

reactor/s01/EventLoop.cc

¹ <http://pubs.opengroup.org/onlinepubs/007908799/xsh/poll.html>


```

63
64     LOG_TRACE << "EventLoop " << this << " stop looping";
65     looping_ = false;
66 }

```

reactor/s01/EventLoop.cc

以上几个 class 尽管简陋，却构成了 Reactor 模式的核心内容。时序图见图 8-1。

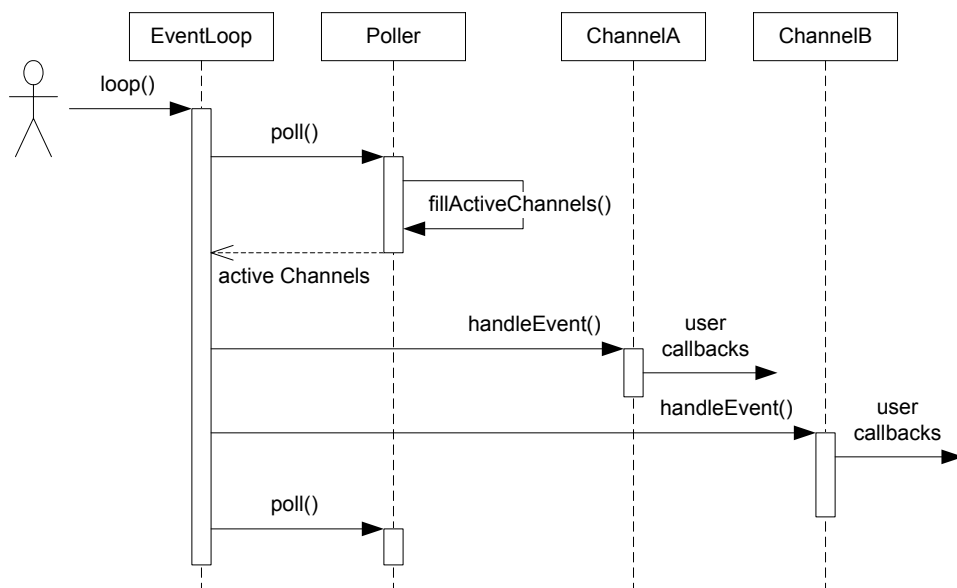


图 8-1

我们现在可以终止事件循环，只要将 `quit_` 设为 `true` 即可，但是 `quit()` 不是立刻发生的，它会在 `EventLoop::loop()` 下一次检查 `while (!quit_)` 的时候起效 (L53)。如果在非当前 IO 线程调用 `quit()`，延迟可以长达数秒，将来我们可以唤醒 `EventLoop` 以缩小延时。但是 `quit()` 不是中断或 `signal`，而是设标志，如果 `EventLoop::loop()` 正阻塞在某个调用中，`quit()` 不会立刻生效。

```

68 void EventLoop::quit()
69 {
70     quit_ = true;
71     // wakeup();
72 }

```

reactor/s01/EventLoop.cc

reactor/s01/EventLoop.cc

`EventLoop::updateChannel()` 在检查断言之后调用 `Poller::updateChannel()`，`EventLoop` 不关心 `Poller` 是如何管理 `Channel` 列表的。

```

74 void EventLoop::updateChannel(Channel* channel)
75 {
76     assert(channel->ownerLoop() == this);
77     assertInLoopThread();
78     poller_->updateChannel(channel);
79 }

```

reactor/s01/EventLoop.cc

reactor/s01/EventLoop.cc

有了以上的 EventLoop、Poller、Channel，我们写个小程序简单地测试一下功能。s01/test3.cc 用 timerfd 实现了一个单次触发的定时器，为 §8.2 的内容打下基础。这个程序利用 Channel 将 timerfd 的 readable 事件转发给 timeout() 函数。

```

5 #include <sys/timerfd.h>
6
7 muduo::EventLoop* g_loop;
8
9 void timeout()
10 {
11     printf("Timeout!\n");
12     g_loop->quit();
13 }
14
15 int main()
16 {
17     muduo::EventLoop loop;
18     g_loop = &loop;
19
20     int timerfd = ::timerfd_create(CLOCK_MONOTONIC, TFD_NONBLOCK | TFD_CLOEXEC);
21     muduo::Channel channel(&loop, timerfd);
22     channel.setReadCallback(timeout);
23     channel.enableReading();
24
25     struct itimerspec howlong;
26     bzero(&howlong, sizeof howlong);
27     howlong.it_value.tv_sec = 5;
28     ::timerfd_settime(timerfd, 0, &howlong, NULL);
29
30     loop.loop();
31
32     ::close(timerfd);
33 }

```

reactor/s01/test3.cc

reactor/s01/test3.cc

由于 poll(2) 是 level trigger，在 timeout() 中应该 read() timerfd，否则下次会立刻触发。现阶段采用 level trigger 的好处之一是可以通过 strace 命令直观地看到每次 poll(2) 的参数列表，容易检查程序的行为。

8.2 TimerQueue 定时器

有了前面的 Reactor 基础，我们可以给 EventLoop 加上定时器功能。传统的 Reactor 通过控制 select(2) 和 poll(2) 的等待时间来实现定时，而现在在 Linux 中有了 timerfd，我们可以用和处理 IO 事件相同的方式来处理定时，代码的一致性更好。muduo 中的 backport.diff 展示了传统方案。

8.2.1 TimerQueue class

muduo 的定时器功能由三个 class 实现，TimerId、Timer、TimerQueue，用户只能看到第一个 class，另外两个都是内部实现细节。TimerId 和 Timer 的实现很简单，这里就不展示源码了。

TimerQueue 的接口很简单，只有两个函数 addTimer() 和 cancel()。本节我们只实现 addTimer()，cancel() 的实现见 p. 328。addTimer() 是供 EventLoop 使用的，EventLoop 会把它封装为更好用的 runAt()、runAfter()、runEvery() 等函数。

```
28 /// reactor/s02/TimerQueue.h
29 /// A best efforts timer queue.
30 /// No guarantee that the callback will be on time.
31 ///
32 class TimerQueue : boost::noncopyable
33 {
34 public:
35     TimerQueue(EventLoop* loop);
36     ~TimerQueue();
37
38     ///
39     /// Schedules the callback to be run at given time,
40     /// repeats if @c interval > 0.0.
41     ///
42     /// Must be thread safe. Usually be called from other threads.
43     TimerId addTimer(const TimerCallback& cb,
44                     Timestamp when,
45                     double interval);
46
47     // void cancel(TimerId timerId);
```

值得一提的是 TimerQueue 的数据结构的选择，TimerQueue 需要高效地组织目前尚未到期的 Timer，能快速地根据当前时间找到已经到期的 Timer，也要能高效地添加和删除 Timer。最简单的 TimerQueue 以按到期时间排好序的线性表为数据结构，muduo 最早也是用这种结构。这种结构的常用操作都是线性查找，复杂度是 $O(N)$ 。

另一种常用做法是二叉堆组织优先队列（libev 用的是更高效的 4-heap），这种做法的复杂度降为 $O(\log N)$ ，但是 C++ 标准库的 `make_heap()` 等函数不能高效地删除 heap 中间的某个元素，需要我们自己实现（令 Timer 记住自己在 heap 中的位置）。

还有一种做法是使用二叉搜索树（例如 `std::set/std::map`），把 Timer 按到期时间先后排好序。操作的复杂度仍然是 $O(\log N)$ ，不过 memory locality 比 heap 要差一些，实际速度可能略慢。但是我们不能直接用 `map<Timestamp, Timer*>`，因为这样无法处理两个 Timer 到期时间相同的情况。有两个解决方案，一是用 `multimap` 或 `multiset`，二是设法区分 key。muduo 现在采用的是第二种做法，这样可以避免使用不常见的 `multimap` class。具体来说，以 `pair<Timestamp, Timer*>` 为 key，这样即便两个 Timer 的到期时间相同，它们的地址也必定不同。

以下是 TimerQueue 的数据成员，这个结构利用了现成的容器库，实现简单，容易验证其正确性，并且性能也不错。TimerList 是 set 而非 map，因为只有 key 没有 value。TimerQueue 使用了一个 Channel 来观察 `timerfd_` 上的 readable 事件。注意 TimerQueue 的成员函数只能在其所属的 IO 线程调用，因此不必加锁。

```

51 // FIXME: use unique_ptr<Timer> instead of raw pointers.
52 typedef std::pair<Timestamp, Timer*> Entry;
53 typedef std::set<Entry> TimerList;
54
55 // called when timerfd alarms
56 void handleRead();
57 // move out all expired timers
58 std::vector<Entry> getExpired(Timestamp now);
59 void reset(const std::vector<Entry>& expired, Timestamp now);
60
61 bool insert(Timer* timer);
62
63 EventLoop* loop_;
64 const int timerfd_;
65 Channel timerfdChannel_;
66 // Timer list sorted by expiration
67 TimerList timers_;
68 };

```

reactor/s02/TimerQueue.h

TimerQueue 的实现目前有一个不理想的地方，Timer 是用裸指针管理的，需要手动 delete。这里用 `shared_ptr` 似乎有点小题大做了。在 C++11 中，或许可以改进为 `unique_ptr`，避免手动管理资源。

来看关键的 `getExpired()` 函数的实现，这个函数会从 `timers_` 中移除已到期的 Timer，并通过 vector 返回它们。编译器会实施 RVO 优化，不必太担心性能，必要时可以像 `EventLoop::activeChannels_` 那样复用 vector。注意其中哨兵值（sentry）

的选取, `sentry` 让 `set::lower_bound()` 返回的是第一个未到期的 `Timer` 的迭代器, 因此 L145 的断言中是 `<` 而非 `≤`。

```

140 std::vector<TimerQueue::Entry> TimerQueue::getExpired(Timestamp now)
141 {
142     std::vector<Entry> expired;
143     Entry sentry = std::make_pair(now, reinterpret_cast<Timer*>(UINTPTR_MAX));
144     TimerList::iterator it = timers_.lower_bound(sentry);
145     assert(it == timers_.end() || now < it->first);
146     std::copy(timers_.begin(), it, back_inserter(expired));
147     timers_.erase(timers_.begin(), it);
148
149     return expired;
150 }

```

reactor/s02/TimerQueue.cc

图 8-2 是 `TimerQueue` 回调用户代码 `onTimer()` 的时序图。

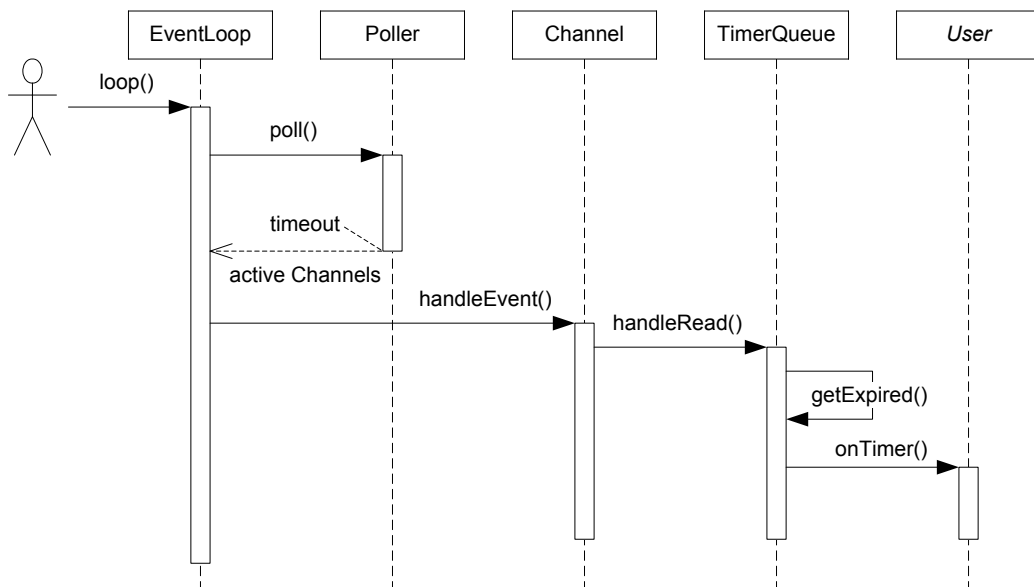


图 8-2

8.2.2 EventLoop 的改动

`EventLoop` 新增了几个方便用户使用的定时器接口, 这几个函数都转而调用 `TimerQueue::addTimer()`。注意这几个 `EventLoop` 成员函数应该允许跨线程使用, 比方说我想在某个 IO 线程中执行超时回调。这就带来线程安全性方面的问题, `muduo` 的解决办法不是加锁, 而是把对 `TimerQueue` 的操作转移到 IO 线程来进行, 这会用到 §8.3 介绍的 `EventLoop::runInLoop()` 函数。

```

reactor/s02/EventLoop.cc
76 TimerId EventLoop::runAt(const Timestamp& time, const TimerCallback& cb)
77 {
78     return timerQueue->addTimer(cb, time, 0.0);
79 }
80
81 TimerId EventLoop::runAfter(double delay, const TimerCallback& cb)
82 {
83     Timestamp time(addTime(Timestamp::now(), delay));
84     return runAt(time, cb);
85 }
86
87 TimerId EventLoop::runEvery(double interval, const TimerCallback& cb)
88 {
89     Timestamp time(addTime(Timestamp::now(), interval));
90     return timerQueue->addTimer(cb, time, interval);
91 }

```

测试代码见 s02/test4.cc, 这与 muduo 正式的用法完全一样。

8.3 EventLoop::runInLoop() 函数

EventLoop 有一个非常有用的功能: 在它的 IO 线程内执行某个用户任务回调, 即 EventLoop::runInLoop(const Functor& cb), 其中 Functor 是 boost::function<void()>。如果用户在当前 IO 线程调用这个函数, 回调会同步进行; 如果用户在其他线程调用 runInLoop(), cb 会被加入队列, IO 线程会被唤醒来调用这个 Functor。

```

reactor/s03/EventLoop.cc
void EventLoop::runInLoop(const Functor& cb)
{
    if (isInLoopThread()) {
        cb();
    } else {
        queueInLoop(cb);
    }
}

```

有了这个功能, 我们就能轻易地在线程间调配任务, 比方说把 TimerQueue 的成员函数调用移到其 IO 线程, 这样可以在不用锁的情况下保证线程安全性。

由于 IO 线程平时阻塞在事件循环 EventLoop::loop() 的 poll(2) 调用中, 为了让 IO 线程能立刻执行用户回调, 我们需要设法唤醒它。传统的办法是用 pipe(2), IO 线程始终监视此管道的 readable 事件, 在需要唤醒的时候, 其他线程往管道里

写一个字节，这样 IO 线程就从 IO multiplexing 阻塞调用中返回。（原理类似 HTTP long polling。）现在 Linux 有了 `eventfd(2)`，可以更高效地唤醒，因为它不必管理缓冲区。以下是 `EventLoop` 新增的成员。

```

96     private:
97
98     void abortNotInLoopThread();
99 + void handleRead(); // waked up
100 + void doPendingFuncutors();
101
102     typedef std::vector<Channel*> ChannelList;
103
104     bool looping_; /* atomic */
105     bool quit_; /* atomic */
106 + bool callingPendingFuncutors_; /* atomic */
107     const pid_t threadId_;
108     Timestamp pollReturnTime_;
109     boost::scoped_ptr<Poller> poller_;
110     boost::scoped_ptr<TimerQueue> timerQueue_;
111 + int wakeupFd_;
112 + // unlike in TimerQueue, which is an internal class,
113 + // we don't expose Channel to client.
114 + boost::scoped_ptr<Channel> wakeupChannel_;
115     ChannelList activeChannels_;
116 + MutexLock mutex_;
117 + std::vector<Functor> pendingFuncutors_; // @BuardedBy mutex_
118 };

```

reactor/s03/EventLoop.h

`wakeupChannel_` 用于处理 `wakeupFd_` 上的 readable 事件，将事件分发至 `handleRead()` 函数。其中只有 `pendingFuncutors_` 暴露给了其他线程，因此用 `mutex` 保护。

`queueInLoop()` 的实现很简单，将 `cb` 放入队列，并在必要时唤醒 IO 线程。

```

114 void EventLoop::queueInLoop(const Functor& cb)
115 {
116     {
117         MutexLockGuard lock(mutex_);
118         pendingFuncutors_.push_back(cb);
119     }
120
121     if (!isInLoopThread() || callingPendingFuncutors_)
122     {
123         wakeup();
124     }
125 }

```

reactor/s03/EventLoop.cc

“必要时”有两种情况，如果调用 `queueInLoop()` 的线程不是 IO 线程，那么唤醒是必需的；如果在 IO 线程调用 `queueInLoop()`，而此时正在调用 `pending functor`，那

么也必须唤醒。换句话说，只有在 IO 线程的事件回调中调用 `queueInLoop()` 才无须 `wakeup()`。看了下面 `doPendingFuncutors()` 的调用时间点，想必读者就能明白为什么。

p. 287 的事件循环 `EventLoop::loop()` 中需要增加一行代码，执行 `pending-Funcutors_` 中的任务回调。

```

77     while (!quit_)
78     {
79         activeChannels_.clear();
80         pollReturnTime_ = poller_>poll(kPollTimeMs, &activeChannels_);
81         for (ChannelList::iterator it = activeChannels_.begin();
82             it != activeChannels_.end(); ++it)
83         {
84             (*it)->handleEvent();
85         }
86 +     doPendingFuncutors();
87     }

```

reactor/s03/EventLoop.cc

`EventLoop::doPendingFuncutors()` 不是简单地在临界区内依次调用 `Functor`，而是把回调列表 `swap()` 到局部变量 `funcutors` 中，这样一方面减小了临界区的长度（意味着不会阻塞其他线程调用 `queueInLoop()`），另一方面也避免了死锁（因为 `Functor` 可能再调用 `queueInLoop()`）。

```

178 void EventLoop::doPendingFuncutors()
179 {
180     std::vector<Functor> funcutors;
181     callingPendingFuncutors_ = true;
182
183     {
184         MutexLockGuard lock(mutex_);
185         funcutors.swap(pendingFuncutors_);
186     }
187
188     for (size_t i = 0; i < funcutors.size(); ++i)
189     {
190         funcutors[i]();
191     }
192     callingPendingFuncutors_ = false;
193 }

```

reactor/s03/EventLoop.cc

由于 `doPendingFuncutors()` 调用的 `Functor` 可能再调用 `queueInLoop(cb)`，这时 `queueInLoop()` 就必须 `wakeup()`，否则这些新加的 `cb` 就不能被及时调用了。`muduo` 这里没有反复执行 `doPendingFuncutors()` 直到 `pendingFuncutors_` 为空，这是有意的，否则 IO 线程有可能陷入死循环，无法处理 IO 事件。

剩下的事情就简单了，在 `EventLoop::quit()` 中增加几行代码，在必要时唤醒 IO 线程，让它及时终止循环。思考：为什么在 IO 线程调用 `quit()` 就不必 `wakeup()`？

```

93 void EventLoop::quit()
94 {
95     quit_ = true;
96 +   if (!isInLoopThread())
97 +   {
98 +       wakeup();
99 +   }
100 }

```

reactor/s03/EventLoop.cc

reactor/s03/EventLoop.cc

`EventLoop::wakeup()` 和 `EventLoop::handleRead()` 分别对 `wakeupFd_` 写入数据和读出数据，代码从略。注意 `muduo` 不是在 `EventLoop::handleRead()` 中执行 `doPendingFuncutors()`，理由见 <http://blog.csdn.net/solstice/article/details/6171831#comments>。

`s03/test5.cc` 是单线程程序，测试了 `runInLoop()` 和 `queueInLoop()` 等新函数。

8.3.1 提高 TimerQueue 的线程安全性

前面提到 `TimerQueue::addTimer()` 只能在 IO 线程调用，因此 `EventLoop::runAfter()` 系列函数不是线程安全的。下面这段代码在 §8.2 中会 crash，因为它在非 IO 线程调用了 `EventLoop::runAfter()`。

```

muduo::EventLoop* g_loop;

void print() { } // 空函数

void threadFunc()
{
    g_loop->runAfter(1.0, print);
}

int main()
{
    muduo::EventLoop loop;
    g_loop = &loop;
    muduo::Thread t(threadFunc);
    t.start();
    loop.loop();
}

```

运行结果：

```

20120901 01:36:26.905473Z 17897 FATAL EventLoop::abortNotInLoopThread -
    EventLoop 0x7fff892d1070 was created in threadId_ = 17896,
    current thread id = 17897 - EventLoop.cc:102
Aborted (core dumped)

```

借助 EventLoop::runInLoop(), 我们可以很容易地将 TimerQueue::addTimer() 做成线程安全的, 而且无须用锁。办法是让 addTimer() 调用 runInLoop(), 把实际工作转移到 IO 线程来做。先新增一个 addTimerInLoop() 成员函数:

```

52     typedef std::pair<Timestamp, Timer*> Entry;
53     typedef std::set<Entry> TimerList;
54
55 + void addTimerInLoop(Timer* timer);
56 // called when timerfd alarms
57 void handleRead();

```

reactor/s03/TimerQueue.h

然后把 addTimer() 拆成两部分, 拆分后的 addTimer() 只负责转发, addTimerInLoop() 完成修改定时器列表的工作。

```

107 TimerId TimerQueue::addTimer(const TimerCallback& cb,
108                             Timestamp when,
109                             double interval)
110 {
111     Timer* timer = new Timer(cb, when, interval);
112 + loop_->runInLoop(
113 +     boost::bind(&TimerQueue::addTimerInLoop, this, timer));
114 + return TimerId(timer);
115 +}
116
117 +void TimerQueue::addTimerInLoop(Timer* timer)
118 +{
119     loop_->assertInLoopThread();
120     bool earliestChanged = insert(timer);
121
122     if (earliestChanged)
123     {
124         resetTimerfd(timerfd_, timer->expiration());
125     }
126 - return TimerId(timer);
127 }

```

reactor/s03/TimerQueue.cc

这样无论在哪个线程调用 addTimer() 都是安全的了, 上一页的代码也能正常运行。

8.3.2 EventLoopThread class

IO 线程不一定是主线程, 我们可以在任何一个线程创建并运行 EventLoop。一个程序也可以有不止一个 IO 线程, 我们可以按优先级将不同的 socket 分给不同的 IO 线程, 避免优先级反转。为了方便将来使用, 我们定义 EventLoopThread class, 这正是 one loop per thread 的本意。

EventLoopThread 会启动自己的线程，并在其中运行 EventLoop::loop()。其中关键的 startLoop() 函数定义如下，这个函数会返回新线程中 EventLoop 对象的地址，因此用条件变量来等待线程的创建与运行。

```
reactor/s03/EventLoopThread.cc
32 EventLoop* EventLoopThread::startLoop()
33 {
34     assert(!thread_.started());
35     thread_.start();
36
37     {
38         MutexLockGuard lock(mutex_);
39         while (loop_ == NULL)
40         {
41             cond_.wait();
42         }
43     }
44     return loop_;
45 }
46 }
```

线程主函数在 stack 上定义 EventLoop 对象，然后将其地址赋值给 loop_ 成员变量，最后 notify() 条件变量，唤醒 startLoop()。

```
reactor/s03/EventLoopThread.cc
48 void EventLoopThread::threadFunc()
49 {
50     EventLoop loop;
51
52     {
53         MutexLockGuard lock(mutex_);
54         loop_ = &loop;
55         cond_.notify();
56     }
57
58     loop.loop();
59     //assert(exiting_);
60 }
```

由于 EventLoop 的生命期与线程主函数的作用域相同，因此在 threadFunc() 退出之后这个指针就失效了。好在服务程序一般不要求能安全地退出 (§9.2.2)，这应该不是什么大问题。

s03/test6.cc 测试了 EventLoopThread 的功能，也测试了跨线程调用 EventLoop::runInLoop() 和 EventLoop::runAfter()，代码从略。

8.4 实现 TCP 网络库

到目前为止，Reactor 事件处理框架已初具规模，从本节开始我们用它逐步实现一个非阻塞 TCP 网络编程库。从 `poll(2)` 返回到再次调用 `poll(2)` 阻塞称为一次事件循环。图 8-3 值得印在脑中，它有助于理解一次循环中各种回调发生的顺序。

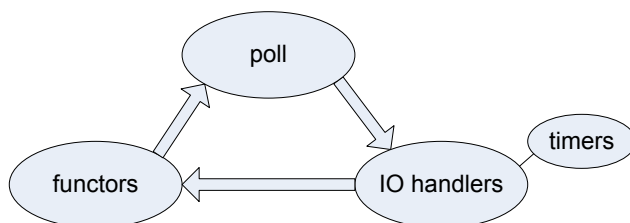


图 8-3

传统的 Reactor 实现一般会把 `timers` 做成循环中单独的一步，而 `muduo` 把它和 `IO handlers` 等同视之，这是使用 `timerfd` 的附带效应。将来有必要时也可以在调用 `IO handlers` 之前或之后处理 `timers`。

后面几节的内容安排如下：

§8.4 介绍 `Acceptor class`，用于 `accept(2)` 新连接。

§8.5 介绍 `TcpServer`，处理新建 `TcpConnection`。

§8.6 处理 `TcpConnection` 断开连接。

§8.7 介绍 `Buffer class` 并用它读取数据。

§8.8 介绍如何无阻塞发送数据。

§8.9 完善 `TcpConnection`，处理 `SIGPIPE`、`TCP keep alive` 等。

至此，单线程 TCP 服务端网络编程已经基本成型，大部分 `muduo` 示例都可以运行。

Acceptor class

先定义 `Acceptor class`，用于 `accept(2)` 新 TCP 连接，并通过回调通知使用者。它是内部 class，供 `TcpServer` 使用，生命期由后者控制。`Acceptor` 的接口如下：

```

26 class Acceptor : boost::noncopyable
27 {
28 public:
29     typedef boost::function<void (int sockfd,
30                                     const InetAddress&)> NewConnectionCallback;
31
32     Acceptor(EventLoop* loop, const InetAddress& listenAddr);
33
34                                     reactor/s04/Acceptor.h
  
```

```

34 void setNewConnectionCallback(const NewConnectionCallback& cb)
35 { newConnectionCallback_ = cb; }
36
37 bool listenning() const { return listenning_; }
38 void listen();

```

 reactor/s04/Acceptor.h

Acceptor 的数据成员包括 Socket、Channel 等。其中 Socket 是一个 RAII handle，封装了 socket 文件描述符的生命期。Acceptor 的 socket 是 listening socket，即 server socket。Channel 用于观察此 socket 上的 readable 事件，并回调 Acceptor::handleRead()，后者会调用 accept(2) 来接受新连接，并回调用户 callback。

```

40 private:
41 void handleRead();
42
43 EventLoop* loop_;
44 Socket acceptSocket_;
45 Channel acceptChannel_;
46 NewConnectionCallback newConnectionCallback_;
47 bool listenning_;
48 };

```

 reactor/s04/Acceptor.h

Acceptor 的构造函数和 Acceptor::listen() 成员函数执行创建 TCP 服务端的传统步骤，即调用 socket(2)、bind(2)、listen(2) 等 Sockets API，其中任何一个步骤出错都会造成程序终止²，因此这里看不到错误处理。

```

19 Acceptor::Acceptor(EventLoop* loop, const InetAddress& listenAddr)
20 : loop_(loop),
21   acceptSocket_(sockets::createNonblockingOrDie()),
22   acceptChannel_(loop, acceptSocket_.fd()),
23   listenning_(false)
24 {
25   acceptSocket_.setReuseAddr(true);
26   acceptSocket_.bindAddress(listenAddr);
27   acceptChannel_.setReadCallback(
28     boost::bind(&Acceptor::handleRead, this));
29 }
30
31 void Acceptor::listen()
32 {
33   loop_->assertInLoopThread();
34   listenning_ = true;
35   acceptSocket_.listen();
36   acceptChannel_.enableReading();
37 }

```

 reactor/s04/Acceptor.cc

² 通常原因是端口被占用。这时让程序异常退出更好，因为能触发监控系统报警，而不是假装正常运行。

Acceptor 的接口中用到了 InetAddress class, 这是对 struct sockaddr_in 的简单封装, 能自动转换字节序, 代码从略。InetAddress 具备值语义, 是可以拷贝的。

Acceptor 的构造函数用到 createNonblockingOrDie() 来创建非阻塞的 socket, 现在的 Linux 可以一步完成 (§4.11), 代码如下。

```

int sockets::createNonblockingOrDie()
{
    int sockfd = ::socket(AF_INET,
                          SOCK_STREAM | SOCK_NONBLOCK | SOCK_CLOEXEC,
                          IPPROTO_TCP);

    if (sockfd < 0)
    {
        LOG_SYSFATAL << "sockets::createNonblockingOrDie";
    }
    return sockfd;
}

```

Acceptor::listen() 的最后一步让 acceptChannel_ 在 socket 可读的时候调用 Acceptor::handleRead(), 后者会接受 (accept(2)) 并回调 newConnectionCallback_。这里直接把 socket fd 传给 callback, 这种传递 int 句柄的做法不够理想, 在 C++11 中可以先创建 Socket 对象, 再用移动语义把 Socket 对象 std::move() 给回调函数, 确保资源的安全释放。

```

39 void Acceptor::handleRead()
40 {
41     loop_>assertInLoopThread();
42     InetAddress peerAddr(0);
43     //FIXME loop until no more
44     int connfd = acceptSocket_.accept(&peerAddr);
45     if (connfd >= 0) {
46         if (newConnectionCallback_) {
47             newConnectionCallback_(connfd, peerAddr);
48         } else {
49             sockets::close(connfd);
50         }
51     }
52 }

```

注意这里的实现没有考虑文件描述符耗尽的情况, muduo 的处理办法见 §7.7。还有一个改进措施, 在拿到大于或等于 0 的 connfd 之后, 非阻塞地 poll(2) 一下, 看看 fd 是否可读写。正常情况下 poll(2) 会返回 writable, 表明 connfd 可用。如果 poll(2) 返回错误, 表明 connfd 有问题, 应该立刻关闭连接。

Acceptor::handleRead() 的策略很简单, 每次 accept(2) 一个 socket。另外还有两种实现策略, 一是每次循环 accept(2), 直至没有新的连接到达; 二是每次尝试 accept(2) N 个新连接, N 的值一般是 10。后面这两种做法适合短连接服务, 而 muduo 是为长连接服务优化的, 因此这里用了最简单的办法。这三种策略的对比见论文《accept()able Strategies for Improving Web Server Performance》³。

利用 Linux 新增的系统调用可以直接 accept(2) 一步得到非阻塞的 socket。

```

94 int sockets::accept(int sockfd, struct sockaddr_in* addr)
95 {
96     socklen_t addrlen = sizeof *addr;
97     #if VALGRIND
98     int connfd = ::accept(sockfd, sockaddr_cast(addr), &addrlen);
99     setNonBlockAndCloseOnExec(connfd);
100 #else
101     int connfd = ::accept4(sockfd, sockaddr_cast(addr),
102                             &addrlen, SOCK_NONBLOCK | SOCK_CLOEXEC);
103 #endif
104     if (connfd < 0)
105     {
106         int savedErrno = errno;
107         LOG_SYSERR << "Socket::accept";
108         switch (savedErrno)
109         {
133     }
134     }
135     return connfd;
136 }

```

这里区分致命错误和暂时错误, 并区别对待。对于暂时错误, 例如 EAGAIN、EINTR、EMFILE、ECONNABORTED 等等, 处理办法是忽略这次错误。对于致命错误, 例如 ENFILE、ENOMEM 等等, 处理办法是终止程序, 对于未知错误也照此办理。

下面写个小程序来试验 Acceptor 的功能, 它在 9981 端口侦听新连接, 连接到达后向它发送一个字符串, 随即断开连接。

```

7 void newConnection(int sockfd, const muduo::InetAddress& peerAddr)
8 {
9     printf("newConnection(): accepted a new connection from %s\n",
10           peerAddr.toHostPort().c_str());
11     ::write(sockfd, "How are you?\n", 13);
12     muduo::sockets::close(sockfd);
13 }

```

³ <http://static.usenix.org/event/usenix04/tech/general/brecht.html>

```

14
15 int main()
16 {
17     printf("main(): pid = %d\n", getpid());
18
19     muduo::InetAddress listenAddr(9981);
20     muduo::EventLoop loop;
21
22     muduo::Acceptor acceptor(&loop, listenAddr);
23     acceptor.setNewConnectionCallback(newConnection);
24     acceptor.listen();
25
26     loop.loop();
27 }

```

reactor/s04/test7.cc

练习 1：把 s04/test7.cc 改写为 daytime 服务器。

练习 2：把 s04/test7.cc 扩充为同时侦听两个 port，每个 port 发送不同的字符串。

8.5 TcpServer 接受新连接

本节会介绍 TcpServer 并初步实现 TcpConnection，本节只处理连接的建立，下一节处理连接的断开，再往后依次处理读取数据和发送数据。

TcpServer 新建连接的相关函数调用顺序见图 8-4（有的函数名是简写，省略了 poll(2) 调用）。其中 Channel::handleEvent() 的触发条件是 listening socket 可读，表明有新连接到达。TcpServer 会为新连接创建对应的 TcpConnection 对象。

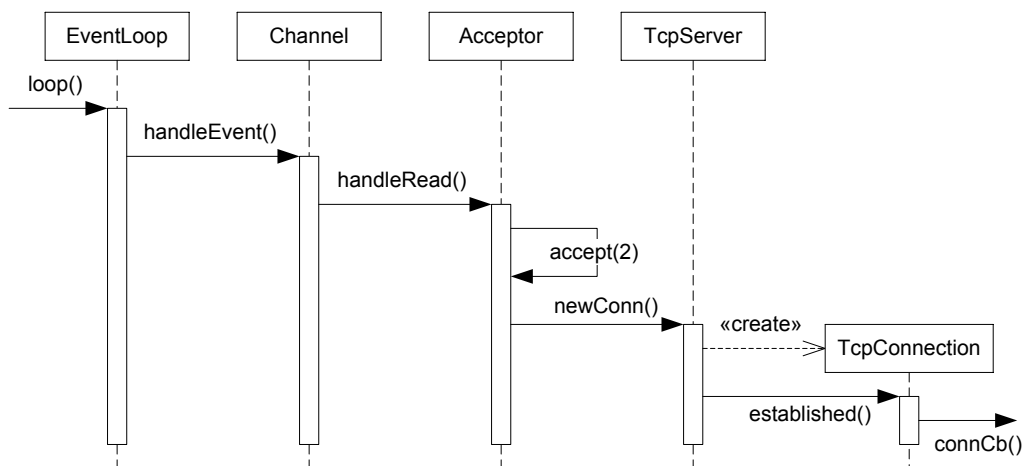


图 8-4

8.5.1 TcpServer class

TcpServer class 的功能是管理 accept(2) 获得的 TcpConnection。TcpServer 是供用户直接使用的，生命期由用户控制。TcpServer 的接口如下，用户只需要设置好 callback，再调用 start() 即可。

```

24 class TcpServer : boost::noncopyable
25 {
26 public:
27
28     TcpServer(EventLoop* loop, const InetAddress& listenAddr);
29     ~TcpServer(); // force out-line dtor, for scoped_ptr members.
30
31     /// Starts the server if it's not listening.
32     ///
33     /// It's harmless to call it multiple times.
34     /// Thread safe.
35     void start();
36
37     /// Set connection callback.
38     /// Not thread safe.
39     void setConnectionCallback(const ConnectionCallback& cb)
40     { connectionCallback_ = cb; }
41
42     /// Set message callback.
43     /// Not thread safe.
44     void setMessageCallback(const MessageCallback& cb)
45     { messageCallback_ = cb; }

```

TcpServer 内部使用 Acceptor 来获得新连接的 fd。它保存用户提供的 ConnectionCallback 和 MessageCallback，在新建 TcpConnection 的时候会原样传给后者。TcpServer 持有目前存活的 TcpConnection 的 shared_ptr（定义为 TcpConnectionPtr），因为 TcpConnection 对象的生命期是模糊的，用户也可以持有 TcpConnectionPtr。

```

47 private:
48     /// Not thread safe, but in loop
49     void newConnection(int sockfd, const InetAddress& peerAddr);
50
51     typedef std::map<std::string, TcpConnectionPtr> ConnectionMap;
52
53     EventLoop* loop_; // the acceptor loop
54     const std::string name_;
55     boost::scoped_ptr<Acceptor> acceptor_; // avoid revealing Acceptor
56     ConnectionCallback connectionCallback_;
57     MessageCallback messageCallback_;
58     bool started_;
59     int nextConnId_; // always in loop thread
60     ConnectionMap connections_;
61 };

```

每个 TcpConnection 对象有一个名字，这个名字是由其所属的 TcpServer 在创建 TcpConnection 对象时生成，名字是 ConnectionMap 的 key。

在新连接到达时，Acceptor 会回调 newConnection()，后者会创建 TcpConnection 对象 conn，把它加入 ConnectionMap，设置好 callback，再调用 conn->connectEstablished()，其中会回调用户提供的 ConnectionCallback。代码如下。

```

50 void TcpServer::newConnection(int sockfd, const InetAddress& peerAddr) reactor/s05/TcpServer.cc
51 {
52     loop_->assertInLoopThread();
53     char buf[32];
54     snprintf(buf, sizeof buf, "%d", nextConnId_);
55     ++nextConnId_;
56     std::string connName = name_ + buf;
57
58     LOG_INFO << "TcpServer::newConnection [" << name_
59               << "]" - new connection [" << connName
60               << "]" from " << peerAddr.toHostPort();
61     InetAddress localAddr(sockets::getLocalAddr(sockfd));
62     // FIXME poll with zero timeout to double confirm the new connection
63     TcpConnectionPtr conn(
64         new TcpConnection(loop_, connName, sockfd, localAddr, peerAddr));
65     connections_[connName] = conn;
66     conn->setConnectionCallback(connectionCallback_);
67     conn->setMessageCallback(messageCallback_);
68     conn->connectEstablished();
69 }

```

练习：给 TcpServer 的构造函数增加 string 参数，用于初始化 name_ 成员变量。

注意 muduo 尽量让依赖是单向的，TcpServer 会用到 Acceptor，但 Acceptor 并不知道 TcpServer 的存在。TcpServer 会创建 TcpConnection，但 TcpConnection 并不知道 TcpServer 的存在。另外 L64 可以考虑改用 make_shared() 以节约一次 new。

8.5.2 TcpConnection class

TcpConnection class 可谓是 muduo 最核心也是最复杂的 class，它的头文件和源文件一共有 450 多行，是 muduo 最大的 class。本章会用 5 节的篇幅来逐渐完善它。

TcpConnection 是 muduo 里唯一默认使用 shared_ptr 来管理的 class，也是唯一继承 enable_shared_from_this 的 class，这源于其模糊的生命期，原因见 §4.7。

```

21 class TcpConnection; reactor/s05/Callbacks.h
22 typedef boost::shared_ptr<TcpConnection> TcpConnectionPtr;

```

```

30 class TcpConnection : boost::noncopyable,
31                       public boost::enable_shared_from_this<TcpConnection>
32 {
33 public:

```

本节的 `TcpConnection` 没有可供用户使用的函数，因此接口从略，以下是其数据成员。目前 `TcpConnection` 的状态只有两个，`kConnecting` 和 `kConnected`，后面几节会逐渐丰富其状态。`TcpConnection` 使用 `Channel` 来获得 `socket` 上的 IO 事件，它会自己处理 `writable` 事件，而把 `readable` 事件通过 `MessageCallback` 传达给客户。`TcpConnection` 拥有 `TCP socket`，它的析构函数会 `close(fd)`（在 `Socket` 的析构函数中发生）。

```

61 private:
62     enum StateE { kConnecting, kConnected, };
63
64     void setState(StateE s) { state_ = s; }
65     void handleRead();
66
67     EventLoop* loop_;
68     std::string name_;
69     StateE state_; // FIXME: use atomic variable
70     // we don't expose those classes to client.
71     boost::scoped_ptr<Socket> socket_;
72     boost::scoped_ptr<Channel> channel_;
73     InetAddress localAddr_;
74     InetAddress peerAddr_;
75     ConnectionCallback connectionCallback_;
76     MessageCallback messageCallback_;
77 };

```

注意 `TcpConnection` 表示的是“一次 TCP 连接”，它是不可再生的，一旦连接断开，这个 `TcpConnection` 对象就没啥用了。另外 `TcpConnection` 没有发起连接的功能，其构造函数的参数是已经建立好连接的 `socket fd`（无论是 `TcpServer` 被动接受还是 `TcpClient` 主动发起），因此其初始状态是 `kConnecting`。

本节的 `MessageCallback` 定义很原始，没有使用 `Buffer class`，而只是把 `(const char* buf, int len)` 传给用户，这种接口用起来无疑是很不方便的。

```

57 void TcpConnection::handleRead()
58 {
59     char buf[65536];
60     ssize_t n = ::read(channel_>fd(), buf, sizeof buf);
61     messageCallback_(shared_from_this(), buf, n);
62     // FIXME: close connection if n == 0
63 }

```

本节的 TcpConnection 只处理了建立连接，没有处理断开连接（例如 handleRead() 中的 read(2) 返回 0），接收数据的功能很简陋，也不支持发送数据，这些都会逐步得到完善。

s05/test8.cc 试验了目前实现的功能，它实际上是个 discard 服务。但目前它永远不会关闭 socket，即永远走不到 else 分支（L14），在遇到对方断开连接的时候会陷入 busy loop。§8.6 会处理连接的断开。

```

6 void onConnection(const muduo::TcpConnectionPtr& conn)
7 {
8     if (conn->connected())
9     {
10         printf("onConnection(): new connection [%s] from %s\n",
11               conn->name().c_str(),
12               conn->peerAddress().toHostPort().c_str());
13     }
14     else
15     {
16         printf("onConnection(): connection [%s] is down\n",
17               conn->name().c_str());
18     }
19 }
20
21 void onMessage(const muduo::TcpConnectionPtr& conn,
22               const char* data,
23               ssize_t len)
24 {
25     printf("onMessage(): received %zd bytes from connection [%s]\n",
26           len, conn->name().c_str());
27 }
28
29 int main()
30 {
31     printf("main(): pid = %d\n", getpid());
32
33     muduo::InetAddress listenAddr(9981);
34     muduo::EventLoop loop;
35
36     muduo::TcpServer server(&loop, listenAddr);
37     server.setConnectionCallback(onConnection);
38     server.setMessageCallback(onMessage);
39     server.start();
40
41     loop.loop();
42 }

```

以上代码看起来和 muduo 的一般用法已经很接近了。

8.6 TcpConnection 断开连接

muduo 只有一种关闭连接的方式：被动关闭（p. 191）。即对方先关闭连接，本地 `read(2)` 返回 0，触发关闭逻辑。将来如果有必要也可以给 `TcpConnection` 新增 `forceClose()` 成员函数，用于主动关闭连接，实现很简单，调用 `handleClose()` 即可。函数调用的流程见图 8-5，其中的“X”表示 `TcpConnection` 通常会在此时析构。

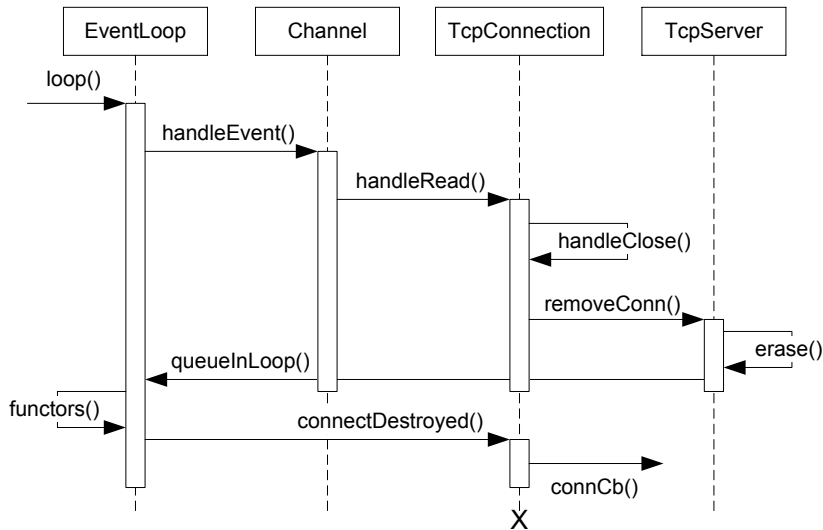


图 8-5

一般来讲数据的删除比新建要复杂，TCP 连接也不例外。关闭连接的流程看上去有点“绕”，根本原因是 p. 274 讲的对象生命期管理的需要。

Channel 的改动

`Channel` class 新增了 `CloseCallback` 事件回调，并且断言（`assert()`）在事件处理期间本 `Channel` 对象不会析构，即不会发生 p. 274 讲的出错情况。

```

32 +Channel::~Channel()
33 +{
34 +  assert(!eventHandling_);
35 +}

42 void Channel::handleEvent()
43 {
44 +  eventHandling_ = true;
45   if (revents_ & POLLNVAL) {
46     LOG_WARN << "Channel::handle_event() POLLNVAL";
47   }

```

reactor/s06/Channel.cc

```

48
49 + if ((revents_ & POLLHUP) && !(revents_ & POLLIN)) {
50 +     LOG_WARN << "Channel::handle_event() POLLHUP";
51 +     if (closeCallback_) closeCallback_();
52 + }
53     if (revents_ & (POLLERR | POLLNVAL)) {
54         if (errorCallback_) errorCallback_();
55     }
56     if (revents_ & (POLLIN | POLLPRI | POLLRDHUP)) {
57         if (readCallback_) readCallback_();
58     }
59     if (revents_ & POLLOUT) {
60         if (writeCallback_) writeCallback_();
61     }
62 +     eventHandling_ = false;
63 }

```

reactor/s06/Channel.cc

TcpConnection 的改动

TcpConnection class 也新增了 CloseCallback 事件回调, 但是这个回调是给 TcpServer 和 TcpClient 用的, 用于通知它们移除所持有的 TcpConnectionPtr, 这不是给普通用户用的, 普通用户继续使用 ConnectionCallback。

```

56     /// Internal use only.
57 + void setCloseCallback(const CloseCallback& cb)
58 + { closeCallback_ = cb; }
59
60     // called when TcpServer accepts a new connection
61     void connectEstablished(); // should be called only once
62 + // called when TcpServer has removed me from its map
63 + void connectDestroyed(); // should be called only once

```

reactor/s06/TcpConnection.h

TcpConnection 把另外几个 handle*() 事件处理函数也补上了, handleWrite() 暂时为空。Channel 的 CloseCallback 会调用 TcpConnection::handleClose(), 依此类推。

```

65     private:
66     ! enum StateE { kConnecting, kConnected, kDisconnected, };
67
68     void setState(StateE s) { state_ = s; }
69     void handleRead();
70 + void handleWrite();
71 + void handleClose();
72 + void handleError();

```

reactor/s06/TcpConnection.h

TcpConnection::handleRead() 会检查 read(2) 的返回值, 根据返回值分别调用 messageCallback_、handleClose()、handleError()。

```

74 void TcpConnection::handleRead()
75 {
76     char buf[65536];
77     ssize_t n = ::read(channel_>fd(), buf, sizeof buf);
78 + if (n > 0) {
79     messageCallback_(shared_from_this(), buf, n);
80 + } else if (n == 0) {
81 +     handleClose();
82 + } else {
83 +     handleError();
84 + }
85 }

```

`TcpConnection::handleClose()` 的主要功能是调用 `closeCallback_`，这个回调绑定到 `TcpServer::removeConnection()`。

```

91 void TcpConnection::handleClose()
92 {
93     loop_>assertInLoopThread();
94     LOG_TRACE << "TcpConnection::handleClose state = " << state_;
95     assert(state_ == kConnected);
96     // we don't close fd, leave it to dtor, so we can find leaks easily.
97     channel_>disableAll();
98     // must be the last line
99     closeCallback_(shared_from_this());
100 }

```

`TcpConnection::handleError()` 并没有进一步的行动，只是在日志中输出错误消息，这不影响连接的正常关闭。

```

102 void TcpConnection::handleError()
103 {
104     int err = sockets::getSocketError(channel_>fd());
105     LOG_ERROR << "TcpConnection::handleError [" << name_
106         << "]" - SO_ERROR = " << err << " " << strerror_tl(err);
107 }

```

`TcpConnection::connectDestroyed()` 是 `TcpConnection` 析构前最后调用的一个成员函数，它通知用户连接已断开。其中的 L68 与上面的 L97 重复，这是因为在某些情况下可以不经由 `handleClose()` 而直接调用 `connectDestroyed()`。

```

63 void TcpConnection::connectDestroyed()
64 {
65     loop_>assertInLoopThread();
66     assert(state_ == kConnected);
67     setState(kDisconnected);
68     channel_>disableAll();
69     connectionCallback_(shared_from_this());
70
71     loop_>removeChannel(get_pointer(channel_));
72 }

```

reactor/s06/TcpConnection.cc

TcpServer 的改动

TcpServer 向 TcpConnection 注册 CloseCallback，用于接收连接断开的消息。

```

50 void TcpServer::newConnection(int sockfd, const InetAddress& peerAddr)
51 {

```

此处省略没有变化的代码。

```

63     TcpConnectionPtr conn(
64         new TcpConnection(loop_, connName, sockfd, localAddr, peerAddr));
65     connections_[connName] = conn;
66     conn->setConnectionCallback(connectionCallback_);
67     conn->setMessageCallback(messageCallback_);
68 +   conn->setCloseCallback(
69 +       boost::bind(&TcpServer::removeConnection, this, _1));
70     conn->connectEstablished();
71 }

```

通常 TcpServer 的生命期长于它建立的 TcpConnection，因此不用担心 TcpServer 对象失效。在 muduo 中，TcpServer 的析构函数会关闭连接，因此也是安全的。

TcpServer::removeConnection() 把 conn 从 ConnectionMap 中移除。这时 TcpConnection 已经是命悬一线：如果用户不持有 TcpConnectionPtr 的话，conn 的引用计数已降到 1。注意这里一定要用 EventLoop::queueInLoop()，否则就会出现 p. 274 讲的对象生命期管理问题。另外注意这里用 boost::bind 让 TcpConnection 的生命期长到调用 connectDestroyed() 的时刻。

```

73 void TcpServer::removeConnection(const TcpConnectionPtr& conn)
74 {
75     loop_->assertInLoopThread();
76     LOG_INFO << "TcpServer::removeConnection [" << name_
77         << "]" - connection " << conn->name();
78     size_t n = connections_.erase(conn->name());
79     assert(n == 1); (void)n;
80     loop_->queueInLoop(
81         boost::bind(&TcpConnection::connectDestroyed, conn));
82 }

```

思考并验证：如果用户不持有 TcpConnectionPtr，那么 TcpConnection 对象究竟在什么时候析构？

有兴趣的读者可以单步跟踪连接断开的流程，s06/test8.cc 不会陷入 busy loop。目前的做法不是最简洁的，但是可以几乎原封不动地用到多线程 TcpServer 中 (§8.10)。

EventLoop 和 Poller 的改动

本节 TcpConnection 不再是只生不灭，因此要求 EventLoop 也提供 unregister 功能。EventLoop 新增了 removeChannel() 成员函数，它会调用 Poller::removeChannel(), 后者定义如下，复杂度为 $O(\log N)$ 。

```

95 void Poller::removeChannel(Channel* channel)
96 {
97     assertInLoopThread();
98     LOG_TRACE << "fd = " << channel->fd();
99     assert(channels_.find(channel->fd()) != channels_.end());
100    assert(channels_[channel->fd()] == channel);
101    assert(channel->isNoneEvent());
102    int idx = channel->index();
103    assert(0 <= idx && idx < static_cast<int>(pollfds_.size()));
104    const struct pollfd& pfd = pollfds_[idx]; (void)pfd;
105    assert(pfd.fd == -channel->fd()-1 && pfd.events == channel->events());
106    size_t n = channels_.erase(channel->fd());
107    assert(n == 1); (void)n;
108    if (implicit_cast<size_t>(idx) == pollfds_.size()-1) {
109        pollfds_.pop_back();
110    } else {
111        int channelAtEnd = pollfds_.back().fd;
112        iter_swap(pollfds_.begin()+idx, pollfds_.end()-1);
113        if (channelAtEnd < 0) {
114            channelAtEnd = -channelAtEnd-1;
115        }
116        channels_[channelAtEnd]->set_index(idx);
117        pollfds_.pop_back();
118    }
119 }

```

reactor/s06/Poller.cc

注意其中从数组 pollfds_ 中删除元素是 $O(1)$ 复杂度，办法是将待删除的元素与最后一个元素交换，再 pollfds_.pop_back()。这需要相应地修改 p. 286 的代码：

```

79     // update existing one
80     assert(channels_.find(channel->fd()) != channels_.end());
81     assert(channels_[channel->fd()] == channel);
82     int idx = channel->index();
83     assert(0 <= idx && idx < static_cast<int>(pollfds_.size()));
84     struct pollfd& pfd = pollfds_[idx];
85     ! assert(pfd.fd == channel->fd() || pfd.fd == -channel->fd()-1);
86     pfd.events = static_cast<short>(channel->events());
87     pfd.revents = 0;
88     if (channel->isNoneEvent()) {
89         // ignore this pollfd
90     !     pfd.fd = -channel->fd()-1;
91     }

```

reactor/s06/Poller.cc

8.7 Buffer 读取数据

Buffer 是非阻塞 TCP 网络编程必不可少的东西 (§7.4)，本节介绍用 Buffer 来处理数据输入，下一节介绍数据输出。Buffer 是另一个具有值语义的对象。

首先修改 s07/Callbacks.h 中 MessageCallback 的定义，现在的参数和 muduo 一样，是 Buffer* 和 Timestamp，不再是原始的 (const char* buf, int len)。

```
27 typedef boost::function<void (const TcpConnectionPtr&,
28                               Buffer* buf,
29                               Timestamp)> MessageCallback;
```

其中 Timestamp 是 poll(2) 返回的时刻，即消息到达的时刻，这个时刻早于读到数据的时刻 (read(2) 调用或返回)。因此如果要比较准确地测量程序处理消息的内部延迟，应该以此时刻为起点，否则测出来的结果偏小，特别是处理并发连接时效果更明显。(为什么?) 为此我们需要修改 Channel 中 ReadEventCallback 的原型，改动如下。EventLoop::loop() 也需要有相应的改动，此处从略。

```
reactor/s07/Channel.h
27 class Channel : boost::noncopyable
28 {
29     public:
30         typedef boost::function<void()> EventCallback;
31 +     typedef boost::function<void(Timestamp)> ReadEventCallback;
32
33         Channel(EventLoop* loop, int fd);
34         ~Channel();
35
36 !     void handleEvent(Timestamp receiveTime);
37 !     void setReadCallback(const ReadEventCallback& cb)
38         { readCallback_ = cb; }
```

reactor/s07/Channel.h

s07/test3.cc 试验了以上改动:

```
reactor/s07/test3.cc
9 !void timeout(muduo::Timestamp receiveTime)
10 {
11 !     printf("%s Timeout!\n", receiveTime.toFormattedString().c_str());
12     g_loop->quit();
13 }
14
15 int main()
16 {
17 +     printf("%s started\n", muduo::Timestamp::now().toFormattedString().c_str());
18     muduo::EventLoop loop;
19     g_loop = &loop;
```

reactor/s07/test3.cc

8.7.1 TcpConnection 使用 Buffer 作为输入缓冲

先给 TcpConnection 添加 inputBuffer_ 成员变量。

```

83     ConnectionCallback connectionCallback_;
84     MessageCallback messageCallback_;
85     CloseCallback closeCallback_;
86 +   Buffer inputBuffer_;

```

reactor/s07/TcpConnection.h
reactor/s07/TcpConnection.h

然后修改 TcpConnection::handleRead() 成员函数，使用 Buffer 来读取数据。

```

74 !void TcpConnection::handleRead(Timestamp receiveTime)
75 {
76 !   int savedErrno = 0;
77 !   ssize_t n = inputBuffer_.readFd(channel_>fd(), &savedErrno);
78   if (n > 0) {
79 !       messageCallback_(shared_from_this(), &inputBuffer_, receiveTime);
80   } else if (n == 0) {
81       handleClose();
82   } else {
83 +       errno = savedErrno;
84 +       LOG_SYSERR << "TcpConnection::handleRead";
85       handleError();
86   }
87 }

```

reactor/s07/TcpConnection.cc
reactor/s07/TcpConnection.cc

修改 s07/test8.cc 以试验本次改动后的新功能。

```

21 void onMessage(const muduo::TcpConnectionPtr& conn,
22 !             muduo::Buffer* buf,
23 !             muduo::Timestamp receiveTime)
24 {
25 +   printf("onMessage(): received %zd bytes from connection [%s] at %s\n",
26 +         buf->readableBytes(),
27 +         conn->name().c_str(),
28 +         receiveTime.toFormattedString().c_str());
29 +
30 +   printf("onMessage(): [%s]\n", buf->retrieveAsString().c_str());
31 }

```

reactor/s07/test8.cc
reactor/s07/test8.cc

这个测试程序看上去和 muduo 的正式用法没有区别。

8.7.2 Buffer::readFd()

我在 p. 208 提到 Buffer 读取数据时兼顾了内存使用量和效率，其实现如下。

```

18 ssize_t Buffer::readFd(int fd, int* savedErrno)
19 {
20     char extrabuf[65536];
21     struct iovec vec[2];
22     const size_t writable = writableBytes();
23     vec[0].iov_base = begin()+writerIndex_;
24     vec[0].iov_len = writable;
25     vec[1].iov_base = extrabuf;
26     vec[1].iov_len = sizeof extrabuf;
27     const ssize_t n = readv(fd, vec, 2);
28     if (n < 0) {
29         *savedErrno = errno;
30     } else if (implicit_cast<size_t>(n) <= writable) {
31         writerIndex_ += n;
32     } else {
33         writerIndex_ = buffer_.size();
34         append(extrabuf, n - writable);
35     }
36     return n;
37 }

```

这个实现有几点值得一提。一是使用了 scatter/gather IO，并且一部分缓冲区取自 stack，这样输入缓冲区足够大，通常一次 readv(2) 调用就能取完全部数据⁴。由于输入缓冲区足够大，也节省了一次 ioctl(socketFd, FIONREAD, &length) 系统调用，不必事先知道有多少数据可读而提前预留 (reserve()) Buffer 的 capacity()，可以在一次读取之后将 extrabuf 中的数据 append() 给 Buffer。

二是 Buffer::readFd() 只调用一次 read(2)，而没有反复调用 read(2) 直到其返回 EAGAIN。首先，这么做是正确的，因为 muduo 采用 level trigger，这么做不会丢失数据或消息。其次，对追求低延迟的程序来说，这么做是高效的，因为每次读数据只需要一次系统调用。再次，这样做照顾了多个连接的公平性，不会因为某个连接上数据量过大而影响其他连接处理消息。

假如 muduo 采用 edge trigger，那么每次 handleRead() 至少调用两次 read(2)，平均起来比 level trigger 多一次系统调用，edge trigger 不见得更高效。

将来的一个改进措施是：如果 `n == writable + sizeof extrabuf`，就再读一次。

⁴ 在一个不繁忙（没有出现消息堆积）的系统上，程序一般等待在 poll(2) 上，一有数据到达就会立刻唤醒应用程序来读取，那么每次 read() 的数据不会超过几 KiB（一两个以太网 frame），这里 64KiB 缓冲足够容纳千兆网在 500μs 内全速收到的数据，在一定意义下可视为延迟带宽积（bandwidth-delay product）。

8.8 TcpConnection 发送数据

发送数据比接收数据更难，因为发送数据是主动的，接收读取数据是被动的。这也是本章先介绍 TcpServer 后介绍 TcpClient 的原因。到目前为止，我们只用到了 Channel 的 ReadCallback：

- TimerQueue 用它来读 timerfd(2)。
- EventLoop 用它来读 eventfd(2)。
- TcpServer/Acceptor 用它来读 listening socket。
- TcpConnection 用它来读普通 TCP socket。

本节会动用其 WriteCallback，由于 muduo 采用 level trigger，因此我们只在需要时才关注 writable 事件，否则就会造成 busy loop。s08/Channel.h 的改动如下：

```

51     void enableReading() { events_ |= kReadEvent; update(); }
52 !   void enableWriting() { events_ |= kWriteEvent; update(); }
53 !   void disableWriting() { events_ &= ~kWriteEvent; update(); }
54     void disableAll() { events_ = kNoneEvent; update(); }
55 +   bool isWriting() const { return events_ & kWriteEvent; }

```

reactor/s08/Channel.h

TcpConnection 的接口中增加了 send() 和 shutdown() 两个函数，这两个函数都可以跨线程调用。为了简单起见，本章只提供一种 send() 重载。

```

51 + //void send(const void* message, size_t len);
52 + // Thread safe.
53 + void send(const std::string& message);
54 + // Thread safe.
55 + void shutdown();

```

reactor/s08/TcpConnection.h

TcpConnection 的状态增加到了 4 个，和目前 muduo 的实现一致。

```
enum StateE { kConnecting, kConnected, kDisconnecting, kDisconnected, };
```

其内部实现增加了两个 *InLoop 成员函数，对应前面的两个新接口函数，并使用 Buffer 作为输出缓冲区。

```

78     void handleClose();
79     void handleError();
80 +   void sendInLoop(const std::string& message);
81 +   void shutdownInLoop();

94     Buffer inputBuffer_;
95 +   Buffer outputBuffer_;

```

reactor/s08/TcpConnection.h

TcpConnection 有一个非常简单的状态图（见图 8-6）。

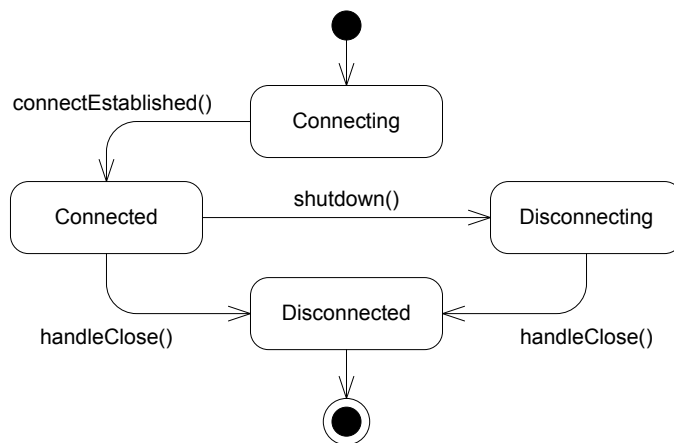


图 8-6

TcpConnection 在关闭连接的过程中与其他操作（读写事件）的交互比较复杂，尚需完备的单元测试来验证各种时序下的正确性。必要时可能要新增状态。

shutdown() 是线程安全的，它会把实际工作放到 shutdownInLoop() 中来做，后者保证在 IO 线程调用。如果当前没有正在写入，则关闭写入端（p. 191）。代码注释给出了两个值得改进的地方。

```

94 void TcpConnection::shutdown()
95 {
96     // FIXME: use compare and swap
97     if (state_ == kConnected)
98     {
99         setState(kDisconnecting);
100         // FIXME: shared_from_this()?
101         loop_->runInLoop(boost::bind(&TcpConnection::shutdownInLoop, this));
102     }
103 }
104
105 void TcpConnection::shutdownInLoop()
106 {
107     loop_->assertInLoopThread();
108     if (!channel_->isWriting())
109     {
110         // we are not writing
111         socket_->shutdownWrite();
112     }
113 }
  
```

reactor/s08/TcpConnection.cc

由于新增了 kDisconnecting 状态，TcpConnection::connectDestroyed() 和 TcpConnection::handleClose() 中的 assert() 也需要相应的修改，代码从略。

send() 也是一样的, 如果在非 IO 线程调用, 它会把 message 复制一份, 传给 IO 线程中的 sendInLoop() 来发送。这么做或许有轻微的效率损失, 但是线程安全性很容易验证, 我认为还是利大于弊。如果真的在乎这点性能, 不如让程序只在 IO 线程调用 send()。另外在 C++11 中可以使用移动语义, 避免内存拷贝的开销。

```

54 void TcpConnection::send(const std::string& message)
55 {
56     if (state_ == kConnected) {
57         if (loop_->isInLoopThread()) {
58             sendInLoop(message);
59         } else {
60             loop_->runInLoop(
61                 boost::bind(&TcpConnection::sendInLoop, this, message));
62         }
63     }
64 }

```

sendInLoop() 会先尝试直接发送数据, 如果一次发送完毕就不会启用 WriteCallback; 如果只发送了部分数据, 则把剩余的数据放入 outputBuffer_, 并开始关注 writable 事件, 以后在 handlerWrite() 中发送剩余的数据。如果当前 outputBuffer_ 已经有待发送的数据, 那么就不能先尝试发送了, 因为这会造成数据乱序。

```

66 void TcpConnection::sendInLoop(const std::string& message)
67 {
68     loop_->assertInLoopThread();
69     ssize_t nwrote = 0;
70     // if no thing in output queue, try writing directly
71     if (!channel_->isWriting() && outputBuffer_.readableBytes() == 0) {
72         nwrote = ::write(channel_->fd(), message.data(), message.size());
73         if (nwrote >= 0) {
74             if (implicit_cast<size_t>(nwrote) < message.size()) {
75                 LOG_TRACE << "I am going to write more data";
76             }
77         } else {
78             nwrote = 0;
79             if (errno != EWOULDBLOCK) {
80                 LOG_SYSERR << "TcpConnection::sendInLoop";
81             }
82         }
83     }
84
85     assert(nwrote >= 0);
86     if (implicit_cast<size_t>(nwrote) < message.size()) {
87         outputBuffer_.append(message.data()+nwrote, message.size()-nwrote);
88         if (!channel_->isWriting()) {
89             channel_->enableWriting();
90         }
91     }
92 }

```

当 socket 变得可写时, Channel 会调用 `TcpConnection::handleWrite()`, 这里我们继续发送 `outputBuffer_` 中的数据。一旦发送完毕, 立刻停止观察 writable 事件 (L160), 避免 busy loop。另外如果这时连接正在关闭 (L161), 则调用 `shutdownInLoop()`, 继续执行关闭过程。这里不需要处理错误, 因为一旦发生错误, `handleRead()` 会读到 0 字节, 继而关闭连接。

```

150 void TcpConnection::handleWrite()
151 {
152     loop_->assertInLoopThread();
153     if (channel_>isWriting()) {
154         ssize_t n = ::write(channel_>fd(),
155                             outputBuffer_.peek(),
156                             outputBuffer_.readableBytes());
157         if (n > 0) {
158             outputBuffer_.retrieve(n);
159             if (outputBuffer_.readableBytes() == 0) {
160                 channel_>disableWriting();
161                 if (state_ == kDisconnecting) {
162                     shutdownInLoop();
163                 }
164             } else {
165                 LOG_TRACE << "I am going to write more data";
166             }
167         } else {
168             LOG_SYSERR << "TcpConnection::handleWrite";
169         }
170     } else {
171         LOG_TRACE << "Connection is down, no more writing";
172     }
173 }

```

reactor/s08/TcpConnection.cc

注意 `sendInLoop()` 和 `handleWrite()` 都只调用了一次 `write(2)` 而不会反复调用直至它返回 `EAGAIN`, 原因是如果第一次 `write(2)` 没有能够发送完全部数据的话, 第二次调用 `write(2)` 几乎肯定会返回 `EAGAIN`。读者可以很容易用下面的 Python 代码来验证这一点。因此 muduo 决定节省一次系统调用, 这么做不影响程序的正确性, 却能降低延迟。

```

#!/usr/bin/python
import socket, sys

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('remote_hostname', 9876)) # 这里最好连接到网络上的一台机器
sock.setblocking(0)
a = 'a' * int(sys.argv[1]) # 两条消息的长度由命令行给出, a 应该足够大
b = 'b' * int(sys.argv[2])

```



```

n1 = sock.send(a)      # 第一次发送
n2 = 0
try:
    n2 = sock.send(b)  # 第二次发送, 遇到 EAGAIN 会抛 socket.error 异常
except socket.error as ex:
    print ex           # socket.error: [Errno 11] Resource temporarily unavailable
print n1
print n2
sock.close()

```

一个改进措施: TcpConnection 的输出缓冲区不必是连续的 (outputBuffer_ 改成 ptr_vector<Buffer>), handleWrite() 可以用 writev(2) 来发送多块数据, 这样或许能减小内存拷贝的次数, 略微提高性能 (但这种性能提高不一定能被外界感知)。

在 level trigger 模式中, 数据的发送比较麻烦, 因为不能一直关注 writable 事件, 不过数据的读取很简单。我认为理想的做法是对 readable 事件采用 level trigger, 对 writable 事件采用 edge trigger, 但是目前 Linux 不支持这种设定。

s08/test9.cc 是 echo server (§6.4.2), 代码从略。s08/test10.cc 试验 TcpConnection::send() 的功能, 它和前面的 Python 示例相近, 都是通过命令行指定两条消息的大小, 然后连续发送两条消息。通过选择不同的消息长度, 可以试验不同的 code path。

```

9 void onConnection(const muduo::TcpConnectionPtr& conn)
10 {
11     if (conn->connected())
12     {
13         printf("onConnection(): new connection [%s] from %s\n",
14             conn->name().c_str(),
15             conn->peerAddress().toHostPort().c_str());
16         conn->send(message1);
17         conn->send(message2);
18         conn->shutdown();
19     }
20     else
21     {
22         printf("onConnection(): connection [%s] is down\n",
23             conn->name().c_str());
24     }
25 }

```

reactor/s08/test10.cc

8.9 完善 TcpConnection

至此 TcpConnection 的主体功能接近完备, 可以应付大部分 muduo 示例的需求了。本节补充几个小功能, 让它成为可以实用的单线程非阻塞 TCP 网络库。

8.9.1 SIGPIPE

SIGPIPE 的默认行为是终止进程，在命令行程序中这是合理的⁵，但是在网络编程中，这意味着如果对方断开连接而本地继续写入的话，会造成服务进程意外退出。

假如服务进程繁忙，没有及时处理对方断开连接的事件，就有可能出现在连接断开之后继续发送数据的情况。下面这个例子模拟了这种情况：

```

10 void onConnection(const muduo::TcpConnectionPtr& conn)
11 {
12     if (conn->connected())
13     {
14         printf("onConnection(): new connection [%s] from %s\n",
15             conn->name().c_str(),
16             conn->peerAddress().toHostPort().c_str());
17 +     if (sleepSeconds > 0)
18 +     {
19 +         ::sleep(sleepSeconds);
20 +     }
21     conn->send(message1);
22     conn->send(message2);
23     conn->shutdown();
24 }
```

reactor/s09/test10.cc

reactor/s09/test10.cc

假设 sleepSeconds 是 5 秒，用 nc localhost 9981 创建连接之后立刻 Ctrl-C 断开客户端，服务进程过几秒就会退出。解决办法很简单，在程序开始的时候就忽略 SIGPIPE，可以用 C++ 全局对象做到这一点。

```

38 class IgnoreSigPipe
39 {
40 public:
41     IgnoreSigPipe()
42     {
43         ::signal(SIGPIPE, SIG_IGN);
44     }
45 };
46
47 IgnoreSigPipe initObj;
```

reactor/s09/EventLoop.cc

reactor/s09/EventLoop.cc

8.9.2 TCP No Delay 和 TCP keepalive

TCP No Delay 和 TCP keepalive 都是常用的 TCP 选项，前者的作用是禁用 Nagle 算法⁶，避免连续发包出现延迟，这对编写低延迟网络服务很重要。后者的作用是定

⁵ 见 p. 104 脚注的例子。

⁶ http://en.wikipedia.org/wiki/Nagle's_algorithm

期探查 TCP 连接是否还存在。一般来说如果有应用层心跳的话，TCP keepalive 不是必需的⁷，但是一个通用的网络库应该暴露其接口。（本书不涉及 TCP_CORK。）

以下是 `TcpConnection::setTcpNoDelay()` 的实现，涉及 3 个文件。

```

----- reactor/s09/TcpConnection.h
55     void shutdown();
56 +   void setTcpNoDelay(bool on);
----- reactor/s09/TcpConnection.h

----- reactor/s09/TcpConnection.cc
118 void TcpConnection::setTcpNoDelay(bool on)
119 {
120     socket_>setTcpNoDelay(on);
121 }
----- reactor/s09/TcpConnection.cc

----- reactor/s09/Socket.cc
60 void Socket::setTcpNoDelay(bool on)
61 {
62     int optval = on ? 1 : 0;
63     ::setsockopt(sockfd_, IPPROTO_TCP, TCP_NODELAY,
64                  &optval, sizeof optval);
65     // FIXME CHECK
66 }
----- reactor/s09/Socket.cc

```

`TcpConnection::setKeepAlive()` 的实现与之类似，此处从略，可参考 muduo 源码。

8.9.3 WriteCompleteCallback 和 HighWaterMarkCallback

非阻塞网络编程的发送数据比读取数据要困难得多：一方面是 §8.8 提到的“什么时候关注 writable 事件”的问题，这只带来编码方面的难度；另一方面是如果发送数据的速度高于对方接收数据的速度，会造成数据在本地内存中堆积，这带来设计及安全性方面的难度。muduo 对此的解决办法是提供两个回调，有的网络库把它们称为“高水位回调”和“低水位回调”，muduo 使用 `HighWaterMarkCallback` 和 `WriteCompleteCallback` 这两个名字。`WriteCompleteCallback` 很容易理解，如果发送缓冲区被清空，就调用它。`TcpConnection` 有两处可能触发此回调：

```

----- reactor/s09/TcpConnection.cc
66 void TcpConnection::sendInLoop(const std::string& message)
67 {
68     loop_>assertInLoopThread();

```

⁷ 如果没有应用层心跳，而对方机器突然断电，那么本机不会收到 TCP 的 FIN 分节。在没有发送消息的情况下，这个“连接”可能一直保持下去。

```

69     ssize_t nwrote = 0;
70     // if no thing in output queue, try writing directly
71     if (!channel_>isWriting() && outputBuffer_.readableBytes() == 0) {
72         nwrote = ::write(channel_>fd(), message.data(), message.size());
73         if (nwrote >= 0) {
74             if (implicit_cast<size_t>(nwrote) < message.size()) {
75                 LOG_TRACE << "I am going to write more data";
76             } else if (writeCompleteCallback_) {
77                 loop_>queueInLoop(
78                     boost::bind(writeCompleteCallback_, shared_from_this()));
79             }
80         } else {
81             nwrote = 0;
82         }
83     }
84 }
85
86 ----- reactor/s09/TcpConnection.cc
87
88 ----- reactor/s09/TcpConnection.cc
89
157 void TcpConnection::handleWrite()
158 {
159     loop_>assertInLoopThread();
160     if (channel_>isWriting()) {
161         ssize_t n = ::write(channel_>fd(),
162                             outputBuffer_.peek(),
163                             outputBuffer_.readableBytes());
164         if (n > 0) {
165             outputBuffer_.retrieve(n);
166             if (outputBuffer_.readableBytes() == 0) {
167                 channel_>disableWriting();
168                 if (writeCompleteCallback_) {
169                     loop_>queueInLoop(
170                         boost::bind(writeCompleteCallback_, shared_from_this()));
171                 }
172                 if (state_ == kDisconnecting) {
173                     shutdownInLoop();
174                 }
175             }
176         }
177     }
178 }
179
180 ----- reactor/s09/TcpConnection.cc

```

TcpConnection 和 TcpServer 也需要相应地暴露 WriteCompleteCallback 的接口，代码从略。

s09/test11.cc 是 chargen 服务 (§7.1)，用到了 WriteCompleteCallback，代码从略。

另外一个有用的 callback 是 HighWaterMarkCallback，如果输出缓冲的长度超过用户指定的大小，就会触发回调（只在上升沿触发一次）。代码见 muduo，此处从略。

如果用非阻塞的方式写一个 proxy，proxy 有 C 和 S 两个连接 (§7.13)。只考虑 server 发给 client 的数据流（反过来也是一样），为了防止 server 发过来的数据撑爆 C 的输出缓冲区，一种做法是在 C 的 HighWaterMarkCallback 中停止读取 S 的数据，而在 C 的 WriteCompleteCallback 中恢复读取 S 的数据。这就跟用粗水管往水桶里灌水，用细水管从水桶中取水一个道理，上下两个水龙头要轮流开合，类似 PWM。

Linux 多线程服务端编程：使用 muduo C++ 网络库

8.10 多线程 TcpServer

本章的最后几节介绍三个主题：多线程 TcpServer、TcpClient、epoll(4)，主题之间相互独立。

本节介绍多线程 TcpServer，用到了 EventLoopThreadPool class。

EventLoopThreadPool

用 one loop per thread 的思想实现多线程 TcpServer 的关键步骤是在新建 TcpConnection 时从 event loop pool 里挑选一个 loop 给 TcpConnection 用。也就是说多线程 TcpServer 自己的 EventLoop 只用来接受新连接，而新连接会用其他 EventLoop 来执行 IO。（单线程 TcpServer 的 EventLoop 是与 TcpConnection 共享的。）muduo 的 event loop pool 由 EventLoopThreadPool class 表示，接口如下，实现从略。

```

27 class EventLoopThreadPool : boost::noncopyable
28 {
29     public:
30         EventLoopThreadPool(EventLoop* baseLoop);
31         ~EventLoopThreadPool();
32         void setThreadNum(int numThreads) { numThreads_ = numThreads; }
33         void start();
34         EventLoop* getNextLoop();
35
36     private:
37         EventLoop* baseLoop_;
38         bool started_;
39         int numThreads_;
40         int next_; // always in loop thread
41         boost::ptr_vector<EventLoopThread> threads_;
42         std::vector<EventLoop*> loops_;
43 };

```

reactor/s10/EventLoopThreadPool.h

reactor/s10/EventLoopThreadPool.h

TcpServer 每次新建一个 TcpConnection 就会调用 getNextLoop() 来取得 EventLoop，如果是单线程服务，每次返回的都是 baseLoop_，即 TcpServer 自己用的那个 loop。其中 setThreadNum() 的参数的意义见 TcpServer 代码注释。

```

25 class TcpServer : boost::noncopyable
26 {
27     public:
28
29         TcpServer(EventLoop* loop, const InetAddress& listenAddr);
30         ~TcpServer(); // force out-line dtor, for scoped_ptr members.
31

```

reactor/s10/TcpServer.h

```

32 + /// Set the number of threads for handling input.
33 + ///
34 + /// Always accepts new connection in loop's thread.
35 + /// Must be called before @c start
36 + /// @param numThreads
37 + /// - 0 means all I/O in loop's thread, no thread will created.
38 + /// this is the default value.
39 + /// - 1 means all I/O in another thread.
40 + /// - N means a thread pool with N threads, new connections
41 + /// are assigned on a round-robin basis.
42 + void setThreadNum(int numThreads);

```

TcpServer 只用增加一个成员函数和一个成员变量。

```

65 private:
66     /// Not thread safe, but in loop
67     void newConnection(int sockfd, const InetAddress& peerAddr);
68 + /// Thread safe.
69     void removeConnection(const TcpConnectionPtr& conn);
70 + /// Not thread safe, but in loop
71 + void removeConnectionInLoop(const TcpConnectionPtr& conn);
72
73     typedef std::map<std::string, TcpConnectionPtr> ConnectionMap;
74
75     EventLoop* loop_; // the acceptor loop
76     const std::string name_;
77     boost::scoped_ptr<Acceptor> acceptor_; // avoid revealing Acceptor
78 + boost::scoped_ptr<EventLoopThreadPool> threadPool_;

```

reactor/s10/TcpServer.h

多线程 TcpServer 的改动很简单, 新建连接只改了 3 行代码。原来是把 TcpServer 自用的 loop_ 传给 TcpConnection, 现在是每次从 EventLoopThreadPool 取得 ioLoop。L81 的作用是让 TcpConnection 的 ConnectionCallback 由 ioLoop 线程调用。

```


```

reactor/s10/TcpServer.cc

```

59 void TcpServer::newConnection(int sockfd, const InetAddress& peerAddr)
60 {
61
62     InetAddress localAddr(sockets::getLocalAddr(sockfd));
63     // FIXME poll with zero timeout to double confirm the new connection
64 + EventLoop* ioLoop = threadPool_->getNextLoop();
65     TcpConnectionPtr conn(
66         new TcpConnection(ioLoop, connName, sockfd, localAddr, peerAddr));
67     connections_[connName] = conn;
68     conn->setConnectionCallback(connectionCallback_);
69     conn->setMessageCallback(messageCallback_);
70     conn->setWriteCompleteCallback(writeCompleteCallback_);
71     conn->setCloseCallback(
72         boost::bind(&TcpServer::removeConnection, this, _1)); // FIXME: unsafe
73     ! ioLoop->runInLoop(boost::bind(&TcpConnection::connectEstablished, conn));
74 }

```

reactor/s10/TcpServer.cc

连接的销毁也不复杂, 把原来的 `removeConnection()` 拆为两个函数, 因为 `TcpConnection` 会在自己的 `ioLoop` 线程调用 `removeConnection()`, 所以需要把它移到 `TcpServer` 的 `loop_` 线程 (因为 `TcpServer` 是无锁的)。L98 再次把 `connectDestroyed()` 移到 `TcpConnection` 的 `ioLoop` 线程进行, 是为了保证 `TcpConnection` 的 `ConnectionCallback` 始终在其 `ioLoop` 回调, 方便客户端代码的编写。

```

84 void TcpServer::removeConnection(const TcpConnectionPtr& conn)
85 {
86 + // FIXME: unsafe
87 + loop_->runInLoop(boost::bind(&TcpServer::removeConnectionInLoop, this, conn));
88 +}
89
90 +void TcpServer::removeConnectionInLoop(const TcpConnectionPtr& conn)
91 +{
92     loop_->assertInLoopThread();
93     ! LOG_INFO << "TcpServer::removeConnectionInLoop [" << name_
94         << "]" - connection " << conn->name();
95     size_t n = connections_.erase(conn->name());
96     assert(n == 1); (void)n;
97 + EventLoop* ioLoop = conn->getLoop();
98     ! ioLoop->queueInLoop(
99         boost::bind(&TcpConnection::connectDestroyed, conn));
100 }

```

reactor/s10/TcpServer.cc

总而言之, `TcpServer` 和 `TcpConnection` 的代码都只处理单线程的情况 (甚至都没有 `mutex` 成员), 而我们借助 `EventLoop::runInLoop()` 并引入 `EventLoopThreadPool` 让多线程 `TcpServer` 的实现易如反掌。注意 `ioLoop` 和 `loop_` 间的线程切换都发生在连接建立和断开的时刻, 不影响正常业务的性能。

`muduo` 目前采用最简单的 round-robin 算法来选取 `pool` 中的 `EventLoop`, 不允许 `TcpConnection` 在运行中更换 `EventLoop`, 这对长连接和短连接服务都是适用的, 不易造成偏载。`muduo` 目前的设计是每个 `TcpServer` 有自己的 `EventLoopThreadPool`, 多个 `TcpServer` 之间不共享 `EventLoopThreadPool`。将来如果有必要, 也可以多个 `TcpServer` 共享 `EventLoopThreadPool`, 比方说一个服务有多个等价的 TCP 端口, 每个 `TcpServer` 负责一个端口, 而来自这些端口的连接(s)共享一个 `EventLoopThreadPool`。

另外一种可能的用法是一个 `EventLoop` `aLoop` 供两个 `TcpServer` 使用 (`a` 和 `b`)。其中 `a` 是单线程服务, `aLoop` 既要 `accept(2)` 连接也要执行 IO; 而 `b` 是多线程服务, 有自己的 `EventLoopThreadPool`, 只用 `aLoop` 来 `accept(2)` 连接。`aLoop` 上还可以运行几个 `TcpClient`。这些搭配都是可行的, 这也正是 `EventLoop` 的灵活性所在, 可以根据需要在多个线程间调配负载。

本节更新了 `test8~test11`, 均支持多线程。

8.11 Connector

主动发起连接比被动接受连接要复杂一些，一方面是错误处理麻烦，另一方面是要考虑重试。在非阻塞网络编程中，发起连接的基本方式是调用 `connect(2)`，当 `socket` 变得可写时表明连接建立完毕。当然这其中要处理各种类型的错误，因此我们把它封装为 `Connector class`。接口如下：

```

25 class Connector : boost::noncopyable                                reactor/s11/Connector.h
26 {
27     public:
28         typedef boost::function<void (int sockfd)> NewConnectionCallback;
29
30         Connector(EventLoop* loop, const InetAddress& serverAddr);
31         ~Connector();
32
33         void setNewConnectionCallback(const NewConnectionCallback& cb)
34         { newConnectionCallback_ = cb; }
35
36         void start(); // can be called in any thread
37         void restart(); // must be called in loop thread
38         void stop(); // can be called in any thread

```

`Connector` 只负责建立 `socket` 连接，不负责创建 `TcpConnection`，它的 `NewConnectionCallback` 回调的参数是 `socket` 文件描述符。以下是一个简单的测试 (`s11/test12.cc`)，它会反复尝试直至成功建立连接。

```

6 muduo::EventLoop* g_loop;                                           reactor/s11/test12.cc
7
8 void connectCallback(int sockfd)
9 {
10     printf("connected.\n");
11     g_loop->quit();
12 }
13
14 int main(int argc, char* argv[])
15 {
16     muduo::EventLoop loop;
17     g_loop = &loop;
18     muduo::InetAddress addr("127.0.0.1", 9981);
19     muduo::ConnectorPtr connector(new muduo::Connector(&loop, addr));
20     connector->setNewConnectionCallback(connectCallback);
21     connector->start();
22
23     loop.loop();
24 }

```

Connector 的实现有几个难点：

- socket 是一次性的，一旦出错（比如对方拒绝连接），就无法恢复，只能关闭重来。但 Connector 是可以反复使用的，因此每次尝试连接都要使用新的 socket 文件描述符和新的 Channel 对象。要留意 Channel 对象的生命期管理，并防止 socket 文件描述符泄漏。
- 错误代码与 accept(2) 不同，EAGAIN 是真的错误，表明本机 ephemeral port 暂时用完，要关闭 socket 再延期重试。“正在连接”的返回码是 EINPROGRESS。另外，即便出现 socket 可写，也不一定意味着连接已成功建立，还需要用 getsockopt(sockfd, SOL_SOCKET, SO_ERROR, ...) 再次确认一下。
- 重试的间隔应该逐渐延长，例如 0.5s、1s、2s、4s，直至 30s，即 back-off。这会造成对象生命期管理方面的困难，如果使用 EventLoop::runAfter() 定时而 Connector 在定时器到期之前析构了怎么办？本节的做法是在 Connector 的析构函数中注销定时器。
- 要处理自连接（self-connection）。出现这种状况的原因如下。在发起连接的时候，TCP/IP 协议栈会先选择 source IP 和 source port，在没有显式调用 bind(2) 的情况下，source IP 由路由表确定，source port 由 TCP/IP 协议栈从 local port range⁸ 中选取尚未使用的 port（即 ephemeral port）。如果 destination IP 正好是本机，而 destination port 位于 local port range，且没有服务程序监听的话，ephemeral port 可能正好选中了 destination port，这就出现 (source IP, source port) = (destination IP, destination port) 的情况，即发生了自连接。处理办法是断开连接再重试，否则原本侦听 destination port 的服务进程也无法启动了。

这里就不展示 Connector class 了，读者可以带着以上疑问去阅读 muduo 源码。

练习 1：改写 s11/test12.cc，通过命令行控制它发起 N 个并发连接，可用于测试 TCP 网络服务程序的并发性。注意这个练习可能没有想象中那么简单，如果同时发起 10 000 个连接，那么某些 TCP SYN 分节可能丢包，而操作系统默认重发 SYN 的延时是 3 秒，我们无法直接控制。因此需要控制并发度，采用流水作业，尽量减少丢包。

练习 2：验证自连接出现的情况。

TimerQueue::cancel()

§8.2 实现的 TimerQueue 不能注销定时器，本节补充这一功能。TimerQueue::cancel() 的一种简单实现是用 shared_ptr 来管理 Timer 对象，再将 TimerId 定义为

⁸ sysctl 中的 net.ipv4.ip_local_port_range，以及 /proc/sys/net/ipv4/ip_local_port_range。

`weak_ptr<Timer>`，这样几乎不用我们做什么事情。在 C++11 中应该也足够高效，因为 `shared_ptr` 具备移动语义，可以做到引用计数值始终不变，没有原子操作的开销。但用 `shared_ptr` 来管理 `Timer` 对象似乎显得有点小题大做，而且这种做法也有一个小小的缺点，如果用户一直持有 `TimerId`，会造成引用计数所占的内存无法释放，而本节展示的做法不会有这个问题。

本节采用更传统的方式，保持现有的设计，让 `TimerId` 包含 `Timer*`。但这是不够的，因为无法区分地址相同的先后两个 `Timer` 对象。因此每个 `Timer` 对象有一个全局递增的序列号 `int64_t sequence_`（用原子计数器（`AtomicInt64`）生成），`TimerId` 同时保存 `Timer*` 和 `sequence_`，这样 `TimerQueue::cancel()` 就能根据 `TimerId` 找到需要注销的 `Timer` 对象。

```

20  ///
21  /// Internal class for timer event.
22  ///
23  class Timer : boost::noncopyable
24  {
25  public:
26      Timer(const TimerCallback& cb, Timestamp when, double interval)
27          : callback_(cb),
28            expiration_(when),
29            interval_(interval),
30            repeat_(interval > 0.0),
31      +    sequence_(s_numCreated_.incrementAndGet())
32      {
33      }
34
35  private:
36      const TimerCallback callback_;
37      Timestamp expiration_;
38      const double interval_;
39      const bool repeat_;
40      +    const int64_t sequence_;
41      +
42      +    static AtomicInt64 s_numCreated_;
43  };

```

reactor/s11/Timer.h

`TimerQueue` 新增了 `cancel()` 接口函数，这个函数是线程安全的。

```

32  class TimerQueue : boost::noncopyable
33  {
34  public:
35
36      +    void cancel(TimerId timerId);
37
38  };

```

reactor/s11/TimerQueue.h

```

49 private:
50
51 // FIXME: use unique_ptr<Timer> instead of raw pointers.
52 typedef std::pair<Timestamp, Timer*> Entry;
53 typedef std::set<Entry> TimerList;
54 + typedef std::pair<Timer*, int64_t> ActiveTimer;
55 + typedef std::set<ActiveTimer> ActiveTimerSet;
56
57 void addTimerInLoop(Timer* timer);
58 + void cancelInLoop(TimerId timerId);

```

cancel() 有对应的 cancelInLoop() 函数，因此 TimerQueue 不必用锁。TimerQueue 新增了几个数据成员，activeTimers_ 保存的是目前有效的 Timer 的指针，并满足 invariant: timers_.size() == activeTimers_.size()，因为这两个容器保存的是相同的数据，只不过 timers_ 是按到期时间排序，activeTimers_ 是按对象地址排序。

```

70 // Timer list sorted by expiration
71 TimerList timers_;
72 +
73 + // for cancel()
74 + bool callingExpiredTimers_; /* atomic */
75 + ActiveTimerSet activeTimers_;
76 + ActiveTimerSet cancelingTimers_;
77 };

```

reactor/s11/TimerQueue.h

由于 TimerId 不负责 Timer 的生命期，其中保存的 Timer* 可能失效，因此不能直接 dereference，只有在 activeTimers_ 中找到了 Timer 时才能提领。注销定时器的流程如下，照例用 EventLoop::runInLoop() 将调用转发到 IO 线程：

```

119 void TimerQueue::cancel(TimerId timerId)
120 {
121     loop_>runInLoop(
122         boost::bind(&TimerQueue::cancelInLoop, this, timerId));
123 }

136 void TimerQueue::cancelInLoop(TimerId timerId)
137 {
138     loop_>assertInLoopThread();
139     assert(timers_.size() == activeTimers_.size());
140     ActiveTimer timer(timerId.timer_, timerId.sequence_);
141     ActiveTimerSet::iterator it = activeTimers_.find(timer);
142     if (it != activeTimers_.end())
143     {
144         size_t n = timers_.erase(Entry(it->first->expiration(), it->first));
145         assert(n == 1); (void)n;
146         delete it->first; // FIXME: no delete please
147         activeTimers_.erase(it);
148     }

```

```

149     else if (callingExpiredTimers_)
150     {
151         cancelingTimers_.insert(timer);
152     }
153     assert(timers_.size() == activeTimers_.size());
154 }

```

reactor/s11/TimerQueue.cc

上面这段代码中的 `cancelingTimers_` 和 `callingExpiredTimers_` 是为了应对“自注销”这种情况，即在定时器回调中注销当前定时器：

```

8  muduo::EventLoop* g_loop;
9  muduo::TimerId toCancel;
10
11 void cancelSelf()
12 {
13     print("cancelSelf()");
14     g_loop->cancel(toCancel);
15 }
16
17 int main()
18 {
19     muduo::EventLoop loop;
20     g_loop = &loop;
21
22     toCancel = loop.runEvery(5, cancelSelf);
23     loop.loop();
24 }

```

s11/test4.cc

s11/test4.cc

当运行到 L14 的时候，`toCancel` 代表的 Timer 已经不在 `timers_` 和 `activeTimers_` 这两个容器中，而是位于 L162 的 `expired` 中（见 p. 292 的 `getExpired()` 实现）。

```

156 void TimerQueue::handleRead()
157 {
158     loop_->assertInLoopThread();
159     Timestamp now(Timestamp::now());
160     readTimerfd(timerfd_, now);
161
162     std::vector<Entry> expired = getExpired(now);
163
164 + callingExpiredTimers_ = true;
165 + cancelingTimers_.clear();
166     // safe to callback outside critical section
167     for (std::vector<Entry>::iterator it = expired.begin();
168         it != expired.end(); ++it)
169     {
170         it->second->run();
171     }
172 + callingExpiredTimers_ = false;
173 }

```

reactor/s11/TimerQueue.cc

```

174     reset(expired, now);
175 }

```

为了应对这种情况, TimerQueue 会记住在本次调用到期 Timer 期间有哪些 cancel() 请求, 并且不再把已 cancel() 的 Timer 添加回 timers_ 和 activeTimers_ 当中。

```

198 void TimerQueue::reset(const std::vector<Entry>& expired, Timestamp now)
199 {
200     Timestamp nextExpire;
201     for (std::vector<Entry>::const_iterator it = expired.begin();
202          it != expired.end(); ++it)
203     {
204         + ActiveTimer timer(it->second, it->second->sequence());
205         ! if (it->second->repeat()
206         +     && cancelingTimers_.find(timer) == cancelingTimers_.end())
207         {
208             it->second->restart(now);
209             insert(it->second);
210         }
211         else
212         {
213             // FIXME move to a free list
214             delete it->second;
215         }
216     }
217 }

```

reactor/s11/TimerQueue.cc

注意 TimerQueue 在执行 L170 时没有检查 Timer 是否已撤销, 这是因为 TimerQueue::cancel() 并不提供 strong guarantee。TimerQueue::getExpired() 和 TimerQueue::insert() 均增加了与 activeTimers_ 有关的处理, 此处从略。

8.12 TcpClient

有了 Connector, TcpClient 就不难实现了, 它的代码与 TcpServer 甚至有几分相似 (都有 newConnection 和 removeConnection() 这两个成员函数), 只不过每个 TcpClient 只管理一个 TcpConnection。代码从略, 此处谈几个要点:

- TcpClient 具备 TcpConnection 断开之后重新连接的功能, 加上 Connector 具备反复尝试连接的功能, 因此客户端和服务端的启动顺序无关紧要。可以先启动客户端, 一旦服务端启动, 半分钟之内即可恢复连接 (由 Connector::kMaxRetryDelayMs 常数控制); 在客户端运行期间服务端可以重启, 客户端也会自动重连。

- 连接断开后初次重试的延迟应该有随机性，比方说服务端崩溃，它所有的客户连接同时断开，然后 0.5s 之后同时再次发起连接，这样既可能造成 SYN 丢包，也可能给服务端带来短期大负载，影响其服务质量。因此每个 TcpClient 应该等待一段随机的时间（0.5~2s），再重试，避免拥塞。
- 发起连接的时候如果发生 TCP SYN 丢包，那么系统默认的重试间隔是 3s，这期间不会返回错误码，而且这个间隔似乎不容易修改。如果需要缩短间隔，可以再用一个定时器，在 0.5s 或 1s 之后发起另一次连接⁹。如果有需求的话，这个功能可以做到 Connector 中。
- 目前本节实现的 TcpClient 没有充分测试动态增减的情况，也就是说没有充分测试 TcpClient 的生命期比 EventLoop 短的情况，特别是没有充分测试 TcpClient 在连接建立期间析构的情况。编写这方面的单元测试多半要用到 §12.4 介绍的技术。

注意目前 muduo 0.8.0 采用 shared_ptr 来管理 Connector，因为在编写这部分代码的时候 TimerQueue 尚不支持 cancel() 操作。将来 muduo 1.0 会在充分测试的前提下改用这里展示的简洁的实现。

8.13 epoll

epoll(4) 是 Linux 独有的高效的 IO multiplexing 机制，它与 poll(2) 的不同之处主要在于 poll(2) 每次返回整个文件描述符数组，用户代码需要遍历数组以找到哪些文件描述符上有 IO 事件（见 p. 285 的 Poller::fillActiveChannels()），而 epoll_wait(2) 返回的是活动 fd 的列表，需要遍历的数组通常会小得多。在并发连接数较大而活动连接比例不高时，epoll(4) 比 poll(2) 更高效。

本节我们把 epoll(4) 封装为 EPoller class，它与 §8.1.2 的 Poller class 具有完全相同的接口。muduo 实际的做法是定义 Poller 基类并提供两份实现 PollPoller 和 EPollPoller。这里为了简单起见，我们直接修改 EventLoop，只需把代码中的 Poller 替换为 EPoller。

EPoller 的关键数据结构如下，其中 events_ 不是保存所有关注的 fd 列表，而是一次 epoll_wait(2) 调用返回的活动 fd 列表，它的大小是自适应的。

```
typedef std::vector<struct epoll_event> EventList;  
typedef std::map<int, Channel*> ChannelMap;
```

⁹ <http://bitsup.blogspot.com/2010/12/accelerated-connection-retry-for-http.html>

```
int epollfd_;           // ::epoll_create()
EventList events_;
ChannelMap channels_;
```

struct epoll_event 的定义如下，注意 epoll_data 是个 union，muduo 使用的是其 ptr 成员，用于存放 Channel*，这样可以减少一步 look up。

```
typedef union epoll_data
{
    void    *ptr;
    int      fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

struct epoll_event
{
    uint32_t      events; /* Epoll events */
    epoll_data_t data;    /* User data variable */
};
```

为了减少转换，muduo Channel 没有自己定义 IO 事件的常量，而是直接使用 poll(2) 的定义（POLLIN、POLLOUT 等等），在 Linux 中它们和 epoll(4) 的常量相等。

```
23 // On Linux, the constants of poll(2) and epoll(4)
24 // are expected to be the same.
25 BOOST_STATIC_ASSERT(Epollin == POLLIN);
26 BOOST_STATIC_ASSERT(Epollpri == POLLPRI);
27 BOOST_STATIC_ASSERT(Epollout == POLLOUT);
28 BOOST_STATIC_ASSERT(Epollrdhup == POLLRDHUP);
29 BOOST_STATIC_ASSERT(Epollerr == POLLERR);
30 BOOST_STATIC_ASSERT(Epollhup == POLLHUP);
```

reactor/s13/EPoller.cc

EPoller::poll() 的关键代码如下。L58 在 C++11 中可写为 events_.data()。L68 表示如果当前活动 fd 的数目填满了 events_，那么下次就尝试接收更多的活动 fd。events_ 的初始长度是 16（kInitEventListSize），其会根据程序的 IO 繁忙程度自动增长，但目前不会自动收缩。

```
55 Timestamp EPoller::poll(int timeoutMs, ChannelList* activeChannels)
56 {
57     int numEvents = ::epoll_wait(epollfd_,
58                                 &events_.begin(),
59                                 static_cast<int>(events_.size()),
60                                 timeoutMs);
61     Timestamp now(Timestamp::now());
```

reactor/s13/EPoller.cc

```

62     if (numEvents > 0)
63     {
64         LOG_TRACE << numEvents << " events happended";
65         fillActiveChannels(numEvents, activeChannels);
66         if (implicit_cast<size_t>(numEvents) == events_.size())
67         {
68             events_.resize(events_.size()*2);
69         }
70     }

```

此处 `epoll_wait(2)` 的错误处理从略。

```

79     return now;
80 }

```

reactor/s13/EPoller.cc

`EPoller::fillActiveChannels()` 的功能是将 `events_` 中的活动 fd 填入 `activeChannels`, 其中 L90~L93 是在检查 invariant。

```

82 void EPoller::fillActiveChannels(int numEvents,
83                                 ChannelList* activeChannels) const
84 {
85     assert(implicit_cast<size_t>(numEvents) <= events_.size());
86     for (int i = 0; i < numEvents; ++i)
87     {
88         Channel* channel = static_cast<Channel*>(events_[i].data.ptr);
89 #ifndef NDEBUG
90         int fd = channel->fd();
91         ChannelMap::const_iterator it = channels_.find(fd);
92         assert(it != channels_.end());
93         assert(it->second == channel);
94 #endif
95         channel->set_revents(events_[i].events);
96         activeChannels->push_back(channel);
97     }
98 }

```

reactor/s13/EPoller.cc

`updateChannel()` 和 `removeChannel()` 的代码从略。因为 `epoll` 是有状态的, 因此这两个函数要时刻维护内核中的 fd 状态与应用程序的状态相符, `Channel::index()` 和 `Channel::set_index()` 被挪用为标记此 `Channel` 是否位于 `epoll` 的关注列表之中。这两个函数的复杂度是 $O(\log N)$, 因为 Linux 内核用红黑树来管理 `epoll` 关注的文件描述符清单。

测试程序无须修改, 全都已经自动用上了 `epoll(4)`。

至此, 一个基于事件的非阻塞 TCP 网络库已经初具规模。

8.14 测试程序一览

本章简要介绍了 muduo 的实现过程，是一个具有教学示范意义的项目，希望有助于读者理解 one loop per thread 这一编程模型背后的实现，在运用时更加得心应手。如果对本章代码有疑问，应该以最新版的 muduo 源码为准。

本节没有配套代码，以下列出前面各节出现的测试代码的功能。

- §8.0 s00/test1.cc 在两个线程里各自运行一个 EventLoop。
- §8.0 s00/test2.cc 试图在非 IO 线程调用 EventLoop::loop()，程序崩溃。
- §8.1 s01/test3.cc 用 Channel 关注 timerfd 的可读事件。
- §8.2 s02/test4.cc TimerQueue 示例。
- §8.3 s03/test5.cc IO 线程调用 EventLoop::runInLoop() 和 EventLoop::runAfter()。
- §8.3 s03/test6.cc 跨线程调用 EventLoop::runInLoop() 和 EventLoop::runAfter()。
- §8.4 s04/test7.cc Acceptor 示例。
- §8.5 s05/test8.cc discard 服务。
- §8.8 s08/test9.cc echo 服务。
- §8.8 s08/test10.cc 发送两次数据，测试 TcpConnection::send()。
- §8.9 s09/test11.cc chargen 服务，使用 WriteCompleteCallback。
- §8.11 s11/test12.cc Connector 示例。
- §8.12 s12/test13.cc TcpClient 示例。

本章 Acceptor、Connector、Reactor 等术语是 Douglas Schmidt 发明的，他的原始论文出处是

- <http://www.cs.wustl.edu/~schmidt/PDF/Reactor1-93.pdf>
- <http://www.cs.wustl.edu/~schmidt/PDF/Reactor2-93.pdf>
- <http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf>
- <http://www.cs.wustl.edu/~schmidt/PDF/reactor-rules.pdf>
- <http://www.cs.wustl.edu/~schmidt/PDF/Acceptor.pdf>
- <http://www.cs.wustl.edu/~schmidt/PDF/Connector.pdf>
- <http://www.cs.wustl.edu/~schmidt/PDF/Acc-Con.pdf>

我在一篇访谈¹⁰中谈到了 muduo 将来的计划：1.0 版完善单元测试，基本覆盖各种 code path，特别是各种 Sockets API 出错情况的测试，以及用户调用与 IO 事件的交互。2.0 版启用 C++11，特别是 rvalue reference 有助于提高性能与资源管理的便利性。以上计划中的版本尚无明确的时间表。

¹⁰ http://www.oschina.net/question/28_61182

第 3 部分

工程实践经验谈

第 9 章

分布式系统工程实践

本章谈的分布式系统是指运行在公司防火墙以内的信息基础设施 (infrastructure), 用于对外 (客户) 提供联机信息服务, 不是针对公司员工的办公自动化系统。服务器的硬件平台是多核 Intel x86-64 处理器、几十 GB 内存、千兆网互联、常规存储、运行 Linux 操作系统。系统的规模大约在几十台到几百台, 可以位于一个机房, 也可以位于全球的多个数据中心。只有两台机器的双机容错 (热备) 系统不是本章的讨论范围。服务程序是普通的 Linux 用户进程, 进程之间通过 TCP/IP 通信。特别是, 本章不考虑分布式存储系统, 只考虑分布式即时计算。

本章不谈 “企业级开发”, 也就是以商用数据库为存储, 使用商用消息中间件 (MQ) 或交易中间件 (Tuxedo), 故障转移切换 (failover) 用 VCS 等商业解决方案。也不谈 “高性能计算 (HPC)”, 这是一个相对成熟的领域, 通常以 MPI 为编程平台。并行算法对延迟有苛刻要求, 通常采用 InfiniBand 为通信方式, 不是常规的基于以太网的 TCP/IP 互联。

先谈钱 每台机器的购买成本是几万元人民币, 每年的使用成本以一万元计 (电费、机位、空调、网管, 不含对外带宽)。换言之, 本章讨论的是运行在几十台或几百台 PC 服务器 (每台价值几万元) 上的分布式系统, 不是运行在几台高端服务器 (每台价值几十万乃至上百万元) 上的系统。换言之, 是 Google、Facebook、Amazon 那种风格的分布式系统, 不是 IBM、Oracle、HP 的 scale up 系统。

在这种用 commodity 硬件 (服务器和网络) 搭建的分布式系统中, 扩容方式主要通过增加机器 (scale out) 进行。理想情况下, 系统架构应该具备线性的伸缩性, 系统实现应该让 “伸缩” 具有较小的比例系数。不妨假定同一批购买的相同用途的机器具有相同的配置, 每次采购总是购买性价比最高的机型。服务器的服役期一般不超过 5 年, 因为一台 5 年前购买的旧机器在消耗相同的电能的情况下, 提供的处理能力只有新机器的一半甚至更少, 不如淘汰它再买新机器更划算。

网络方面，可以认为同一数据中心的任何两台机器之间有千兆带宽，常用的做法是采用 Clos/Fat-tree 网络拓扑¹。TCP/IP 协议原本是为广域网设计的，但数据中心里的网络特性与传统广域网不同²，会出现 TCP Incast 症状^{3 4}。

也就是说，本章讨论在均质（homogeneous）的硬件和网络情况下设计系统。

具体考虑以下有代表性的两种情况⁵，假设每台机器每年的固定支出是 1 万元，再加上购买成本：

- 一台低端的价值 2 万元的服务器，使用寿命 4 年，平均每年支出 1.5 万元。
- 一台中端的价值 5 万元的服务器，预期服役 3 年，平均每年支出 2.7 万元。

为了便于计算，假定一台服务器一年的使用成本为 3 万元，来做一个非常粗略的估算：

- 一个普通程序员，公司每月支出 2 万元⁶，一年 24 万，相当于 8 台服务器。
- 一个高级程序员，公司每月支出 3 万元，一年 36 万，相当于 12 台服务器。

一个高级程序员花 3 个月时间，把系统性能提高了 20%，公司的成本是 9 万元，大约相当于 3 台中端服务器一年的使用费。如果原来有 5 台服务器，性能提高 20% 就意味着能节约 1 台服务器，这不见得划算。如果有 50 台服务器，节约了 10 台，有可能划得来。如果有 500 台服务器，节约了 100 台，肯定划得来。可见在需要提高系统处理能力的时候，优化代码不见得是首要的，有时候买新机器更划算。

硬件与操作系统 这种几万元级别的 x86 服务器的一般配置是：

- 双路多核 CPU，一共 8~16 核（不含超线程）。
- 几十 GB ECC 内存。
- 几块硬盘⁷，容量几百 GB 至几 TB。
- 冗余电源。
- 千兆网卡（GbE）或万兆网卡（10GbE）。

这种级别的硬件的可靠性见 §9.2，但是可以想见，不能指望单机具有坚不可摧的可靠性。分布式系统的可靠性不能依赖“硬件不会停机”这一假设。

¹ <http://dl.acm.org/citation.cfm?id=1402967>

² <http://www.scs.stanford.edu/11au-cs144/notes/l17.pdf>

³ <http://www.snookles.com/slf-blog/2012/01/05/tcp-incast-what-is-it/>

⁴ <http://www.pdl.cmu.edu/Incast/>

⁵ 这是非常业余的估算，只考虑了机器本身，没有考虑网络设备、制冷等方面：进一步的内容可以参考 Google 工程师写的《The Datacenter as a Computer》[DCC]。

⁶ 注意这不是员工的税前工资，它包括了公司的其他用人成本，如场地租金、办公设施、五险一金等。

⁷ 本书不涉及存储，也就不仔细区分 SSD、SAS、SATA 的不同。

这种服务器运行的通常是免费的 Linux 发行版。为什么不用 Windows 或其他商业系统？假设有 200 万元服务器硬件投资，可以买 100 台 2 万元的服务器。如果用 Windows Server 标准版，每台机器增加 3000 元成本⁸，只能买 87 台服务器。Linux 方案的硬件 raw 处理能力比其高 15%。现在我们面临的不是 Windows 与 Linux 谁快的问题，而是 Windows 能否比 Linux 快 15% 以上，让投资回报合理。

在价值几万元这个级别的服务器上，我认为 Windows 比 Linux 快是不成立的。本书讨论的分布式系统对操作系统的功能需求是：

- 管理十几个核上的任务调度。
- 管理几十 GB 物理内存的分配释放。
- 驱动一两个千兆或万兆网卡。
- 驱动十来块普通服务器级的硬盘。

Linux 内核可以很好地完成以上这些任务，我不认为其他操作系统能把这几样普通硬件管得更好。或许在高端 128 核 1TB 内存的安腾服务器上 Windows 表现更佳，但这就不是本书讨论的范围了。既然操作系统选定为 Linux，那自然不必考虑跨平台的问题，程序开发工作因此也简化了许多。

做分布式系统一个有意思的现象：公司越大，技术能力越强，用的机器越便宜。一般的公司会购买品牌服务器，配备冗余的电源和网卡，硬盘通常是配置为 RAID 5/6/10 等阵列，商用 SAN 存储也不少见。技术领先的互联网公司为了压缩成本，往往采用单电源、单网卡，存储也用一两块普通 SATA 硬盘（并且不用 RAID），但无论如何，使用的还是服务器级的多路 CPU 和 ECC 内存^{9 10}。有的公司甚至用 Intel Atom 或 ARM 来替换 Xeon 服务器，以进一步降低能耗，但是由于可靠性较低（内存无校验），这些低端“服务器”通常用于静态 cache 之类的场合。

9.1 我们在技术浪潮中的位置

单机服务端编程问题已经基本解决 编写高吞吐、高并发、高性能的服务端程序的技术已经成熟。无论是程序设计还是性能调优，都有成熟的办法。在分布式系统中，单机表现出来就是一个网口（§9.7.3），能收发消息，至于它内部用什么语言什么编程模型都是次要的。在满足性能要求的前提下，应该用尽量简单直接的编程方式。

⁸ 零售价是 5000 元，这里是我估计的批发价。

⁹ http://news.cnet.com/8301-1001_3-10209580-92.html

¹⁰ <http://www.datacenterknowledge.com/archives/2012/06/27/video-facebook-compute-unit/>

单机的技术热点不在于提高性能，而在于解放程序员的生产力，例如牺牲少许性能，用更易于开发的语言。

在编程模型方面，分布式对象已被淘汰。准确地说是远程对象¹¹，对象位于另一个进程（可能运行在另一台机器上），程序就像操作本地对象一样通过成员函数调用来使用远程服务。这种模型的本质难点在于容错语义。假设对象所在的机器坏了怎么办？已经发起但尚未返回的调用到底有没有成功？调用远程对象的 `method` 应该是阻塞还是抛异常呢？假设持有对象引用的机器崩溃怎么办？对象有机会被回收吗？你理解并信得过它内置的容错与对象迁移机制吗？

20 世纪 80 年代提出这种编程模型的前提是服务器的可靠性极高，有相当强的容错能力，几乎不存在失效的可能。这一前提在目前的分布式系统开发中是不成立的，这种技术适合所谓的企业级开发，不适合面向业务的分布式系统。这里推荐一篇 Google 的好文《Introduction to Distributed System Design》¹²。其中的点睛之笔是：分布式系统设计，是 `design for failure`。设计分布式系统不能基于错误的假设¹³。

大规模分布式系统处于技术浪潮的前期。大家都在摸索中前进，尚未形成一套完整的方法论。某些领域相对成熟一些（分布式非结构化存储、离线数据处理等），有一些开源的组件。但更多更本质的问题（正确性、可靠性、可用性、容错性、一致性）尚没有一套行之有效的方法论来指导实践，有的只是一些相对零散的经验¹⁴。有人开玩笑说：“我不知道哪种方法一定能行，但是知道哪些方法是行不通的。”这或许正是我们这一阶段的真实写照，分布式系统开发还处于“摸着石头过河”阶段。

市面上分布式系统方面的书籍，大多谈的是高性能科学计算、并行算法，或者某些分布式算法的学术问题¹⁵，这些书对面向业务的分布式系统的指导意义有限。从另一个方面讲，面试一个“分布式系统的职位”都没有公认的好的面试题¹⁶，往往只能从项目经历来考察应聘者的水平。

我们怎么办？勿在浮沙筑高台，只用成熟的基础设施。目前看来，Linux、多线程编程、TCP/IP 网络编程¹⁷是成熟的，我认为没有哪个“C++ 分布式中间件”是成

¹¹ 包括 CORBA、DCOM、RMI、EJB、.NET Remoting 等等。

¹² <http://code.google.com/edu/parallel/dsd-tutorial.html>

¹³ 《Fallacies of Distributed Computing Explained》(<http://www.rgoarchitects.com/Files/fallacies.pdf>)

¹⁴ 别人分享的经验也不一定完全可信，因为他可能分享的是一个阶段性成果。他告诉你故事的开头，不一定也告诉你故事的结局。参见 Amazon 的 Dynamo 存储架构和采用这一思想的 Cassandra 项目的兴衰。

¹⁵ 分布式锁、选举等，见 http://en.wikipedia.org/wiki/Distributed_algorithm。

¹⁶ 如果要考察 C++ 程序员，虚析构是必问的；如果要考察 C# 程序员，`value type` 与 `reference type` 是必问的；如果要考察多线程程序员，死锁与 `race condition` 是必问的。考察分布式程序员呢？

¹⁷ 而且只使用最基本的 `read/write` 数据流，不使用 `out-of-band`，也不使用 SCTP 等不同寻常的协议。

熟的¹⁸。2000年，Linux和多线程编程都不成熟；2004年Linux 2.6内核支持epoll和NTPL，Linux服务端多线程编程基本成熟。1990年，TCP/IP网络编程不成熟；W. Richard Stevens的传世经典《TCP/IP详解》和《UNIX网络编程（第2版）》分别在1993和1998年出版，网络编程基本成熟。现在，如果要学习Linux性能调优、多线程编程、网络编程、TCP/IP协议等等知识，都能找到非常好的书籍和网上资源，“能够靠读书、看文章、读代码、做练习学会的东西没什么门槛”¹⁹。

这也是本书主讲Linux多线程TCP网络编程的重要原因。但是这距离设计分布式系统还有巨大的鸿沟，本章的一些个人经验或许能让读者稍微少走一些弯路。

9.1.1 分布式系统的本质困难

Jim Waldo等人写的《A Note on Distributed Computing》²⁰一针见血地指出分布式系统的本质困难在于partial failure。

拿我们熟悉的单机和分布式做个对比，初看起来，分布式系统很像是放大的单机。一台机器通过总线把CPU、内存、扩展卡（网卡和磁盘控制器）连到一起²¹，一个分布式系统通过网络把服务进程连到一起，网络就是总线。这种看法对吗？单机和分布式的区别究竟在哪里？能不能按照编写单机程序的思路来设计分布式系统？

分布式系统不是放大的单机系统，根本原因在于单机没有部分故障（partial failure）一说。对于单机，我们能轻易判断某个进程、某个硬件是否还在正常工作。而在分布式系统中，这是无解的，我们无法及时得知另外一台机器的死活，也无法把机器崩溃与网络故障区分开来²²。这正是分布式系统与单机的最大区别。

例如一次RPC调用超时，调用方无法区分

- 是网络故障还是对方机器崩溃？
- 软件还是硬件错误？
- 是去的路上出错还是回来的路上出错？
- 对方有没有收到请求，能不能重试？

¹⁸ 对任何宣称“像开发单机程序一样写分布式应用”的广告语保持警惕，分布式系统有其本质困难。

¹⁹ 孟岩的《技术路线的选择重要但不具有决定性》（<http://blog.csdn.net/myan/article/details/3247071>）。

²⁰ http://labs.oracle.com/techrep/1994/sml_i_tr-94-29.pdf：这篇文章同时指出延迟和内存访问也是重要区别。

²¹ 现在的CPU往往内置了内存控制器，不通过总线直接访问内存，但不影响此处的讨论。

²² 《A Note on Distributed Computing》中说道：“[T]here is no common agent that is able to determine what component has failed and inform the other components of that failure, no global state that can be examined that allows determination of exactly what error has occurred. In a distributed system, the failure of a network link is indistinguishable from the failure of a processor on the other side of that link.”

在本机调用成员函数根本不会出现这种情况²³。这不是 RPC 的过错，而是分布式系统固有的特点，此处把 RPC 换成网络消息的请求响应也是一样的。简单地说，单机的编程经验不能直接套用在分布式系统上，分布式系统需要用单独的理论来分析²⁴。

单机（集中式）与分布式的根本区别在于进程的地址空间（address space）是一个还是多个，对于分布式系统来说，如果把进程比喻成“人”（§3.1），那么这些“人”不是在一个屋子里交谈，而是通过电话会议交谈。或者类比成一群盲人在屋子里交谈。重要的区别在于，通过电话会议交谈的时候只能听到别人的发言，如果有人离场，其他人不会立刻得知，通常只能通过“一段时间没有发言”或者“叫他的名字没有回答”来间接判断某人已经离场。但是一个人被其他事情吸引（短暂过载）或开小差（网络暂时故障）也会表现为“一段时间没有发言”或者“叫他的名字没有回答”，其他人无法区分这两种情况。换言之，进程间通过收发消息来交换信息，一个进程看不到别的进程的数据，也不能立刻判断别的进程的死活²⁵。当然，这个比喻本身也有问题，它假设了同时性和事件顺序的确定性。一个人说的话会立刻被其他人听到，甲乙两个人先后说话，那么其他人听到的顺序都是先甲后乙。这在分布式系统中是不成立的，见 p. 348 的例子。

分布式系统设计以进程为基本单位，先确定有哪些功能，需要做几个程序，每个程序的职责和它掌握的数据。然后安排这些程序在多台机器上的分布，规划每个程序起几个进程。进程之间的传输协议很容易确定，使用 TCP 长连接即可（§3.4）。比较费脑筋的是进程之间的通信协议，即发送哪些消息，每条消息包含哪些内容。随着系统的演化，消息的内容也会变化，因此要提前做好准备（§9.6）。

9.1.2 分布式系统是个险恶的问题

险恶的问题（wicked problem）²⁶的意思是：你必须首先把这个问题“解决”一遍，以便能够明确地定义它，然后再解决一遍。在实现一个系统之前，很可能无法预料哪个技术方案行得通。这里举两个虚构的例子说明其险恶。

²³ Ken Arnold 说道：“Now this is not a question you ask in local programming. You invoke a method and an object. You don't ask, “Did it get there?” The question doesn't make any sense. But it is *the* question of distributed computing.”（<http://www.artima.com/intv/distrib.html>）。

²⁴ 类似电路理论里的集总参数电路和分布参数电路，前者适用基尔霍夫定律，后者适用传输线理论。

²⁵ 《Introduction to Distributed System Design》中讲：“We are forced to deal with uncertainty. A process knows its own state, and it knows what state other processes were in recently. But the processes have no way of knowing each other's current state. They lack the equivalent of shared memory. They also lack accurate ways to detect failure, or to distinguish a local software/hardware failure from a communication failure.”（<http://code.google.com/edu/parallel/dsd-tutorial.html>）。

²⁶ 《代码大全（第2版）》[CC2e]第5.1节：只有通过解决或部分解决才能被明确定义的问题。

假设有一个缩略图（Thumbnailer）服务，它的功能是将用户提供的数码照片按比例缩小为固定尺寸，这是一个典型的无状态服务。它的实现很简单，不过是给 ImageMagick 的 `convert(1)` 命令提供一层网络封装。计算缩略图是一项相当耗时的任务²⁷，平均每张图片用时 0.5s，一台 8 核服务器每秒只能处理 16 张图片。相比之下，一台 8 核 Web 服务器可支撑每秒 8000 次 HTTP 请求响应，平均每个 HTTP 请求只占用 1ms CPU 时间。为了避免压缩照片影响 Web 服务器的性能，我们把生成缩略图功能移到单独的服务器中。系统中有多台 Web 服务器，连接到多台 Thumbnailer 服务器。现在的问题是，我们该如何做负载均衡？

第一个想法是每台 Web 服务器只和一台 Thumbnailer 打交道，通过 Web 本身的负载均衡来让图片压缩请求均匀地分散到多个 Thumbnailer 上。如图 9-1 所示的两种做法。

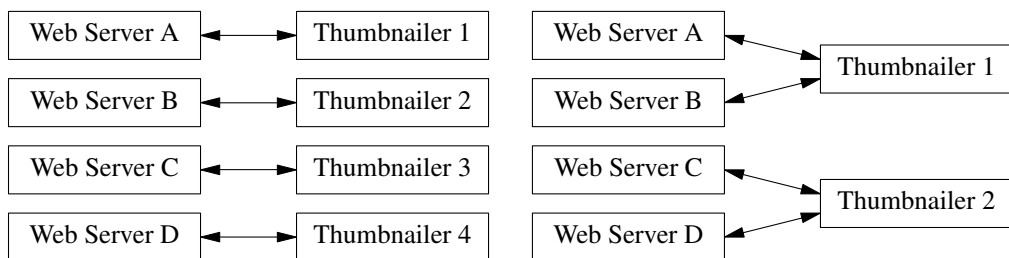


图 9-1

这种做法足够简单，但是 Web 负载从短期来看是非均匀的，具有突发性（burst）。假如某个用户通过某一个 Web 服务器上传了一堆照片，那么在现在这个设计中，会有一个 Thumbnailer 满负荷，而其他 Thumbnailer 都闲着，这不利于快速响应。

自然地，我们让每个 Web 服务器都可以和每个 Thumbnailer 服务器打交道，以期充分均匀地分散某一 Web 服务器的突发负载，形成了如图 9-2 所示的连接关系。那么负载均衡又该怎么做呢？

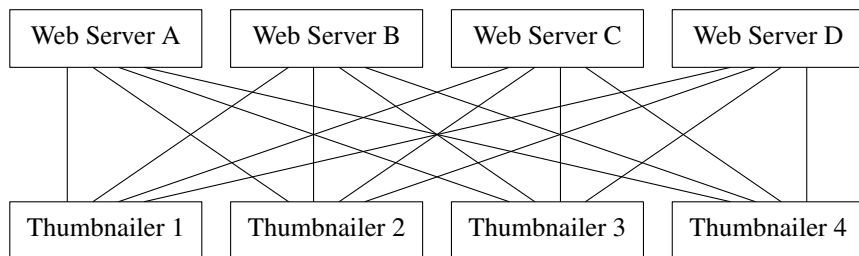


图 9-2

²⁷ 这是一个重采样（resampling）过程，为了保证质量，用的是双三次（bicubic）插值算法。

有几个实践证明不靠谱的做法：

1. 每个 Web 服务器轮流向 Thumbnailer $\{1, 2, 3, \dots, N\}$ 发送请求，结果发现 Thumbnailer 的负载像走马灯一样移动，因为 Web 服务器先集中火力攻击第 1 台 Thumbnailer，然后再集中攻击第 2 台 Thumbnailer，如此等等。
2. 既然轮流发送请求不合适，就采用随机的方式选择 Thumbnailer，随机数的种子用时间初始化。但是 Web 服务器几乎同时启动，它们用于初始化的种子相同，产生的伪随机序列也相同，造成与第 1 种做法相同的“潮涌”现象。

前面这两条都是开环控制，下面考虑闭环控制，让 Web 服务器知道 Thumbnailer 的实际负载，并从中选出负载最轻的来发送请求。

3. 让 Thumbnailer 向 Web 服务器定期汇报当前负载情况，这种做法的缺点是消息数目与服务器数目呈平方关系，有 M 台 Web 服务器， N 台 Thumbnailer 服务器，每个周期要发送 $M \times N$ 条消息，伸缩性不佳。而且“负载”强弱本身也不易定义。
4. 通过某个集中的负载均衡器（load balancer）来收集并分发负载情况，好处是把消息数目降为 $M + N$ ，但是造成了单点故障（Single Point of Failure, SPoF）。

这几个想法初看上去都挺合理，但是仔细分析却有各自的问题。这里提出一种完全基于客户端视角的负载均衡策略。

第 3 和第 4 两种方案是基于 Thumbnailer 服务的当前负载的反馈控制，每次新请求都发向当前负载最轻的服务端。那么我们遇到的一个更本质问题是，如何定义服务端的负载？或者说 Thumbnailer 如何算出自己当前负载的单值，以供客户端排序？其当前负载值与本机 CPU 使用率、内存占用率、硬盘剩余空间比例、网络带宽使用率是什么关系？各部分权重大小如何分配？有没有考虑同时运行在同一台机器上的其他服务进程也会消耗资源？

我们注意到，响应客户端（这里是 Web 服务器）请求的快慢直接反应了服务端（Thumbnailer）的负载。客户端根本无须关心服务端负载的具体情况（CPU 负载、网络带宽负载、内存使用率等），只需要看它响应自己请求的速度就可以判断应该把下一个请求发给哪个服务端。具体地说是选择活动请求（已经发出请求而尚未收到响应）数目最少的那个服务端。这样一来客户端无须定期查询各个服务端的负载，只要根据自己以往的调用情况就能做出判断。这个做法大大简化了系统的设计。

客户端把服务端看成一个循环队列，在选择服务端时，从上次调用的服务端的下一个位置开始遍历，找出负载最轻的服务端。每次遍历的起点选在上次遍历终点的下一位置，这是为了在服务端负载相等的情况下轮流使用各个服务端，使各服务端负载

大致相当。算法如下，其中 `last` 表示上次选取的服务端编号。

```
int selectLeastLoad(const vector<Endpoint>& endpoints, int* last)
{
    int N = endpoints.size();
    int start = (*last + 1) % N; // 每次从前次调用的下一位置找起

    int min_load_idx = start;    // 从 start 开始找负载最轻的服务端
    int min_load = endpoints[start].active_reqs(); // 活动请求数目

    for (int i = 0; i < N; ++i) {
        int idx = (start + i) % N;
        int load = endpoints[idx].active_reqs(); // 负载即活动请求数目

        if (load < min_load) { // 找到更小的负载
            min_load = load;
            min_load_idx = idx;
            if (min_load == 0) // 已经找到最小负载，无须再找
                break;
        }
    }

    *last = min_load_idx;
    return min_load_idx;
}
```

举例来说，有 4 台 Thumbnailer 服务器。在某一时刻，客户端 Web Server A 向这 4 台服务器已发起而尚未结束的请求（即前述“活动请求”）的数目分别为 3、2、3、4，Thumbnailer 2 负载最轻，那么这时新的图片压缩任务将发给 Thumbnailer 2。在下一次请求到来时，活动请求数目分别为 3、3、3、3，客户端从 Thumbnailer 2 的下一位置（即 Thumbnailer 3）开始查找可用的服务端，没有哪个服务端的负载比 Thumbnailer 3 更轻，因此新的图片压缩任务将发给 Thumbnailer 3。

在最初的两种反馈控制设计中（3 和 4），是站在服务器的角度评估服务器的当前负载。某个服务程序的“当前负载”是一个全局数据，由服务端产生，每个客户端都希望这个全局数据随时保持更新。而在新设计中，客户端根本不用关心这个全局数据，只要从自己的角度看，哪个服务器负载轻、等待响应的活动请求少，就把下一个请求发给哪个服务器。各个客户端看到的服务器负载情况可能不尽相同，不过从统计上看，负载仍然是均匀分配的，实验结果很好地支持了这一点。新的设计规避了分布式系统中保持全局数据一致性这个老大难问题。

在多个客户端（Web Server）的情况下，为了避免潮涌，每个 Web Server 用于初始化 `last` 值的随机数种子应该有足够的随机性，例如可以包括 IP 地址、MAC 地址、当前时间、PID 号等等。

这个简单的负载均衡策略在实际应用中获得了良好的效果。

第二个例子，分布式系统的险恶之处还在于时间与事件顺序违反直觉（具有狭义相对论效应，每个本地观察者有自己的时钟和事件顺序^{28 29}），因为消息传递的延时是不固定的。

比方说，顾客向商店订购了一件商品（0:order），商店先是确认订单已收到（a:ack），再通知仓库发货（b:ship），随后立刻通知客户货物已发出（c:confirm），最后客户收到货物（d:deliver）。消息流程如图 9-3 所示。

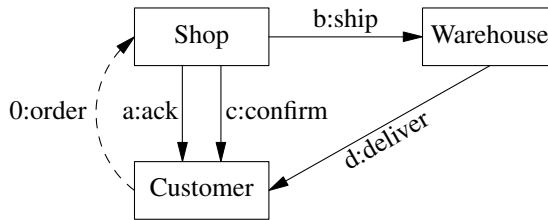


图 9-3

按照常规的想法，a、c、d 这 3 条消息发送的顺序是明确的，那么 Customer 收到消息的顺序也应该是 a、c、d。但是在分布式系统中，a、c、d 这 3 条消息到达 Customer 的顺序有 6 种可能。即便 Shop 与 Customer 采用一个 TCP 连接通信，保证 a 先于 c 到达，那么 Customer 收到这 3 条消息仍然有 3 种可能的顺序：

1. a, c, d
2. a, d, c
3. d, a, c

由于 a、c 与 d 由两个不同的 TCP 连接发送，它们之间没有确定的先后关系。

同样的道理，如果客户端往 Master 发出一个请求，Master 指定某个 Work 来完成，Worker 把计算结果发给客户端，消息流程如图 9-4 所示。

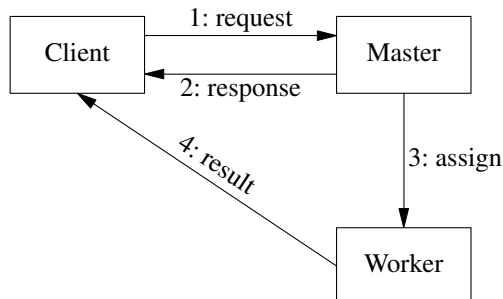


图 9-4

²⁸ <http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf>

²⁹ <http://www.stanford.edu/class/ee380/Abstracts/091111-RethinkingTime.pdf>

那么“4:result”完全有可能先于“2:response”到达客户端，因为 TCP 重传的首次间隔是 200ms，如果发送“2:response”的时候发生了重传，那么它会比没有重传的“4:result”更晚到达客户端。网络上消息传递的延迟没有上界，完全有可能出现后发先至的情况。客户端程序必须预见到并能正确应对这种乱序的情况。

另外，Client 也没有办法判断“4:result”和“2:response”的发送哪个在前、哪个在后，即便 Master 和 Worker 都在消息中加上发送时间戳（timestamp）。这是因为分布式系统中每台机器有自己的本地时钟，Master 和 Worker 的时钟之间肯定是有误差的，而 Client 并不知道它们的时间差多少。

在局域网内，消息的传输延迟不能通过发送方和接收方时间戳的差值算出来。因为在局域网中，虽然 NTP 对时的精度可以达到 1 毫秒之内，但消息的延迟本身也在 1 毫秒以内。测量值和未知系统误差³⁰在同一量级，测量结果是无意义的。必要的时候我们可以先测量两台机器之间的时间差，用来修正延迟测量的结果 (§7.9)。

9.2 分布式系统的可靠性浅说

本节谈谈我对分布式系统可靠性的理解。要谈可靠性，必须要谈基本指标 t_{MTBF} （平均无故障运行时间³¹，单位通常是小时）。 t_{MTBF} 与可靠性的关系如下，其中 t 是系统运行时间。

$$\text{Reliability} = \exp\left(-\frac{t}{t_{\text{MTBF}}}\right)$$

按照上式，当 $t = t_{\text{MTBF}}$ 时，系统的可靠度为 36.8%。也就是说当系统连续运转 t_{MTBF} 这么长时间后，发生故障的概率为 63.2%。见图 9-5 中的指数曲线。

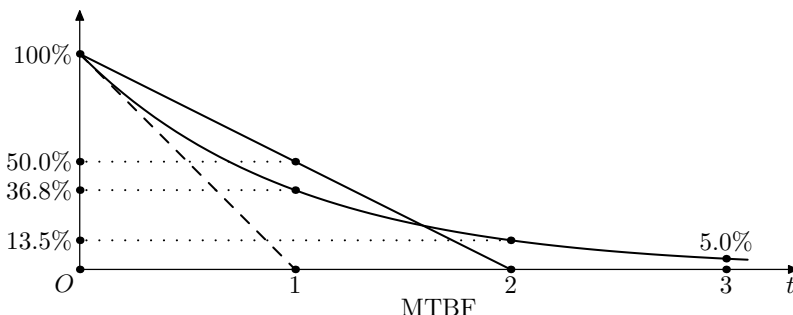


图 9-5

³⁰ http://en.wikipedia.org/wiki/Systematic_error

³¹ http://en.wikipedia.org/wiki/Reliability_engineering http://en.wikipedia.org/wiki/Mean_time_between_failures

图 9-5 的横坐标为 t_{MTBF} 的倍数, 纵坐标为可靠度。在粗略估算的时候, 可以用直线代替曲线, 达到 t_{MTBF} 时系统发生故障的概率为 50% (图 9-5 中的实线) 或 100% (图 9-5 中的虚线)。如果要粗略估算短期可靠性, 应该用图 9-5 中的虚线, 它是指数曲线在 $t = 0$ 附近的一阶近似 (斜率相等, 为 -1)。在估算可靠性的时候, 一两倍的差距无伤大雅, 因为 t_{MTBF} 本身只是平均数, 而设备损坏是非均匀的³²。

t_{MTBF} 与使用寿命无关。硬盘的寿命通常是 3~5 年, 但其标称 t_{MTBF} 是 100 万小时, 即 114 年。这两个数字并不矛盾, t_{MTBF} 为 100 万小时, 意味着如果有 1 万块硬盘同时运行, 那么平均每 200 小时会坏掉一块³³。如果按 t_{MTBF} 为 100 万小时计算, 硬盘每年的故障率不到 1%, 但是硬盘实际的年故障率在 3%~8%, 因此不能一味相信厂家给出的数据^{34 35 36}。

一个系统由多个部件组成, 系统的整体可靠性取决于部件之间是“并联”还是“串联”。所谓两个部件“并联”, 指的是两个部件同时坏掉会导致系统失灵, 比方说冗余电源就是“并联”的。所谓两个部件“串联”, 指的是只要有一个部件坏掉, 系统就失灵了; 一般的入门级服务器上, 主板、CPU、内存都是“串联”的。

“并联”可以极大地提高可靠性。例如一台服务器有两个电源, 同时工作, 而且可以热插拔。如果单个电源的 t_{MTBF} 是 10 万小时, 更换坏电源需要 24 小时。假设有 100 台单电源的机器, 平均每 42 天会有一台机器出现电源故障³⁷。如果改用双电源, 在出现坏掉一个电源的情况后, 在 24 小时内更换它, 可确保不停机。那么这台机器的另一个电源在这 24 小时内坏掉的可能性是 $1 - \exp(-24/100\,000) = 0.024\%$, 因此双电源的可靠性是 99.976%。

再来计算硬盘存储数据的可靠性³⁸。为了便于计算, 假设硬盘的年故障率是 3.65%, 更换坏硬盘并重建数据需要 24 小时。一块硬盘在 24 小时内坏掉的可能性约是 0.01% (即 10^{-4})。如果我们按 GFS 的思路把数据存 3 份, 在一块硬盘坏掉之后, 立刻用另外两份拷贝开始在空余的硬盘上重建数据。那么在接下来的 24 小时里,

³² t_{MTBF} 是用来估算机房需要准备多少块备用硬盘的, 而不是预测下一次出现硬盘损坏是在什么时候。

³³ 也可以保守地估算为平均每 100 小时会坏掉一块。

³⁴ 《Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?》

<http://www.cs.cmu.edu/~bianca/fast07.pdf> <http://storagemojo.com/2007/02/20/everything-you-know-about-disks-is-wrong/>

³⁵ 《Failure Trends in a Large Disk Drive Population》

http://research.google.com/archive/disk_failures.pdf <http://storagemojo.com/2007/02/19/googles-disk-failure-experience/>

³⁶ 《Empirical Measurements of Disk Failure Rates and Error Rates》

<http://research.microsoft.com/pubs/64599/tr-2005-166.pdf>

³⁷ 因为 $24 \times 100 \times 42 = 108\,000$ 小时。这里也可以估算为 83 天, 无妨。

³⁸ 注意这里仅仅考虑了硬盘本身故障的因素, 目的是举例说明 t_{MTBF} 的意义, 不是严肃的可靠性估计。

另外两份拷贝同时坏掉的可能性是 10^{-8} 。在一年之内，某一块数据丢失的可能性是 3.65×10^{-10} 。换言之，数据的可靠性（durability）是 9 个 9。要想进一步提高可靠性，可行的方法有两个：一是提高冗余，比如每块数据存 6 份；二是降低重建数据的时间，比如把更换并重建硬盘的时间降为 12 小时。（通过千兆网全速复制一个 2TB 的 SATA 硬盘的全部数据约需 6 小时。）

前面考虑的是从客户的角度看某一块指定的数据不丢失的概率，如果从存储服务端的角度考虑系统全部数据都不丢失的概率，情况就大不一样了。假设一个硬盘每天坏掉的概率为 p ，一共有 n 块硬盘，一天之内恰好坏 k 块硬盘的概率满足二项分布³⁹ $f(k; n, p) = C_n^k p^k (1-p)^{n-k}$ ，其中 $C_n^k = \frac{n!}{k!(n-k)!}$ 是二项式系数。在 3 份数据冗余的情况下，如果一天之内坏掉 3 块以上的硬盘，就有可能丢数据。因为如果有某块数据的 3 份冗余正好位于这 3 块坏掉的硬盘，那么这块数据就丢失了。因此不出现数据丢失的概率是 $f(0; n, p) + f(1; n, p) + f(2; n, p)$ 。把 $p = 10^{-4}$, $n = 100, 1000, 10\,000$ 分别带入公式，可知当 $n = 100$ 时，数据的可靠性接近 100%；当 $n = 1000$ 时，数据可靠性为 99.985%；而当 $n = 10\,000$ 时，数据的可靠性降为 91.971%，即每天有 8.029% 的概率出现 3 块以上硬盘同时损坏的情况。如果某个有 10 000 个硬盘的存储系统连续运行一个月，那么几乎肯定会遇到某天坏掉 3 块硬盘的情况出现，连续运行一年，几乎肯定会遇到数据丢失的情况出现。如果磁盘总数 n 继续增大，数据的总可靠性迅速降低。在无法改变硬盘故障率 p 的情况下，如果基于 GFS 思路不变，那么增加数据的冗余份数和降低重建数据的时间是提高可靠性的两个可行办法。

这看似矛盾的结果其实也很容易理解：一个人买一张彩票中头奖的概率是几百万分之一，一期彩票有几百万人购买，那么几乎每期都能开出头奖来。

可靠性与可用性（availability）⁴⁰ 是两码事，可靠性指的是数据不丢失的概率，可用性指的是数据或服务能被随时访问到的概率。可用性 = $\frac{t_{\text{MTBF}}}{t_{\text{MTBF}} + t_{\text{MTTR}}}$ ，其中 t_{MTTR} 是平均修复时间。因此为了提高可用性，提高 t_{MTBF} 和降低 t_{MTTR} 都是可行的。例如假设某服务的 t_{MTBF} 短到只有 24 小时，但 t_{MTTR} 做到 10 秒，可用性还是高达 99.988%。

值得一提的是，前面只考虑了硬盘整体故障，没有考虑数据读写错误⁴¹。普通 SATA 硬盘的误码率（bit error rate）约是 10^{-14} ，也就是说大约每读 12TB 的数据就

³⁹ http://en.wikipedia.org/wiki/Binomial_distribution

⁴⁰ 《Why Do Computers Stop and What Can Be Done About It?》

<http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf>

⁴¹ 《Understanding latent sector errors and how to protect against them》

<http://www.cs.toronto.edu/~bianca/fast10.pdf> <http://storageemojo.com/2010/03/05/storageemojos-best-paper-of-fast-10/>

会遇到有数据读不出来⁴²。磁盘带宽按 100MB/s 算,那么持续全速读 33 小时就会出现这种错误。而 ECC 内存的可靠度远高于硬盘,大约每年有 1.3% 的机器会遇到不可恢复的内存故障⁴³。因此在没有使用 RAID 的廉价服务器上,应该关闭 swap 分区,避免因磁盘读写错误而损害非存储业务的可靠性。

单机易坏的部件通常都有廉价的冗余方案(双电源、热插拔硬盘、双口网卡),但是其余的核心部件(主板⁴⁴、CPU、内存⁴⁵)没有廉价的热插拔方案,出现故障必须停机修复。如此看来,分布式系统中服务器硬件的可靠性并不如想象中高^{46 47},如果服务器的 t_{MTBF} 是 10 万小时,在 100 台服务器组成的分布式系统中,每个月出现一次服务器硬件故障的可能性略大于 50%。

这还没有考虑需要停机维护的其他原因,包括机器搬动、空调故障、供电故障、网络交换机或路由器故障、机房进水或漏雨、操作系统或其他系统软件(固件)的安全补丁等等。因此,在设计分布式系统的时候,要把这些硬件和环境的不可靠因素考虑进去,避免制定出不切实际的单机软件可靠性指标(7×24 是 overkill)。考虑了硬件不可靠的因素,实际上能降低软件的编码难度。

硬件故障固然不可避免,不过软件故障和人为故障往往更容易制造麻烦。软件故障的很大一部分是资源不足,例如内存耗尽、硬盘写满、网络带宽占满,以及文件描述符或 i-node 用完等。应对这种故障的办法是持续监控并报警,必要时自动或人工干预。有了这样的监控系统,也能减轻应用程序开发的负担,比如日志库就不必在意磁盘是否写满,因为机器上肯定有监测磁盘剩余空间的程序。

9.2.1 分布式系统的软件不要求 7×24 可靠

运行在一台机器(设备)上的软件的可靠性受限于硬件,如果硬件本身的可靠性不高,那么软件做得再可靠也没有意义。自己开发的软件的可靠性只需要略高于硬件及操作系统即可,即“不当木桶的短板”。学软件(计算机科学系)出身的人往往认

⁴² 这大大影响了 RAID5 的可用性(<http://www.zdnet.com/blog/storage/why-raid-5-stops-working-in-2009/162>)。

⁴³ 《DRAM errors in the wild: A Large-Scale Field Study》(<http://www.cs.toronto.edu/~bianca/papers/sigmetrics09.pdf>)。

⁴⁴ 某款 Intel 服务器主板的 t_{MTBF} 在 10 万至 20 万小时之间,与环境温度相关。

http://download.intel.com/support/motherboards/server/s5520hc/sb/e39529013_s5520hc_s5500hcv_s5520hct_tps_r1_9.pdf

⁴⁵ 每条 ECC 内存条每年出现不可恢复的错误的概率是 0.22%,见脚注 43。ECC 内存能纠正单 bit 错误,检测双 bit 错误,即 SECDED。如果出现多 bit 同时翻转,ECC 无能为力。

⁴⁶ 有报告称某 x86 服务器的 t_{MTBF} 是 5 万小时。

http://www.dell.com/content/topics/global.aspx/power/en/ps3q02_shetty

⁴⁷ 高端 IBM System z 的 t_{MTBF} 是 35 万小时(http://en.wikipedia.org/wiki/IBM_System_z),普通 PC 台式机的 t_{MTBF} 是 3 万小时,那么 PC 服务器的 t_{MTBF} 按 5~10 万小时估算似乎是合理的。

为硬件不会坏，而学硬件（电子信息系）出身的人一般都认为硬件不会坏才怪。半导体器件是非常娇弱的，宇宙射线的中子和集成电路封装材料中的同位素衰变产生的 α -粒子在击中硅片时会释放能量，有可能影响储能器件的“状态”，造成 bit 翻转⁴⁸。

前面分析过，如果一台服务器的 t_{MTBF} 是 10 万小时，连续运行一年出现故障的概率是 8.4%；如果一台网络交换机的 t_{MTBF} 是 20 万小时，它连续运行一年出现故障的概率是 4.3%。在编写单机服务软件或网络交换机固件的时候，程序应该尽量可靠（ 7×24 ），要能连续稳定运行一年才不会影响系统的可靠性。

但是，在一个 100 台服务器规模的分布式系统中，每个月出现一次硬件故障的概率是 51.3%；在一个 1000 台服务器规模的分布式系统中，每周出现硬件故障的概率是 81.4%。在开发运行于这些硬件上的分布式服务软件时，要求单个程序“连续稳定运行一年”是做无用功。如果一年之内因为硬件或操作失误造成 10 次停机，软件故障造成两次停机，消除这两次软件故障并不能有效地提高系统的可靠性。

要求分布式中的单个服务进程“ 7×24 不停机”通常是错误地理解了需求与约束。高可用的关键不在于做到不停机；恰恰相反，要做到能随时重启任何一个进程或服务。通过容错策略让系统保持整体可用，关键是要设计合理的协议来避免对单机过高的可靠性要求。只要重启或故障转移（failover）的时间足够短（秒级），则可用性仍然相当高。要设法从架构上搬掉这块“绊脚石”，通过多机协作达到可用性指标。在不可靠的硬件上，只有通过软件手段来提高系统的整体可用度。比方说 §9.1.2 举的 Thumbnailer 就不必做到 7×24 ，通过合理的设计协议，任何一个 Thumbnailer 都可以随时重启。

如果真要 7×24 连续运行，应该有明确的 t_{MTBF} 指标。另外， 6.9×24 行不行？ 7×23.9 行不行⁴⁹？对于非性命攸关的系统，在星期天凌晨 3 点短暂不可用会有多大的实际影响呢？

既然预料到硬件会出现故障，就能避免不切实际的软件可靠性指标。对于分布式系统中的进程来说，考虑到平均一两个月就会有程序版本更新，那么进程能连续运行数星期就可算达标了，软件升级的时候反正还是要重启进程的⁵⁰。

以上理由不是给写出低质量代码找开脱的借口，而是说在编程的时候，不必纠结于想尽一切办法防止程序崩溃。这样可以简化错误处理，用最自然的方式编写 C++

⁴⁸ http://en.wikipedia.org/wiki/Soft_error, http://people.rit.edu/lffeee/lec_reli.pdf 第 23 页。

⁴⁹ 据我所知，在金融领域，证券行业的服务程序必要的话可以每天重启，因为收市之后无交易；外汇交易的服务器可以每周重启，因为周末无交易。

⁵⁰ 在运行期热替换 DLL 通常是走火入魔的标志；真的需要在运行时替换程序逻辑的话，可以用嵌入脚本语言，把代码转换为数据。例如 §9.7 介绍的用 Groovy 编写测试逻辑。

代码, 该 new 的就 new, 该用 STL 就用, 不要视动态分配内存为“洪水猛兽”。不要把时间浪费在解决错误的问题⁵¹, 应集中精力应付更本质的业务问题。

比方说, 对于某些资源耗尽的错误可以简化处理, 在编写 64-bit 程序时也可以不必在意内存碎片 (理由见 §A.1.8)。遇到某些发生概率很小的严重错误事件时, 可以直接退出进程, 举例来说

- 如果初始化 mutex 失败, 直接退出进程好了, 反正程序也无法正确执行下去。
- 一般的程序⁵²不必在意内存分配失败, 遇到这种情况直接退出即可。一方面是在程序分配内存失败之前, 资源监控系统应该已经报警⁵³, 实施负载迁移; 另一方面, 如果真遇到 `std::bad_alloc` 异常, 也没有特别有效的办法来应对⁵⁴。
- 程序也不必考虑磁盘写满⁵⁵, 因为在磁盘写满之前, 监控系统已经报警⁵⁶。如果是关键业务, 必然已经有人采取必要的措施来腾出磁盘空间。

9.2.2 “能随时重启进程”作为程序设计目标

既然硬件和软件条件都不需要 (不允许) 程序长期运行, 那么程序在设计的时候必须想清楚重启进程的方式与代价。进程重启大致可分为软硬件故障导致的意外重启与软硬件升级引起的有计划主动重启。无论是哪种重启, 都最好让最终用户感觉不到程序在重启。重启耗时应尽量短, 中断服务的时间也尽量短, 或者最好能做到根本不中断服务。重启进程之后, 应该能自动恢复服务, 最好避免需要手动恢复。

《Google File System》论文⁵⁷第 5.1.1 节“Fast Recovery”提到:

Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated. In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process.

以上说明, 由于不必区分进程的正常退出与异常终止, 程序也就不必做到能安全退出, 只要能安全被杀即可。这大大简化了多线程服务端编程, 我们只需关心正常的业务逻辑, 不必为安全退出进程费心。

⁵¹ 西谚有云: “Wait until you have a problem before you look for a solution.”

⁵² 指非内存数据库、memcached 之类的以内存为主要资源的程序。

⁵³ 对于 32-bit 程序, 在进程内存使用超过 2GB 时; 对于 64-bit 程序, 当空闲物理内存少于 20% 时。

⁵⁴ C++ `new_handler` 在多线程中的作用非常有限。

⁵⁵ 仅考虑写诊断日志这一用途。分布式存储系统自然要设法应对磁盘写满的这一情况。

⁵⁶ 在磁盘空间使用量达到 80%、90%、95%、99% 时以不同级别报警。

⁵⁷ <http://research.google.com/archive/gfs-sosp2003.pdf>

无论是程序主动调用 `exit(3)` 或是被管理员 `kill(1)`，进程都能立即重启。这就要求程序只使用操作系统能自动回收的 IPC，不使用生命期大于进程的 IPC，也不使用无法重建的 IPC。具体说，只用 TCP 为进程间通信的唯一手段，进程一退出，连接与端口自动关闭。而且无论连接的哪一方断连，都可以重建 TCP 连接，恢复通信。

不要使用跨进程的 `mutex` 或 `semaphore`，也不要使用共享内存，因为进程意外终止的话，无法清理资源，特别是无法解锁。另外也不要使用父子进程共享文件描述符的方式来通信（`pipe(2)`），父进程死了，子进程怎么办？`pipe` 是无法重建的。

意外重启的常见情况及其原因是

- 服务进程本机重启 —— 程序 bug 或内存耗尽。
- 机器重启 —— kernel bug，偶然硬件错误。
- 服务进程移机重启 —— 硬件/网络故障。

协议设计时应该要求客户端在 TCP 连接断开后能自动重连，`muduo` 的 `TcpClient` 自带此功能。但在某些故障中客户端不能立刻收 TCP 断开的消息，因此也要求客户端检测服务端心跳，并能自动 `failover` 到备用地址（§9.3）。但是换机器的话，如何通知客户端？（§9.8.4）

如何优雅地重启？对于计划中的重启，一般可以采取以下步骤。

1. 先主动停止一个服务进程心跳：
 - 对于短连接，关闭 `listen port`，不会有新请求到达。
 - 对于长连接，客户会主动 `failover` 到备用地址或其他活着的服务端。
2. 等一段时间，直到该服务进程没有活动的请求。
3. `kill` 并重启进程（通常是新版本）。
4. 检查新进程的服务正常与否。
5. 依次重启服务端剩余进程，可避免中断服务。

除了要求客户端能正确处理心跳和 TCP 重连，还要求客户端能同时兼容新旧版本的服务端协议（§9.6）。

升级 §9.1.2 提到的 `Thumbnailer` 服务就可以采取这个办法，完全可以做到不中断服务，因为每步只杀掉一个 `Thumbnailer` 进程，缩略图服务始终是可用的。如果要升级 Web 服务器，可以考虑 Joshua Zhu 介绍的 `Nginx` 热升级办法⁵⁸。

另外一种升级软件的做法是“迁移”。先启动一个新版本的服务进程，然后让旧版本的服务进程停止接受新请求，把所有新请求都导向新进程。这样一段时间之后，

⁵⁸ <http://blog.zhuzhaoyuan.com/2009/09/nginx-internals-slides-video/>，PPT 最后 3 页。

旧版本的服务进程上已经没有活动请求，可以直接 kill 进程，完成迁移和升级。在此升级过程中服务不中断，每个用户不必在意自己是连接到新版本还是旧版本的服务。一些看似不能中断的服务可以采用这种方式升级，因为单个请求的时长总是有限的。

扯远一句，火星探路者（pathfinder）也经历过真正的远程重启⁵⁹，发生在距离地球几亿千米的火星上。

9.3 分布式系统中心跳协议的设计

前面提到使用 TCP 连接作为分布式系统中进程间通信的唯一方式，其好处之一是任何一方进程意外退出的时候对方能及时得到连接断开的通知，因为操作系统会关闭进程使用中的 TCP socket，会往对方发送 FIN 分节（TCP segment）。尽管如此，应用层的心跳还是必不可少的。原因有

- 如果操作系统崩溃导致机器重启，没有机会发送 FIN 分节。
- 服务器硬件故障导致机器重启，也没有机会发送 FIN 分节。
- 并发连接数很高时，操作系统或进程如果重启，可能没有机会断开全部连接。换句话说，FIN 分节可能出现丢包，但这时没有机会重试。
- 网络故障，连接双方得知这一情况的唯一方案是检测心跳超时。

为什么 TCP keepalive 不能替代应用层心跳？心跳除了说明应用程序还活着（进程还在，网络通畅），更重要的是表明应用程序还能正常工作。而 TCP keepalive 由操作系统负责探查，即便进程死锁或阻塞，操作系统也会如常收发 TCP keepalive 消息。对方无法得知这一异常。

心跳协议的基本形式是：如果进程 C 依赖 S，那么 S 应该按固定周期向 C 发送心跳⁶⁰，而 C 按固定的周期检查心跳。换言之，通常是服务端向客户端发送心跳，例如 §9.1.2 提到的 Thumbnailer 服务应该向 Web Server 定期发送心跳，如图 9-6 所示。

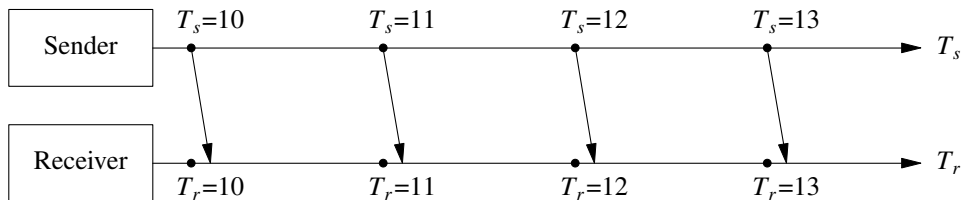


图 9-6

⁵⁹ <http://www.drdobbs.com/a-conversation-with-glenn-reeves/184411097>

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/

⁶⁰ 心跳消息就像看门狗（watch dog）电路，只有不断地逗狗才能防止电路复位（reset）。

图 9-6 中 Sender 以 1 秒为周期向 Receiver 发送心跳消息，而 Receiver 以 1 秒为周期检查心跳消息。注意到 Sender 和 Receiver 的计时器是独立的，因此可能会出现图 9-7 所示的“发送和检查时机不对齐”情况，这是完全正常的。

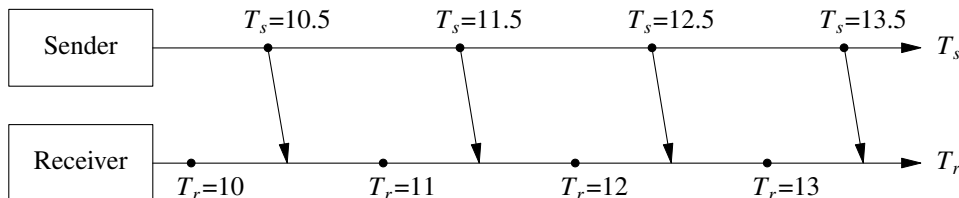


图 9-7

心跳的检查也很简单，如果 Receiver 最后一次收到心跳消息的时间与当前时间之差超过某个 timeout 值，那么就判断对方心跳失效。例如 Sender 所在的机器在 $T_s = 11.5$ 时刻崩溃，Receiver 在 $T_r = 12$ 时刻检查心跳是正常的，在 $T_r = 13$ 时刻发现过去 timeout 秒之内没有收到心跳消息，于是判断心跳失效（图 9-8）。注意到这距离实际发生崩溃的时刻已过去了 1.5 秒，这是不可避免的延迟。分布式系统没有全局瞬时状态，不存在立刻判断对方故障的方法，这是分布式系统的本质困难 (§9.1.1)。

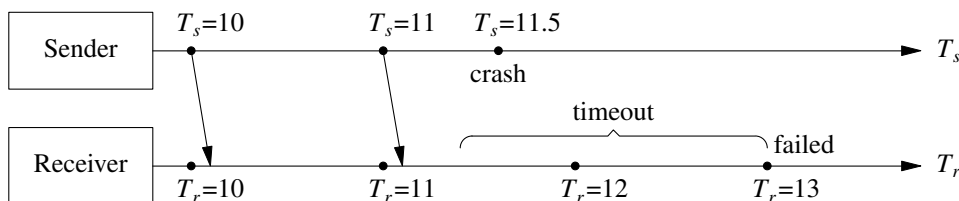


图 9-8

如果要保守一些，可以在连续两次检查都失效的情况下认定 Sender 已无法提供服务，但这种方法发现故障的延迟比前一种方法要多一个检查周期。这反映了心跳协议的内在矛盾：高置信度与低反应时间不可兼得。

现在的问题是如何确定发送周期、检查周期、timeout 这三个值。通常 Sender 的发送周期和 Receiver 的检查周期相同，均为 T_c ；而 $\text{timeout} > T_c$ ，timeout 的选择要能容忍网络消息延时波动和定时器的波动。图 9-9 中 $T_s = 12.1$ 发出的消息由于网络延迟波动，错过了检查点，如果 timeout 过小，会造成误报。

尽管发送周期和检查周期均为 T_c ，但无法保证每个检查周期内恰好收到一条心跳，有可能一条也没有收到。因此为了避免误报（false alarm），通常可取 $\text{timeout} = 2T_c$ 。

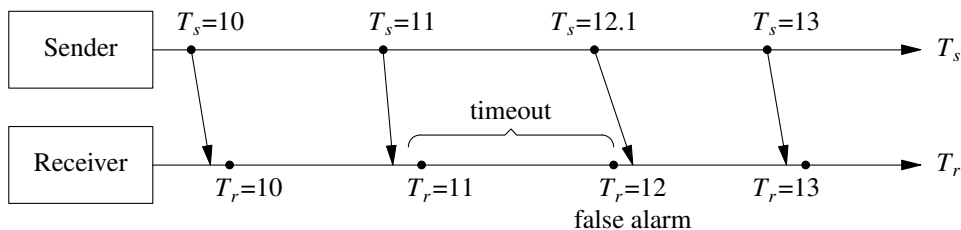


图 9-9

T_c 的选择要平衡两方面因素： T_c 越小，Sender 和 Receiver 单位时间内处理的心跳消息越多，开销越大； T_c 越大，Receiver 检测到故障的延迟也就越大。在故障延迟敏感的场所，可取 $T_c = 1s$ ，否则可取 $T_c = 10s$ 。总结一下心跳的判断规则：如果最近的心跳消息的接收时间早于 $now - 2T_c$ ，可判断心跳失效。

心跳消息应该包含发送方的标识符，可按 §9.4 的方式确定分布式系统中每个进程的唯一标识符。建议也包含当前负载，便于客户端做负载均衡。由于每个程序对“负载”的定义不同，因此心跳消息的格式也就各不相同。我认为可以在某些公共字段的基础上增加应用程序的特定字段，而不要强行规定全部程序都用相同的心跳消息格式。

以上是 Sender 和 Receiver 直接通过 TCP 连接发送心跳的做法，如果 Sender 和 Receiver 之间有其他消息中转进程，那么还应该在心跳消息中加上 Sender 的发送时间，防止消息在传输过程中堆积而导致假心跳（见图 9-10）。相应的判断规则改为：如果最近的心跳消息的发送时间早于 $now - 2T_c$ ，心跳失效。使用这种方式时，两台机器的时间应该都通过 NTP 协议与时间服务器同步，否则几秒的时钟差可能造成误判心跳失效，因为 Receiver 始终收到的是“过去”发送的消息⁶¹。

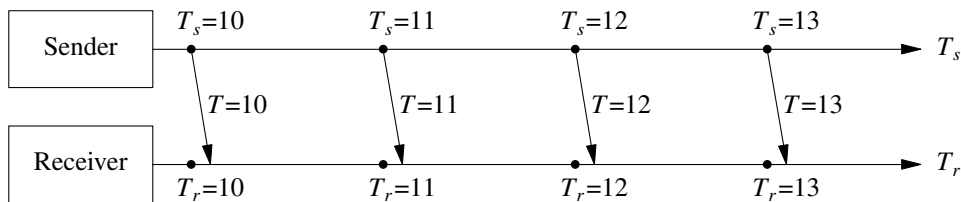


图 9-10

考虑到闰秒的影响， T_c 小于 1 秒是无意义的，因为闰秒会让两台机器的相对时差发生跳变，可能造成误报警，如图 9-11 所示。

⁶¹ 我曾经见过 x86 服务器上的集成显卡的驱动有 bug，偶尔导致机器的时间发生跳变（这跟当时 Linux 内核的时钟管理机制有关），造成某台机器的时间大大超前于其他机器（几秒），这台机器上的进程会认为它收到的心跳都是失效的。这时可以用 `ntpdate(1)` 命令强制同步时间，即可立刻修复故障。

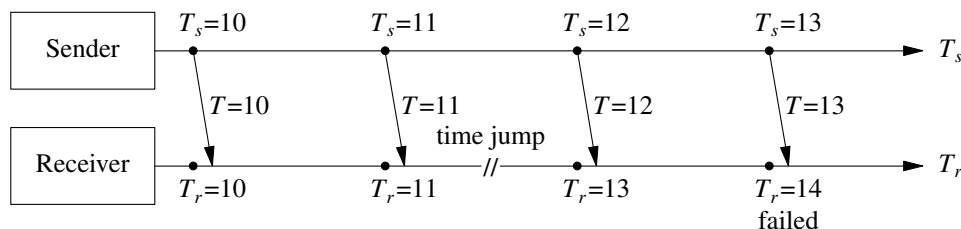


图 9-11

计算机的计时是 UTC 时间，UTC 时间会受闰秒的影响，它不是完全均匀流逝的。目前闰秒的插入点是在每年的固定日期（12 月 31 日或 6 月 30 日），不考虑星期几。这会对日常生活（特别是电子化交易）造成影响。我认为应该修改规则，在年末或年中的某个星期日凌晨（GMT 时区）插入闰秒，避免在交易时段出现时间跳变。NTP 协议对闰秒的处理也比较僵硬，基本采取暂停时钟的办法来插入闰秒，这会造成分布式系统中两台机器在发生闰秒时突然出现时间差，即便它们的时钟都是和 NTP 服务器对准的。在分布式系统中，有时我们需要特别处理这一问题，尤其在设计容错协议的时候，见 Google 的一篇 blog⁶²。

心跳协议还有两个实现上的关键点：

1. 要在工作线程发送，不要单独起一个“心跳线程”。
2. 与业务消息用同一个连接，不要单独用“心跳连接”。

这么做的原因是为了防止伪心跳。

对于第 1 点，这是防止工作线程死锁或阻塞时还在继续发心跳。对于采用 §6.6 方案 5 的单线程服务程序，应该用 `EventLoop::runEvery()` 注册周期性定时器回调，在回调函数中发送心跳消息。对于采用方案 8 的多线程服务器，应该用 `EventLoop::runEvery()` 注册周期性定时器回调，在回调函数中往线程池 post 一个任务，该任务会发送心跳消息。这样就能有效地检测工作线程死锁或阻塞的情况。对于方案 9 和方案 11 也可以采取类似的办法，对多个 `EventLoop` 轮流调用 `runInLoop()`，以防止某个业务线程死锁还继续发送心跳。

对于第 2 点，心跳消息的作用之一是验证网络畅通，如果它验证的不是收发业务数据的 TCP 连接畅通，那其意义就大为缩水了。特别要避免用 TCP 做业务连接，用 UDP 发送心跳消息，防止一旦 TCP 业务连接上出现消息堆积而影响正常业务处理时，程序还一如既往地发送 UDP 心跳，造成客户端误认为服务可用。《Release It》一书第 4.1 节“The 5 a.m. Problem”讲了一个生动的例子，用于描述心跳也是合适的。这

⁶² <http://googleblog.blogspot.hk/2011/09/time-technology-and-leaping-seconds.html>

个例子说的是 Sender 和 Receiver 位于两个数据中心，之间有网络防火墙。网络防火墙的一个特点是会自动检测 TCP 死链接，即长期没有消息往来的 TCP 连接，并清除内存中的连通规则。原来的程序通过单独的 TCP 连接发送心跳，与业务数据不在同一 TCP 连接。由于心跳始终在周期性地发送，因此，防火墙认为这个 TCP 连接是活动的。但是业务连接在每天晚上有很长一段时间没有数据交互，防火墙就判断其为死链接，并且不再转发此链接的 IP packet。尽管 Sender 和 Receiver 还认为这个 TCP 业务连接活着，但防火墙实际上已经让连接断开了。当每天早上 5 点钟第一笔订单进来的时候，始终会出现超时错误，因为业务连接的 TCP segment 无法到达对方。TCP 协议要经过很长一段时间才能真正判断连接断开（相当于中途断网，TCP 会重试很多次），这时只有重启一方的进程才能快速修复错误。当把心跳消息放到业务连接上之后，问题就迎刃而解了。

9.4 分布式系统中的进程标识

本节假定一台机器（host）只有一个 IP，不考虑 multihome 的情况⁶³。同时假定分布式系统中的每一台机器都正确运行了 NTP，各台机器的时间大体同步。

“进程（process）”是操作系统的两大基本概念之一，指的是在内存中运行的程序。在日常交流中，“进程”这个词通常不止这一个意思。有时候我们会说“httpd 进程”或者“mysqld 进程”，指的其实是 program，而不一定是特指某一个“进程”——某一次 fork() 系统调用的产物。一个“httpd 进程”重启了，它还是“一个 httpd 进程”。本文讨论的是，如何为一个程序每次运行的进程取一个唯一标识符。也就是说，httpd 程序第一次运行，进程是 httpd_1，它原地重启了，进程是 httpd_2。

本节所指的“进程标识符”是用来唯一标识一个程序的“一次运行”的。每次启动一个进程，这个进程应该被赋予一个唯一的标识符，与当前正在运行的所有进程都不同；不仅如此，它应该与历史上曾经运行过，目前已消亡的进程也都不同（这两条的直接推论是，与将来可能运行的进程也都不同）。“为每个进程命名”在分布式系统中有相当大的实际意义，特别是在考虑 failover 的时候。因为一个程序重启之后的新进程和它的“前世进程”的状态通常不一样，凡是与它打交道的其他进程(s)最好能通过它的进程标识符变更来很容易地判断该程序已经重启，而采取必要的救灾措施，防止搭错话。

⁶³ 如果要考虑 multihome，把文中的 IP 换为 hostname 即可，结论一样成立。

本节先假定每个服务端程序的端口是静态分配的，在公司内部有一个公用 wiki 来记录端口和程序的对应关系（然后通过 NIS 或 DNS 发布）。比如端口 11211 始终对应 memcached，其他程序不会使用 11211 端口；3306 始终留给 mysqld；3690 始终留给 svnserve。在分布式系统的初级阶段，这是通常的做法；到了高级阶段，多半会用动态分配端口号（§9.8），因为端口号只有 6 万多个，是稀缺资源，在公司内部也有分配完的一天。

我们假定在一台机器上，一个 listening port 同时只能由一个进程使用，不考虑古老的 listen() + fork() 模型（多个进程可以 accept 同一个端口上进来的连接）。关于这点我已经写了很多，见第 3 章。本书只考虑 TCP 协议，不考虑 UDP 协议，“端口”指的都是 TCP 端口。

9.4.1 错误做法

在分布式系统中，如何指涉（refer to）某一个进程呢，或者说一个进程如何取得自己的全局标识符（以下简称 gpid）？容易想到的有两种做法：

- ip:port⁶⁴
- host:pid

而这两种做法都有问题。为什么？

如果进程本身是无状态的，或者重启了也没有关系，那么用 ip:port 来标识一个“服务”是没问题的，比如常见的 httpd 和 memcached 都可以用它们的惯用 port（80 和 11211）来标识。我们可以在其他程序里安全地引用（refer to）“运行在 10.0.0.5:80 的那个 HTTP 服务器”，或者“10.0.0.6:11211 的 memcached”，就算这两个 service 重启了，也不会有太恶劣的后果，大不了客户端重试一下，或者自动切换到备用地址。

如果服务是有状态的，那么 ip:port 这种标识方法就有大问题，因为客户端无法区分从头到尾和自己打交道的是一个进程还是先后多个进程。如果客户端和服务端直接通过 TCP 相连，那么可以获知进程退出引发的连接断开事件。但是如果客户端与服务端之间用某种消息中间件来回转发消息，那么客户端必须通过进程标识才能识别服务端。在开发服务端程序的时候，为了能快速重启，我们一般都会设置 SO_REUSEADDR，这样的结果是前一秒站在 10.0.0.7:8888 后面的进程和后一秒占据 10.0.0.7:8888 的进程可能不相同——服务端程序快速重启了。

⁶⁴ port 是这个进程对外提供网络服务的端口号，一般就是它的 TCP listening port。

比方说，考虑一个类似 GFS 的分布式文件系统的 master，如果它仅以 ip:port 来标识自己，然后它向 shadows（不是 chunk server）下达同步指令，那么 shadows 如何得知 master 是不是已经重启呢？发指令的是 master 的“前世”还是“今生”？是不是应该拒绝“前世”的遗命？

考虑如果改成 host:pid 这种标识方式会不会好一点？我认为换汤不换药，因为 pid 的状态空间很小，重复的概率比较大。比如 Linux 的 pid 的最大值默认⁶⁵是 32768，一个程序重启之后，获得与“前世”相同 pid 的概率是 1/32768。或许有读者不相信重启之后 pid 会重复，理由是因为 pid 是递增的，遇到上限再回到目前空闲的最小 pid。考虑一个服务端程序 A，它的 pid 是 1234，它已经稳定运行了好几天，这期间，pid 已经增长了几个轮回（因为这台机器时常会启动一些后台脚本执行一些辅助工作）。在 A 崩溃的前一刻，最近被使用的 pid 已经回到了 1232，当 A 崩溃之后，某个守护进程启动一个脚本（pid = 1233）来清理 A 的 log，然后再重启 A 程序；这样一来，重启之后的 A 程序的 pid 碰巧和它的前世相同，都是 1234。也就是说，用 host:pid 不能唯一标识进程。

那么合在一起，用 ip:port:pid 呢？也不能做到唯一。它和 host:pid 面临的问题是一样的，因为 ip:port 这部分在重启之后不会变，pid 可能轮回。

我猜这时有人会想，建一个中心服务器，专门分配系统的 gpid 好了，每个进程启动的时候向它询问自己的 gpid。这错得更远：这个全局 pid 分配器的 gpid 由谁来定？如何保证它分配的 gpid 不重复（考虑这个程序也可能意外重启）？它是不是成为系统的 single point of failure？如果要对该 gpid 分配器做容错，是不是面临分布式系统的基本问题：状态迁移？

还有一种办法，用一个足够强的随机数做 gpid，这样一来确实不会重复，但是这个 gpid 本身也没有多大额外的意义，不便于管理和维护，比方说根据 gpid 找到是哪个机器上运行的哪个进程。

9.4.2 正确做法

正确做法：以四元组 ip:port:start_time:pid 作为分布式系统中进程的 gpid，其中 start_time 是 64-bit 整数，表示进程的启动时刻（UTC 时区，从 Unix Epoch 到现在的微秒数，muduo::Timestamp）。理由如下：

- 容易保证唯一性。如果程序短时间重启，那么两个进程的 pid 必定不重复（还没有走完一个轮回：就算每秒创建 1000 个进程，也要 30 多秒才会轮回，而以

⁶⁵ /proc/sys/kernel/pid_max

这么高的速度创建进程的话，服务器已基本瘫痪了。); 如果程序运行了相当长一段时间再重启，那么两次启动的 `start_time` 必定不重复。

- 产生这种 `gp_id` 的成本很低（几次低成本系统调用），没有用到全局服务器，不存在 `single point of failure`。
- `gp_id` 本身有意义，根据 `gp_id` 立刻就能知道是什么进程（`port`），运行在哪台机器（IP），是什么时间启动的，在 `/proc` 目录中的位置（`/proc/pid`）等，进程的资源使用情况也可以通过运行在那台机器上的监控程序报告出来。
- `gp_id` 具有历史意义，便于将来追溯。比方说进程 `crash`，那么我知道它的 `gp_id`，就可以去历史记录中查询它 `crash` 之前的 CPU 和内存负载有多大。

如果仅以 `ip:port:start_time` 作为 `gp_id`，则不能保证唯一性，如果程序短时间重启（间隔一秒或几秒），`start_time` 可能会来回跳变（NTP 在调时间）或暂停（正好处于闰秒期间）。

没有 `port` 怎么办？一般来说，一个网络服务程序会侦听某个端口来提供服务，如果它是个纯粹的客户端，只主动发起连接，没有主动侦听端口，`gp_id` 该如何分配呢？根据 §9.5 的观点，分布式系统中的每个长期运行的、会与其他机器打交道的进程都应该提供一个管理接口，对外提供一个维修探查通道，可以查看进程的全部状态。这个管理接口就是一个 TCP server，它会侦听某个 `port`。

使用这样的维修通道的一个额外好处是，可以自动防止重复启动程序。因为如果重复启动，`bind` 到那个运维 `port` 的时候会出错（端口已被占用），程序会立刻退出。更妙的是，不用担心进程 `crash` 来不及及清理锁（如果用跨进程的 `mutex` 就有这个风险），进程关闭的时候操作系统会自动把它打开的 `port` 都关上，下一个进程可以顺利启动。

进一步，还可以把程序的名称和版本号作为 `gp_id` 的一部分，这起到锦上添花的作用。

有了唯一的 `gp_id`，那么生成全局唯一的消息 `id` 字符串也十分简单，只要在进程内使用一个原子计数器，用计数器递增的值和 `gp_id` 即可组成每个消息的全局唯一 `id`。这个消息 `id` 本身包含了发送者的 `gp_id`，便于追溯。当消息被传递到多个程序中，也可以根据 `gp_id` 追溯其来源。

9.4.3 TCP 协议的启示

本节讲的这个 `gp_id` 其实是由 TCP 协议启发而来的。TCP 用 `ip:port` 来表示 `end-point`，两个 `endpoint` 构成一个 `socket`。这似乎符合一开始提到的以 `ip:port` 来标

识进程的做法。其实不然。在发起 TCP 连接的时候，为了防止前一次同样地址的连接（相同的 `local_ip:local_port:remote_ip:remote_port`）的干扰（称为 wandering duplicates，即流浪的 packets），TCP 协议使用 seq 号码（这种在 SYN packet 里第一次发送的 seq 号码称为 initial sequence number，ISN）来区分本次连接和以往的连接。TCP 的这种思路与我们防止进程的“前世”干扰“今生”很相像。内核每次新建 TCP 连接的时候会设法递增 ISN 以确保与上次连接最后使用的 seq 号码不同。相当于说把 start_time 加入到了 endpoint 之中，这就很接近我们后面提到的“正确的 gpid”做法了。（当然，原始 BSD 4.4 的 ISN 生成算法有安全漏洞，会导致 TCP sequence prediction attack，Linux 内核已经采用更安全的办法来生成 ISN。）

9.5 构建易于维护的分布式程序

本节标题中的“易于维护”指的是 supportability，不是 maintainability。前者是从运维人员的角度说，程序管理起来很方便，日常的劳动负担小；后者是从开发人员的角度说，代码好读好改。

在《分布式系统的工程化开发方法》⁶⁶ 演讲中我提到了一个观点：分布式系统中的每个长期运行的、会与其他机器打交道的进程都应该提供一个管理接口，对外提供一个维修探查通道，可以查看进程的全部状态。一种具体的做法是在程序里内置 HTTP 服务器，能查看基本的进程健康状态与当前负载，包括活动连接及其用途，能从 root set 开始查到每一个业务对象的状态。这种做法类似 Java 的 JMX，又类似 memcached 的 stats 命令。

这里展开谈一谈这么做的必要性。分成两个方面来说：一、在服务程序内置监控接口的必要性；二、HTTP 协议的便利性。

必要性

在程序中内置监控接口可以说是受了 Linux procfs 的启发。在 Linux 下，查看内核的状态不需要任何特殊的工具，只要用 ls 和 cat 在 /proc 目录下查看文件就行了。要知道当前系统中运行了哪些进程、每个进程都打开了哪些文件、进程的内存和 CPU 使用情况如何、每个进程启动了几个线程、当前有哪些 TCP 连接、每个网卡收发的字节数等等，都可以在 /proc 中找到答案。Linux Kernel 通过 procfs 这么一个探查接口把状态充分暴露出来，让监控操作系统的运行变得容易。

⁶⁶ <http://blog.csdn.net/Solstice/article/details/5950190>

但是 `procfs` 也有两点明显的不足：

1. 它只能暴露 `system-wide` 的数据，不能查看每个进程内部的数据。
2. 它是本地文件系统，必须要登录到这台机器上才能查看，如果要管理很多台机器，则势必增加工作量。

对于第一点，举例来说，我想知道某个我们自己编写的服务进程的运行情况：

- 到目前为止累计接受了多少个 TCP 连接。
- 当前有多少活动连接（这个可以通过 `procfs` 查看）。
- 每个活动连接的用途是什么。
- 一共响应了多少次请求。
- 每次请求的平均输入输出数据长度是多少字节。
- 每次请求的平均响应时间是多少毫秒。
- 进程平均有多少个活动请求（并发请求）。
- 并发请求数的峰值是多少，出现在什么时候。
- 某个连接上平均有多少个活动请求。
- 进程中 `XXXRequest` 对象有多少份实体。
- 进程中打开了多少个数据库连接，每个连接的存活时间是多少。
- 程序中有一个 `hashmap`，保存了当前的活动请求，我想把它打印出来。
- 某个请求似乎卡在某个步骤了，我想打印进程中该请求的状态。

这些正当需求只有通过程序主动暴露状态才能满足；否则，就算 `ssh` 登录到这台机器上，也看不到这些有用的进程内部信息。（总不能 `gdb attach` 吧？那就让服务进程暂停响应了。且不说 `gdb` 打印一个 `hashmap` 有多麻烦。）

便利性

如果程序要主动暴露内部状态，那么以哪种方式最为便利呢？当然是 `HTTP`。`HTTP` 的好处如下：

- 它是 `TCP server`，可以远程访问，不必登录到这台机器上。
- `TCP server` 的另一个好处是能安全方便地防止程序重复启动，这一点在 §9.4 已有论述。
- 最基本的 `HTTP` 协议实现起来很简单，不会给服务端程序带来多大负担，见 `muduo::net::HttpServer` 的例子。

- 不必使用特定的客户端程序，用普通 Web 浏览器就能访问。
- 可以比较容易地用脚本语言实现客户端，便于自动化的状态收集与分析。
- HTTP 是文本协议，紧急情况下在命令行用 `curl/wget` 甚至 `telnet` 也能访问（比方说你在家通过 `ssh` 连到公司服务器解决某个线上问题，这时候没有 Web 浏览器可用）。
- 借助 URL 路径区分，很容易实现有选择地查看一些信息，而不是把进程的全部状态一股脑儿全 `dump` 出来，见 `muduo::net::Inspector` 的例子，如 `http://host:port/request/xxx` 表示 ID 为 `xxx` 的请求的状态。
- HTTP 天生支持聚合，一个浏览器页面可以内置多个 `iframe`，一眼就能看清多个进程的状态。
- 除了 `GET` method，如果有必要，还可以实现 `PUT/POST/DELETE`，通过 HTTP 协议来控制并修改进程的状态，让程序“能观能控”⁶⁷。
- 最好能在运行时修改程序用到的后台服务的 `host:port`（原本写在配置文件中），这样可以随时主动切换后台服务（平滑升级或故障预防），而无须重启本进程。
- 必要的时候还可以用 `REST` 的方式实现高级的聚合，见我在演讲中的“一种 `REST` 风格的监控”。

另外，我们讨论分布式系统是运行在企业防火墙之内的基础设施，HTTP 的安全性应该由防火墙保证。就好比你的 Hadoop master 和 memcached 不会暴露给外网一样，在公司内部使用 HTTP，只要没有人故意搞破坏就没事。

实例

演讲中我举了 Google 的例子⁶⁸，Google 的每个服务进程（无论 C++ 或 Java）都会

- 提供 HTML 的状态页面，以便快速诊断问题；
- 通过某种标准接口暴露一组 `key-value pairs`；
- 监控程序定期从全部服务进程收集性能数据；
- RPC 子系统对全部请求采样：错误的，耗时 $> 0.05s$, $> 0.1s$, $> 0.5s$, $> 1.0s$ ……

当然，我们看不到 Google 内部的服务器的状态页面究竟是什么样子，不过可以看看别的例子，比如 Hadoop。Hadoop 有四种主要 services: `NameNode`、`DataNode`、

⁶⁷ “能观性 (Observability)” 和 “能控性 (Controllability)” 是自动控制领域的术语，此处 “能观性” 是指能获知进程的一切状态，“能控性” 是指能让进程达到我们想要的任何状态。

⁶⁸ 见 Jeff Dean 演讲中的 “Add Sufficient Monitoring/Status/Debugging Hooks” 一节，
<http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf>。

JobTracker、TaskTracker。每种 service 都内置了 HTTP 状态页面，其默认 HTTP 端口分别是：

- NameNode —— 50070
- DataNode —— 50075
- JobTracker —— 50030
- TaskTracker —— 50060

如果某台机器运行了 DataNode 和 TaskTracker，那么我们可以通过访问 `http://hostname:50075` 和 `http://hostname:50060` 来方便地查询其运行状态。

例外

如果不方便内置 HTTP 服务，那么内置一个简单的 telnet 服务也不难，就像 memcached 的 stats 命令那样。

如果服务程序本身以 RPC 方式提供服务，那么可以不必内置 HTTP 服务，而是增加一个 RPC 调用实现相同的功能。这个 RPC 可以命名为 `inspect()`，输入的内容类似 URL，返回的是该 URL 对应的页面内容，可以是文本格式，也可以是 RPC 原生的打包格式。

如果是 Java 程序，可以直接使用 JMX，也可以使用本节提到的 HTTP 方法，这样管理和监控的一致性较好，至少不需要为 Java 服务进程准备特殊的客户端。

小结

在自己编写分布式程序的时候，提供一个维修通道是很有必要的，它能帮助日常运维，而且在出现故障的时候帮助排查。相反，如果不在程序开发的时候统一预留这些维修通道，那么运维起来就抓瞎了——每个进程都是黑盒子，出点什么情况都得拼命查 log 试图恢复（猜测）进程的状态，工作效率不高。

9.6 为系统演化做准备

一个分布式系统的生命期会长达数年，在首次上线运行之后，系统会经历多次升级和演化，因此在一开始设计的时候要适当为将来考虑。一个典型的考虑点是：通信的双方很有可能不会同时升级。通信双方可能由不同的开发团队开发，开发和发布周期不同步。有可能为了稳妥起见先升级其中一方，验证稳定性，然后再升级另一方。当然，升级之前一定要制定好 rollback 计划，留好退路。

具体来说,服务端新加功能,不一定所有的客户端都会马上升级并用上新功能,因此新的服务端上线之后要保证和现有的客户端的功能和协议的兼容性,这样才能平稳升级。系统中的不同组件可能用不同的编程语言来编写,有时候会把一个组件换一种语言重写,因此应该使用一种跨语言的可扩展消息格式。

9.6.1 可扩展的消息格式

考虑服务端升级的可能时,一种很容易想到的做法是在消息中放入版本号,服务端每次收到消息,先根据版本号做分发(dispatch)。实践证明这种做法是非常不靠谱的,很容易在服务端留下一堆垃圾代码,时间一长,谁也弄不清版本之间具体有哪些细微差别,也不敢轻易删掉处理旧版本消息的代码,历史包袱就一直背下去。

因此,可扩展消息格式的第一条原则是避免协议的版本号,否则代码里会有一堆堆难以维护的 switch-case,就像 Google Protocol Buffers 文档举的反面例子⁶⁹。

```
if (version == 3) {  
    // ...  
} else if (version > 4) {  
    if (version == 5) {  
        // ...  
    }  
    // ...  
}
```

另一种常见错误是通过 TCP 连接发送 C struct 或使用 bit fields。这或许是因为在学习 TCP/IP 协议和网络编程的时候,书上一般会画出 IP header 和 TCP header,其中就有 bit fields,这给人留下了一个错误印象,似乎网络协议应该这么设计。其实不是这样的,C struct 和 bit fields 的缺点很多。其一是不易升级。如果在 C struct 里新加了一些元素,通常要求客户端和服务端一起升级,否则就语言不通了。其二是不跨语言⁷⁰。如果客户端和服务端用不同的语言来编写,那么让非 C/C++ 语言生成和解析这种消息格式是比较麻烦的。而且更重要的是需要时刻维护其他语言的打包、解包代码与 C/C++ 头文件里 struct 定义的同步,稍不注意就会造成格式解析错乱。

解决办法是,采用某种中间语言来描述消息格式(schema),然后生成不同语言的解析与打包代码。如果用文本格式,可以考虑 JSON 或 XML;如果用二进制格式,可以考虑 Google Protocol Buffers。使用文本格式的一个常见问题是处理转义字符(escape character),比如消息 id 字段如果出现 '&',在 XML 中要写成 &。如果公司名字是 AT&T,在 XML 中要写成 <company>AT&T</company>。

⁶⁹ <https://developers.google.com/protocol-buffers/docs/overview>

⁷⁰ 就算是 C/C++ 也要考虑 32-bit 或 64-bit 平台、endian、编译器对齐(alignment)的影响等等。

Google Protobuf 是结构化的二进制消息格式⁷¹，兼顾性能⁷²与可扩展性。其文档中说（<https://developers.google.com/protocol-buffers/docs/cpptutorial>）：

Importantly, the protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format.

这种“中间语言”或者叫“数据描述语言”定义的消息格式可以有可选字段（optional fields），一举解决了服务端和客户端升级的难题。新版的服务端可以定义一些 optional fields，根据请求中这些字段的存在与否来实施不同的行为，即可同时兼容旧版和新版的客户端。给每个 field 赋终生不变的 id 是保证兼容性的绝招，Google Protobuf 的文档强调在升级 proto 文件时要注意⁷³：

- you must *not* change the tag numbers of any existing fields.
- you must *not* add or delete any required fields.

proto 文件就像 C/C++ 动态库的头文件，其中定义的消息就是库（分布式服务）的接口，一旦发布就不能做有损二进制兼容性的修改。因此 §11.2 的知识可以套用过来，包括不能更改已有的 enum 类型的成员的值等。

PNG 文件给我们很好的启示。PNG 是一种精心设计的二进制文件格式，文件由一系列数据块（chunks）组成，每个数据块的前 4 个字节表示该数据块的长度，接下来的 4 个字节代表该数据块的类型。PNG 的解译程序会忽略那些自己不认识的数据块，因此 PNG 文件没有版本之说，不存在前后版本不兼容的问题。

Google Protobuf 是精心设计的协议格式，还体现在客户端可以先升级，发送服务端不认识的 field，服务端可以安全地跳过这些字段。

TCP/IP 在设计的时候也在固定长度的 header 之后预留了可选项，目前广泛使用的有 window scale 和 timestamp 等。

9.6.2 反面教材：ICE 的消息打包格式

ICE⁷⁴ 是一个对象中间件，它实现类似 CORBA 的跨语言、跨进程的函数调用。我对 ICE 的设计及实现很不以为然。其中一个原因是它按 struct field 和函数参数的

⁷¹ 结构化的意思是说一个消息可以使用其他自定义消息类型为成员，也可以包含数组，数组的元素可以是其他自定义消息类型。

⁷² Protobuf 二进制格式中的整数采用变长编码，可以节约带宽，降低延迟（<https://developers.google.com/protocol-buffers/docs/encoding>）。

⁷³ <https://developers.google.com/protocol-buffers/docs/proto#updating>

⁷⁴ <http://www.zeroc.com>

顺序来打包消息，难以无痛升级。一旦给 `struct` 新加一个成员或者给函数新加一个参数，客户端和服务端必须同时升级，否则就言语不通了。另外一个原因是它的远程函数调用居然能返回异常。也就是说，当服务端的 RPC 函数抛出异常时，RPC 机制会捕捉这个异常，通过网络传送到客户端，在客户端重新抛出这个异常。我实在不理解这种异常捕捉下来有何用处，客户端可能是 Python，服务端是 C++，Python 代码拿到 C++ 异常能干什么？还不如老老实实直接返回错误代码，处理起来更简单。

9.7 分布式程序的自动化回归测试

本节所谈的“测试”指的是“开发者测试（developer testing）”，由程序员自己做，不是由 QA 团队进行的系统测试。这两种测试各有各的用途，不能相互替代。

§11.1 “朴实的 C++ 设计”中谈道：“为了确保正确性，我们另外用 Java 写了一个测试夹具（test harness）来测试我们这个 C++ 程序。这个测试夹具模拟了所有与我们这个 C++ 程序打交道的其他程序，能够测试各种正常或异常的情况。”

本节详细介绍一下这个 test harness 的做法。

自动化测试的必要性

我想自动化测试的必要性无须赘言，自动化测试是 absolutely good stuff。

基本上，要是没有自动化的测试，我是不敢改产品代码的（“改”包括添加新功能和重构）。自动化测试的作用是把程序已经实现的 features 以 test case 的形式固化下来，将来任何代码改动如果破坏了现有的功能需求就会触发测试 failure。好比 DNA 双链的互补关系，这种互补结构对保持生物遗传的稳定有重要作用。类似地，自动化测试与被测程序的互补结构对保持系统的功能稳定有重要作用。

9.7.1 单元测试的能与不能

一提到自动化测试，我猜很多人想到的是单元测试（unit testing）。单元测试确实有很大的用处，对于解决某一类型的问题很有帮助。粗略地说，单元测试主要用于测试一个函数、一个 class 或者相关的几个 class。

最典型的是测试纯函数，比如计算个人所得税的函数，输入是“起征点、扣除五险一金之后的应纳税所得额、税率表”，输出是应该缴的个税。又比如，我在《程序

中的日期与时间》的第一章“日期计算”⁷⁵中用单元测试来验证 Julian day number 算法的正确性。再比如，我在《“过家家”版的移动离线计费系统实现》⁷⁶和《模拟银行窗口排队叫号系统的运作》⁷⁷中用单元测试来检查程序运行的结果是否符合预期。（最后这个或许不是严格意义上的单元测试，更像是验收测试。）

为了能用单元测试，程序代码有时候需要做一些改动。这对 Java 通常不构成问题（反正都编译成 jar 文件，在运行的时候指定 entry point）。对于 C++，一个程序只能有一个 main() 入口点，要采用单元测试的话，需要把功能代码（被测对象）做成一个 library，然后让单元测试代码（包含 main() 函数）link 到这个 library 上；当然，为了正常启动程序，我们还需要写一个普通的 main()，并 link 到这个 library 上。

单元测试的缺点

根据我的个人经验，我发现单元测试有以下缺点。

阻碍大型重构 单元测试是白盒测试，测试代码直接调用被测代码，测试代码与被测代码紧耦合。从理论上说，“测试”应该只关心被测代码实现的功能，不用管它是如何实现的（包括它提供什么样的函数调用接口）。比方说，以前面的个税计算器函数为例，作为使用者，我们只关心它算的结果是否正确。但是，如果要写单元测试，测试代码必须调用被测代码，那么测试代码必须要知道个税计算器的 package、class、method name、parameter list、return type 等等信息，还要知道如何构造这个 class。以上任何一点改动都会造成测试失败（编译就不通过）。

在添加新功能的时候，我们常会重构已有的代码，在保持原有功能的情况下让代码的“形状”更适合实现新的需求。一旦修改原有的代码，单元测试就可能编译不过：比如给成员函数或构造函数添加一个参数，或者把成员函数从一个 class 移到另一个 class。对于 Java，这个问题还比较好解决，因为 IDE 的重构功能很强，能自动找到 references，并修改之。

对于 C++，这个问题更为严重，因为一改功能代码的接口，单元测试就编译不过了，而 C++ 通常没有自动重构工具（语法太复杂，语意太微妙）可以帮我们，都得手动来。要么每改动一点功能代码就修复单元测试，让编译通过；要么留着单元测试编译不通过，先把功能代码改成我们想要的样子，再来统一修复单元测试。

⁷⁵ <http://blog.csdn.net/solstice/article/details/5814486>

⁷⁶ <http://www.cnblogs.com/Solstice/archive/2011/04/22/2024791.html>

⁷⁷ <http://blog.csdn.net/Solstice/article/details/6324749>

这两种做法都有困难，前者，C++ 编译缓慢，如果每改动一点就修复单元测试，一天下来也前进不了几步，很多时间都浪费在了等待编译上；后者，问题更严重，单元测试与被测代码的互补性是保证程序功能稳定的关键，如果大幅修改功能代码的同时又大幅修改了单元测试，那么如何保证前后的单元测试的效果（测试点）不变？如果单元测试自身的代码发生了改动，如何保证它测试结果的有效性？会不会某个手误让功能代码和单元测试犯了相同的错误，负负得正，测试结果还是绿的，但是实际功能已经亮了红灯？难道我们要为单元测试编写单元测试吗？

有时候，我们需要重新设计并重写某个程序（有可能换用另一种语言）。这时候旧代码中的单元测试完全作废了（代码结构发生巨大改变，甚至连编程语言都换了），其中包含的宝贵的业务知识也付之东流，岂不可惜？

为了方便测试而施行依赖注入，破坏代码的整体性 为了让代码具有“可测试性”，我们常会使用依赖注入技术，这么做的好处据说是“解耦”，坏处就是割裂了代码的逻辑：单看一块代码不知道它是干嘛的，它依赖的对象不知道是在哪儿创建的，如果一个 `interface` 有多个实现，不到运行的时候不知道用的是哪个实现。（动态绑定的初衷就是如此，想来读过“以面向对象思想实现”的代码的人都明白我在说什么。）

以 §7.3 “Boost.Asio 的聊天服务器”中出现的聊天服务器 `ChatServer` 为例，`ChatServer` 直接使用了 `muduo::net::TcpServer` 和 `muduo::net::TcpConnection` 来处理网络连接并收发数据，这个设计简单直接。如果要为 `ChatServer` 写单元测试，那么首先它肯定不能在构造函数里初始化 `TcpServer` 了。

稍微复杂一点的测试要用 mock object `ChatServer` 用 `TcpServer` 和 `TcpConnection` 来收发消息，为了能单元测试，我们要为 `TcpServer` 和 `TcpConnection` 提供 mock 实现，原本一个具体类 `TcpServer` 就变成了一个 `TcpServer interface` 加两个实现 `TcpServerImpl` 和 `TcpServerMock`，同理 `TcpConnection` 也一化为三。`ChatServer` 本身的代码也变得复杂，我们要设法把 `TcpServer` 和 `TcpConnection` 注入其中，`ChatServer` 不能自己初始化 `TcpServer` 对象。

这恐怕是在 C++ 中使用单元测试的主要困难之一。Java 有动态代理，还可以用 `cglib` 来操作字节码以实现注入。而 C++ 比较原始，只能自己手工实现 `interface` 和 `implementations`。这样原本紧凑的以 `concrete class` 构成的代码结构因为单元测试的需要而变得松散（所谓“面向接口编程”嘛），而这么做的目的仅仅是为了满足“源码级的可测试性”，是不是有一点因小失大呢？（这里且暂时忽略虚函数和普通函数在性能上的些微差别。）对于不同的 `test case`，可能还需要不同的 mock 对象，比如 `TcpServerMock` 和 `TcpServerFailureMock`，这又增加了编码的工作量。

此外，如果程序中用到的涉及 IO 的第三方库没有以 `interface` 方式暴露接口，而是直接提供的 `concrete class`（这是对的，因为 C++ 中应该“避免使用虚函数作为库的接口”，见 §11.3），这也让编写单元变得困难，因为总不能自己挨个 `wrapper` 一遍吧？难道用 `link-time` 的注入技术？

某些 `failure` 场景难以测试 而考察这些场景对编写稳定的分布式系统有重要作用。比方说：网络连不上、数据库超时、系统资源不足。

对多线程程序无能为力 如果一个程序的功能涉及多个线程合作，那么就比较难用单元测试来验证其正确性。

如果程序涉及比较多的交互（指和其他程序交互，不是指图形用户界面），用单元测试来构造测试场景比较麻烦，每个场景要写一堆无趣的代码。而这正是分布式系统最需要测试的地方。

总的来说，单元测试是一个值得掌握的技术，用在适当的地方确实能提高生产力。同时，在分布式系统中，我们还需要其他的自动化测试手段。

9.7.2 分布式系统测试的要点

在分布式系统中，`class` 与 `function` 级别的单元测试对整个系统的帮助不大。这种单元测试对单个程序的质量有帮助，但是，一堆砖头垒在一起是变不成大楼的。

分布式系统测试的要点是测试进程间的交互：一个进程收到客户请求，该如何处理，然后转发给其他进程；收到响应之后，又修改并应答客户。测试这些多进程协作的场景才算测到了点子上。

假设一个分布式系统由四五种进程组成，每个程序有各自的开发人员。对于整个系统，我们可以用脚本来模拟客户，自动化地测试系统的整体运作情况，这种测试通常由 QA 团队来执行，也可以作为系统的冒烟测试。

对于其中每个程序的开发人员，上述测试方法对日常的开发帮助不大。因为测试要能通过，必须整个系统都正常运转才行，在开发阶段，这一点不是时时刻刻都能满足的（有可能你用到的新功能对方还没有实现，这反过来影响了你的进度）。另一方面，如果出现测试失败，开发人员不能立刻知道这是自己的程序出错（也有可能是环境原因造成的错误），这通常要去读程序日志才能判定。还有，作为开发者测试，我们希望它无副作用，每天反复多次运行也不会增加整个环境的负担，以整个 QA 系统为测试平台不可避免地要留下一些垃圾数据，而清理这些数据又会花一些宝贵的工作

时间。（你得判断数据是自己的测试生成的还是别人的测试留下的，不能误删了别人的测试数据。）

作为开发人员，我们需要一种单独针对自己编写的那个程序的自动化测试方案，一方面提高日常开发的效率，另一方面作为自己那个程序的功能验证测试集，以及回归测试（regression tests）。

9.7.3 分布式系统的抽象观点

一台机器两根线

形象地来看（见图 9-12），一个分布式系统就是一堆机器，每台机器的“屁股”上拖着两根线：电源线和网线（不考虑 SAN 等存储设备），电源线插到电源插座上，网线插到交换机上。

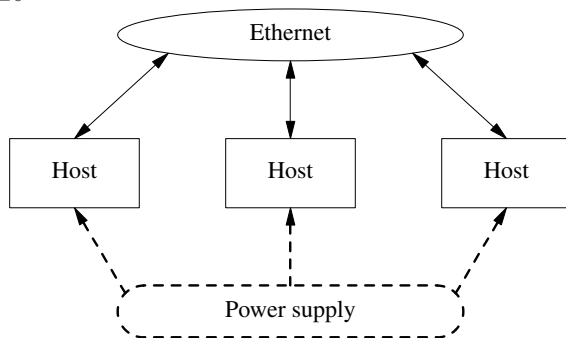


图 9-12

这个模型实际上说明，一台机器、一个程序表现出来的行为完全由它接出来的两根线展现，本书不谈电源线，只谈网线。（“在乎服务器的功耗”在我看来就是公司利润率很低的标志，要从电费上抠成本。）

如果网络是普通的千兆以太网，那么吞吐量不大于 125MB/s。这个吞吐量比起现在的 CPU 运算速度和内存带宽简直小得可怜。这里我想提的是，对于不特别在意 latency 的应用，只要能让千兆以太网的吞吐量饱和或接近饱和，用什么编程语言其实无所谓。Java 做网络服务端开发也是很好的选择（不是指 Web 开发，而是做一些基础的分布式组件，例如 ZooKeeper 和 Hadoop 之类）。尽管可能 C++ 只用了 15% 的 CPU，而 Java 用了 30% 的 CPU，Java 还占用更多的内存，但是千兆网卡带宽都已经跑满，那些省下的资源也只能浪费了；对于外界（从网线上看来）而言，两种语言的效果是一样的，而通常 Java 的开发效率更高。Java 比 C++ 慢一些，但是通过千兆网络不一定能看得出这个区别来。同样的道理，单机程序的某些“性能优化”不一

定真能提高系统整体表现出来的、能被观察到的性能，这也是本书基本不谈微观性能优化的主要原因。在弄清楚系统瓶颈之前贸然投入优化往往是浪费时间和精力，还耽误了项目进度，颇为不值。

进程间通过 TCP 相互连接

我在 §3.4 提倡仅使用 TCP 作为进程间通信的手段，此处这个观点将再次得到验证。

图 9-13 是 Hadoop 的分布式文件系统 HDFS 的架构简图。

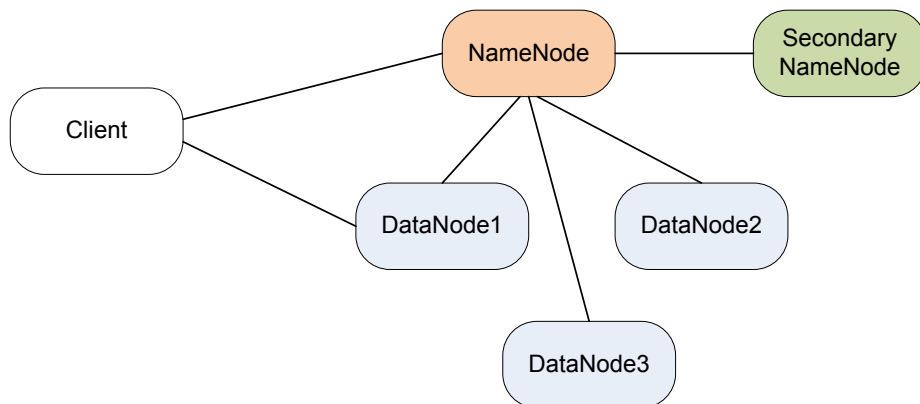


图 9-13

HDFS 有四个角色参与其中，**NameNode**（保存元数据）、**DataNode**（存储节点，多个）、**Secondary NameNode**（定期写 check point）、**Client**（客户，系统的使用者）。这些进程运行在多台机器上，之间通过 TCP 协议互联。程序的行为完全由它在 TCP 连接上的表现决定（TCP 就好比前面提到的“网线”）。

在这个系统中，一个程序其实不知道与自己打交道的到底是什么。比如，对于 **DataNode**，它其实不在乎自己连接的是真的 **NameNode** 还是某个调皮的小孩用 Telnet 模拟的 **NameNode**，它只管接受命令并执行。对于 **NameNode**，它其实也不知道 **DataNode** 是不是真的把用户数据存到磁盘上去了，它只需要根据 **DataNode** 的反馈更新自己的元数据就行。这已经为我们指明了方向。

9.7.4 一种自动化的回归测试方案

假如我是 **NameNode** 的开发者，为了能自动化测试 **NameNode**，我可以为它写一个 test harness（这是一个独立的进程），这个 test harness 仿冒（mock）了与被测试进程打交道的全部程序。如图 9-14 所示，是不是有点像“缸中之脑”？

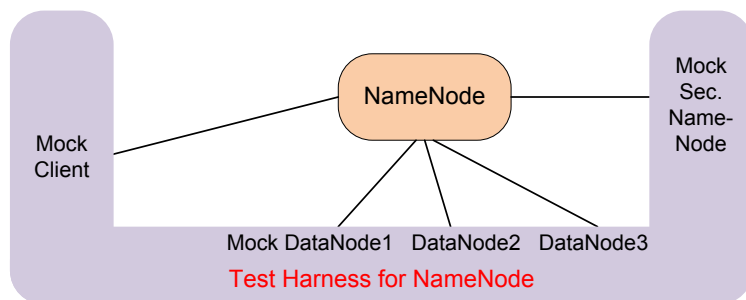


图 9-14

对于 DataNode 的开发者，他们也可以写一个专门的 test harness，模拟 Client 和 NameNode（见图 9-15）。

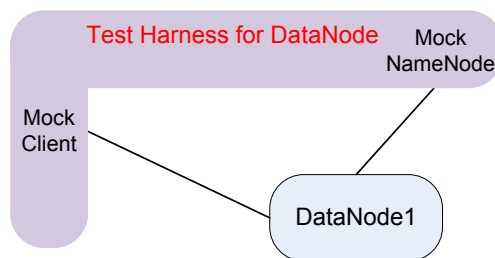


图 9-15

test harness 的优点

- 完全从外部观察被测程序，对被测程序没有侵入性，代码该怎么写就怎么写，不需要为测试留路。
- 能测试真实环境下的表现，程序不是单独为测试编译的版本，而是将来真实运行的版本。数据也是从网络上读取，发送到网络上。
- 允许被测程序做大的重构，以优化内部代码结构，只要其表现出来的行为不变，测试就不会失败。（在重构期间不用修改 test case。）
- 能比较方便地测试 failure 场景。比如，若要测试 DataNode 出错时 NameNode 的反应，只要让 test harness 模拟的那个 mock DataNode 返回我们想要的出错信息。要测试 NameNode 在某个 DataNode 失效之后的反应，只要让 test harness 断开对应的网络连接即可。要测量某请求超时的反应，只要让 test harness 不返回结果即可。这对构建可靠的分布式系统尤为重要。
- 帮助开发人员从使用者的角度理解程序，程序的哪些行为在外部是看得到的，哪些行为是看不到的。

- 有了一套比较完整的 test cases 之后，甚至可以换种语言重写被测程序（假设为了提高内存利用率，换用 C++ 来重新实现 NameNode），测试用例依旧可用。这时 test harness 起到知识传承的作用。
- 发现 bug 之后，往 test harness 里添加能复现 bug 的 test case，修复 bug 之后，test case 继续留在 harness 中，防止出现回归（regression）。

实现要点

- test harness 的要点在于隔断被测程序与其他程序的联系，它冒充了全部其他程序。这样被测程序就像被放到测试台上观察一样，让我们只关注它一个。
- test harness 要能发起或接受多个 TCP 连接，可能需要用某个现成的 NIO 网络库，如果不想写成多线程程序的话。
- test harness 可以与被测程序运行在同一台机器，也可以运行在两台机器上。在运行被测程序的时候，可能要用一个特殊的启动脚本把它依赖的 host:port 指向 test harness。
- test harness 只需要表现得跟它要 mock 的程序一样，不需要真的去实现复杂的逻辑。比如 mock DataNode 只需要对 NameNode 返回 “Yes sir, 数据已存好”，而不需要真的把数据存到硬盘上。若要 mock 比较复杂的逻辑，可以用 “记录 + 回放” 的方式，把预设的响应放到 test case 里回放（replay）给被测程序。
- 因为通信走 TCP 协议，test harness 不一定要和被测程序用相同的语言，只要符合协议就行。试想如果用共享内存实现 IPC，这是不可能的。本书 §7.6 提到利用 Protobuf 的跨语言特性，我们可以采用 Java 为 C++ 服务程序编写 test harness。其他跨语言的协议格式也行，比如 XML 或 JSON。
- test harness 运行起来之后，等待被测程序的连接，或者主动连接被测程序，或者兼而有之，取决于所用的通信方式。
- 一切就绪之后，test harness 依次执行 test cases。一个 NameNode test case 的典型过程是：test harness 模仿 client 向被测 NameNode 发送一个请求（如创建文件），NameNode 可能会联络 mock DataNode，test harness 模仿 DataNode 应有的响应，NameNode 收到 mock DataNode 的反馈之后发送响应给 client，这时 test harness 检查响应是否符合预期。
- test harness 中的 test cases 以配置文件（每个 test case 有一个或多个文本配置文件，每个 test case 占一个目录）方式指定。test harness 和 test cases 连同程序代码一起用 version control 工具管理起来。这样能复现以外任何一个版本的应有行为。

- 对于比较复杂的 test case, 可以用嵌入式脚本语言来描述场景。如果 test harness 是用 Java 写的, 那么可以嵌入 Groovy, 就像笔者在《“过家家”版的移动离线计费系统实现》(地址见 p. 371 脚注 76) 中用 Groovy 实现计费逻辑一样。Groovy 调用 test harness 模拟多个程序分别发送多份数据并验证结果, Groovy 本身就是程序代码, 可以有逻辑判断甚至循环。这种动静结合的做法在不增加 test harness 复杂度的情况下提供了相当高的灵活性。
- test harness 可以有一个命令行界面, 程序员输入 “run 10” 就选择执行第 10 号 test case。

几个实例

test harness 这种测试方法适合测试有状态的、与多个进程通信的分布式程序, 除了 Hadoop 中的 NameNode 与 DataNode, 我还能想到几个例子。

chat 聊天服务器 聊天服务器会与多个客户端打交道, 我们可以用 test harness 模拟 5 个客户端, 模拟用户上下线、发送消息等情况, 自动测试聊天服务器的功能。

连接服务器、登录服务器、逻辑服务器 这是云风在他的 blog 中提到的三种网游服务器⁷⁸, 我这里借用来说例子。

如果要为连接服务器写 test harness, 那么需要模拟客户 (发起连接)、登录服务器 (验证客户资料)、逻辑服务器 (收发网游数据), 有了这样的 test harness, 可以方便地测试连接服务器的正确性, 也可以方便地模拟其他各个服务器断开连接的情况, 看看连接服务器是否应对自如。

同样的思路, 可以为登录服务器写 test harness。(我估计不用为逻辑服务器再写了, 因为肯定已经有自动测试了。)

见 §7.12 的一个具体示例。

多 master 之间的二段提交 这是分布式容错的一个经典做法。用 test harness 能把 primary master 和 secondary masters 单独拎出来测试。在测试 primary master 的时候, test harness 扮演 name service 和 secondary masters。在测试 secondary master 的时候, test harness 扮演 name service、primary master、其他 secondary masters。可以比较容易地测试各种 failure 情况。如果不这么做, 而直接部署多个 masters 来测试, 恐怕很难做到自动化测试。

⁷⁸ http://blog.codingnow.com/2007/02/user_authenticate.html

http://blog.codingnow.com/2006/04/iocp_kqueue_epoll.html

http://blog.codingnow.com/2010/11/go_prime.html

Paxos 的实现 Paxos 协议的实现肯定离不了单元测试，因为涉及多个角色中比较复杂的状态变迁。同时，如果我要写 Paxos 实现，那么 test harness 也是少不了的，它能自动测试 Paxos 节点在真实网络环境下的表现，并且轻松模拟各种 failure 场景。

局限性

如果被测程序有 TCP 之外的 IO，或者其 TCP 协议不易模拟（比如通过 TCP 连接数据库），那么这种测试方案会受到干扰。

对于数据库，如果被测程序只是简单地从数据库 SELECT 一些配置信息，那么或许可以在 test harness 里内嵌一个 in-memory H2 DB engine，然后让被测程序从这里读取数据。当然，前提是被测程序的 DB driver 能连上 H2（或许不是大问题，H2 支持 JDBC 和部分 ODBC）。如果被测程序有比较复杂的 SQL 代码，那么 H2 表现的行为不一定和生产环境的数据库一致，这时候恐怕还是要部署测试数据库（有可能为每个开发人员部署一个小的测试数据库，以免相互干扰）。

如果被测程序有其他 IO（写 log 不算），比如 DataNode 会访问文件系统，那么 test harness 没有能把 DataNode 完整地包裹起来，有些 failure case 不是那么容易测试的。这时或许可以把 DataNode 指向 tmpfs，这样能比较容易地测试磁盘满的情况。当然，这样也有局限性，因为 tmpfs 没有真实磁盘那么大，也不能模拟磁盘读写错误。我不是分布式存储方面的专家，这些问题留给分布式文件系统的实现者去考虑吧。（测试 Paxos 节点似乎也可以用 tmpfs 来模拟 persist storage，由 test case 填充所需的初始数据。）

9.7.5 其他用处

test harness 除了实现 features 的回归测试外，还有别的用处。

加速开发，提高生产力 前面提到，如果有个新功能（增加一种新的 request type）需要改动两个程序，有可能造成相互等待：客户程序 A 说要先等服务程序 B 实现对应的功能响应，这样 A 才能发送新的请求，不然每次请求就会被拒绝，无法测试；服务程序 B 说要先等 A 能够发送新的请求，这样自己才能开始编码与测试，不然都不知道请求长什么样子，也触发不了新写的代码。（当然，这是我虚构的例子。）

如果 A 和 B 都有各自的 test harness，事情就好办了，双方大致商量一个协议格式，然后分头编码。程序 A 的作者在自己的 harness 里边添加一个 test case，模拟他认为 B 应有的响应，这个响应可以 hard code 某种最常见的响应，不必真的实现所需

的判断逻辑（毕竟这是程序 B 的作者该干的事情），然后程序 A 的作者就可以编码并测试自己的程序了。同理，程序 B 的作者也不用等 A 拿出一个半成品来发送新请求，他往自己的 harness 添加一个 test case，模拟他认为 A 应该发送的请求，然后就可以编码并测试自己的新功能了。双方齐头并进，减少扯皮。等功能实现得差不多了，两个程序互相连一连，如果发现协议不一致，检查一下 harness 中的新 test cases（这代表了 A/B 程序对对方的预期），看看哪边改动比较方便，很快就能解决问题。

压力测试 test harness 稍加改进还可以变功能测试为压力测试，供程序员 profiling 用。比如反复不间断发送请求，向被测程序加压。不过，如果被测程序是用 C++ 写的，而 test harness 是用 Java 写的，有可能出现 test harness 占 100% CPU，而被测程序还跑得优哉游哉的情况。这时候可以单独用 C++ 写一个负载生成器。

小结

以单独的进程作为 test harness 对于开发分布式程序相当有帮助，它能达到单元测试的自动化程度和细致程度，又避免了单元测试对功能代码结构的侵入与依赖。

9.8 分布式系统部署、监控与进程管理的几重境界

约定：本节只考虑 Linux 系统，文中涉及的“服务程序”是以 C++ 或 Java 编写的，编译成二进制可执行文件（binary 或 jar），程序启动的时候一般会读取配置文件（或者以其他方式获得配置信息），同一个程序每个服务进程的配置文件可能略有不同。“服务器”这个词有多重含义，为避免混淆，本节以 host 指代服务器硬件，以“服务端程序/进程”指代服务器软件（或者具体说 Web Server 和 Sudoku Solver，这两个都是服务软件）。

在进入正题之前，先看一个虚构但典型的例子：Sudoku Solver。（Sudoku Solver 是个均质的无状态服务，分布式系统中进程的状态迁移不是本节的主题。）

假设你们公司的分布式系统中有一个专门求解数独（Sudoku）的服务程序，这个程序是你们团队开发并维护的。通常 Web Server 会使用这个 Sudoku Solver 提供的服务，用户通过 Web 页面提交一个 Sudoku 谜题，Web Server 转而向 Sudoku Solver 寻求答案。每个 Web Server 会同时跟多个 Sudoku Solver 联系，以实现负载均衡。系统的消息收发关系大致如图 9-16 所示，每个矩形是一个进程，运行在各自的 host 上。

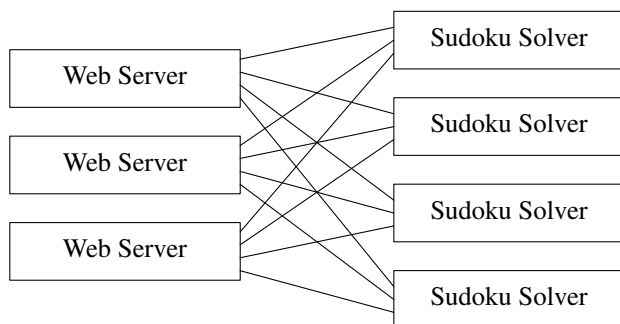


图 9-16

图 9-16 中的 Web Server 请不要简单理解为 `httpd + cgi`，它其实泛指一切客户端，其本身可能是个 `stateful` 的服务程序。

当然，系统不是一开始就是这样的，它经历了多步演化。

1. 最开始的时候，Sudoku 求解直接在 Web Server 内完成。后来为了提高负载能力，把 Sudoku 单独做成服务。一开始系统规模很小，只有一个 Sudoku Solver，也只是一台 Web Server，是个简单的一对一（1:1）的使用关系，如图 9-17 所示。



图 9-17

2. 随后，随着业务量增加，一台 host 不堪重负，于是又部署了几台 Sudoku Solver，变成了一对多（1:N）的使用关系，如图 9-18 所示。

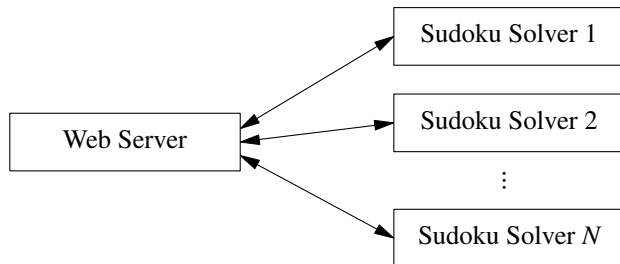


图 9-18

3. 再后来，一台 Web Server 撑不住了，于是部署了几台 Web Server，形成了我们一开始看到的图 9-16 中的多对多（M:N）的使用关系。

在分布式系统中部署并运行 Sudoku Solver，需要考虑以下几个问题：

- Sudoku Solver 如何部署到多台 host 上运行？是把可执行文件拷贝过去吗？程序用到的库怎么办？配置文件怎么办？
- 如何启动服务程序 Sudoku Solver？如果每个 Solver 的配置文件稍有不同（比如每个 Solver 有自己的 service name），那么配置文件是自动生成吗？
- Sudoku Solver 的 listening port 如何配置？如何保证它不与其他服务程序重复？
- 如果程序 crash，谁来重启？能否自动重启？开发/运维人员能否及时收到 alert？
- 如果想主动重启 Sudoku Solver，要不要登录到那台 host 上去 kill？还是能够远程控制？
- 如果要升级 Sudoku Solver 程序，如何重新部署？如何（尽量）做到不中断服务？
- Web Server 如何知道那些 Sudoku Solver 的地址？是不是静态写到 Web Server 的配置文件里？
- 如果 Sudoku Solver 所在的 host 发生硬件故障，管理人员是否能立刻得知这一状况？Web Server 能否自动 failover 到其他 alive 的 Solver 上？
- 部署新的 Sudoku Solver 之后，Web Server 能否自动开始使用新的 Solver 而无需重启？（重启 Web Server 似乎不是大问题，这里我们进一步考虑 client 是个有状态的服务，应该尽量避免无谓的重启。）
- 程序可否安全地退役？比方说公司不再做求解 Sudoku 的业务，那么关闭全部 Sudoku Solver 会不会对其他业务造成影响？

这些问题可以大致归结为几个方面：部署（含升级）可执行文件与配置文件、监控进程状态、管理服务进程，故障响应⁷⁹，这些合起来可称为运维（operation）。

根据公司的规模和技术水平不同，分布式系统的运维分为几重境界，以下是我对各重境界的简要描述。

9.8.1 境界 1：全手工操作

这个大概是高校实验室的水平，分布式系统的规模不大，可能十来台机器上下。分布式系统的实现者为在校学生。

系统完全是手工搭起来的，host 的 IP 地址采用静态配置。

⁷⁹ <http://www.ukuug.org/events/spring2007/programme/ThatCouldntHappenToUs.pdf>

部署 编译之后手工把可执行文件拷贝到各台机器上，或者放到公用的 NFS 目录下。配置文件也手工修改并拷贝到各台机器上（或者放到每个 Sudoku Solver 自己单独的 NFS 目录下）。

管理 手工启动进程，手工在命令行指定配置文件的路径。重启进程的时候需要登录到 host 上并 kill 进程。

升级 如果需要升级 Sudoku Solver，则需要手工登录多台 hosts，可以拷贝新的可执行文件覆盖原来的，并重启。

配置 Web Server 的配置文件里写上 Sudoku Solver 的 ip:port。如果部署了新的 Sudoku Solver，多半要重启 Web Server 才能发挥作用。

监控 无。系统不是真实的商业应用，仅仅用作学习研究，发现哪儿不对劲了就登录到那台 host 上去看看，手工解决问题。

这个级别可算是“过家家”，系统时灵时不灵，可以跑跑测试，发发 paper。

9.8.2 境界 2：使用零散的自动化脚本和第三方组件

这大概是刚起步的公司的水平，系统已经投入商业应用。公司的开发重心放在实现核心业务、添加新功能方面，暂时还顾不上高效的运维，或许系统的运维任务由开发人员或网管人员兼任。公司已经有了基本的开发流程，代码采用中心化的版本管理工具（比如 SVN），有比较正式的 QA sign-off 流程。

公司内网有 DNS，可以把 hostname 解析为 IP 地址，host 的 IP 地址由 DHCP 配置。公司内部的 host 的软硬件配置比较统一，比如硬件都是 x86-64 平台，操作系统统一使用 Ubuntu 10.04 LTS，每天机器上安装的 package 和第三方 library 也是完全一样的（版本号也相同），这样任何一个程序在任何一台 host 上都能启动，不需要单独的配置。

假设各台 host 已经配置好了 SSH authentication key 或者 GSSAPI，不需要手工输入密码。如果要在 host1、host2、host3、host4 上运行 md5sum 命令，看一下各台机器上的 SudokuSolver 可执行文件的内容是否相同，可以在本机执行：

```
for h in host1 host2 host3 host4;
do ssh $h md5sum /path/to/SudokuSolver/version/bin/sudoku-solver ;
done
```

公司的技术人员有能力配置使用 cron、at、logrotate、rrdtool 等标准的 Linux 工具来将部分运维任务自动化。

部署 可执行文件必须经过 QA 签署放行才能部署到生产环境（如有必要，QA 要签署可执行文件的 md5）。为了可靠性，可能不会把可执行文件放到 NFS 上（如果 NFS 发生故障，整个系统就瘫痪了）。有可能采用 rsync 把可执行文件拷贝到本机目录（考虑到可执行文件比较大，估计不适合直接放到版本管理库里），并且用 md5sum 检查拷贝之后的文件是否与源文件相同。部署可执行文件这一步骤应该可以用脚本自动执行⁸⁰。为了让 C++ 可执行文件拷贝到 host 上就能用，通常采用静态链接，以避免.so 版本不同造成故障。

Sudoku Solver 的配置文件会放到版本管理工具里，每个 Solver instance 可能有自己的 branch，每次修改都必须入库。程序启动的时候用的配置文件必须从 SVN 里 check-out，不能手工修改（减少人为错误）。

管理 第一次启动进程的时候，会从 SVN check-out 配置文件；以后重启进程的时候可以从本地 working copy 读取配置文件（以避免 SVN 服务器故障对系统造成影响），只在改过配置文件之后才要求 svn update。服务进程使用 daemon 方式管理（/sbin/init 或 upright 工具），crash 之后会立刻自动重启（利用 respawn 功能）。服务进程一般会随 host 启动而启动（放到 /etc/init.d 里），如果要重启 hostA 上的服务进程，可以通过 SSH 远程操作⁸¹。进程管理是分散的，每台 host 运行哪些 service 完全由本机的 /etc/init.d 目录决定。把一个 service 从一台 host 迁移到另一台 host，需要登录到这两台 host 上去做一些手工配置。

升级 可执行文件也有一套版本管理（不一定通过 SVN），发布新版本的时候严禁覆盖已有的可执行文件。比方说，现在运行的是

```
/path/to/SudokuSolver/1.0.0/bin/sudoku-solver
```

那么新版本的 Sudoku Solver 会发布到

```
/path/to/SudokuSolver/1.1.0/bin/sudoku-solver
```

这么做的原因是，对于 C++ 服务程序，如果在程序运行的时候覆盖了原有的可执行文件，那么可能会在一段时间之后出现 bus error，程序因 SIGBUS 而 crash。另外，如果程序发生 core dump，那么验尸（post mortem）的时候必须用“产生 core dump 的可执行文件”配合 core 文件。如果覆盖了原来的可执行文件，post mortem 将无法进行。

⁸⁰ 比方说 `ssh $host rsync /path/to/source/on/nfs /path/to/local/copy/。`

⁸¹ 比如在本机运行 `ssh hostA /etc/init.d/sudoku-solver restart。`

配置 Web Server 的配置文件里写上 Sudoku Solver 的 host:port（比境界 1 有所提高，这里依赖 DNS，通常 DNS 有一主一备，可靠性足够高）。不过 Web Server 的配置文件和 Sudoku Solver 的配置文件是独立的，如果新增了 Sudoku Solver 或者迁移了 host，除了修改 Sudoku Solver 的配置文件外，还要修改所有用到它的 Web Server 的配置文件。这在系统规模比较小的时候尚且可行，系统规模一大，这种服务之间的依赖关系会变得隐晦。如果关闭了某个服务程序，就可能一不小心造成其他组的某个服务失灵。如孟岩在《通过一个真实故事理解 SOA 监管》⁸² 举的那个例子一样。

监控 公司会使用一些开源的监控工具（以下以 Monit 为例）来监控每台 host 的资源使用情况（内存、CPU、磁盘空间、网络带宽等等）。必要的话可以写一些插件，使之能监控我们自己写的服务程序（Sudoku Solver）。但是这些监控工具通常只是观察者，它们与进程管理工具是独立的，只能看，不能动。这些监控工具有自己的配置文件，这些配置需要与 Sudoku Solver 的配置同步修改。Monit 可以管理进程，但是它判断服务进程是否能正常工作是通过定时轮询进行的，不一定能立刻（几秒之内）发现问题。

在这个境界，分布式系统已经基本可用了，但也有一些隐患。

配置零散

每个服务程序有自己独立的配置，但是整个系统没有全局的部署配置文件（比方说哪个服务程序应该运行在哪些 hosts 上）。

服务程序的配置文件和用到此服务的客户端程序的配置是独立的，如果把 Sudoku Solver 迁移到另一台 host，那么不仅要修改 Sudoku Solver 的配置，还要修改用到 Sudoku Solver 的 Web Server 的配置，以及监控 Sudoku Solver 的 Monit 的配置。如果忘记修改其中的一处，就会造成系统故障。

分布式系统中服务程序的依赖关系是个令人头疼的问题，“依赖”还好办（程序的作者知道自己这个服务程序会依赖哪些其他服务），“被依赖”则比较棘手（如何才能知道停掉自己这个程序会不会让公司其他系统崩溃？）。这也从一个侧面证明使用 TCP 协议作为唯一的 IPC 手段的必要性。如果采用 TCP 通信，为了查出有哪些程序用到了我的 Sudoku Solver（假设 listening port 是 9981），那么我只要运行 `netstat -tpn | grep 9981` 就能找到现在的客户；或者让 Sudoku Solver 自己打印 `accept(2)` log，连续检查一周或者一个月就能知道有哪些程序用到了 Sudoku Solver。

⁸² <http://blog.csdn.net/myan/archive/2007/08/09/1734343.aspx>

进程管理分散

如果 hostA 发生硬件故障，如何能快速地用一台备用服务器硬件顶替它？能否先把它上面原来运行的 Sudoku Solver 迁移到空闲的 hostB 上，然后通知 Web Server 用 hostB 上的 Sudoku Solver？“通知 Web Server”这一步要不要重启 Web Server？

9.8.3 境界 3：自制机群管理系统，集中化配置

这可能是比较成熟的大公司的水平。

境界 2 中的分散式进程管理已经不能满足业务灵活性方面的需求，公司开始整合现有的运维工具，开发一套自己的机群管理软件。我还没有找到一个开源的符合我的要求的机群管理软件，以下虚构一套名为 Zurg⁸³（名字取自科幻电影《第五元素》，拼写稍有不同；Zurg 也是《玩具总动员》中的一个角色。）的分布式系统管理软件⁸⁴。

Zurg 的架构很简单，典型的 Master/Slave 结构，见 §3.5.3 中对“管理 Linux 服务器机群”的描述（见图 9-19）。图 9-19 中矩形为服务器，圆角矩形为进程，实线箭头表示 TCP 连接，虚线表示进程的父子关系。

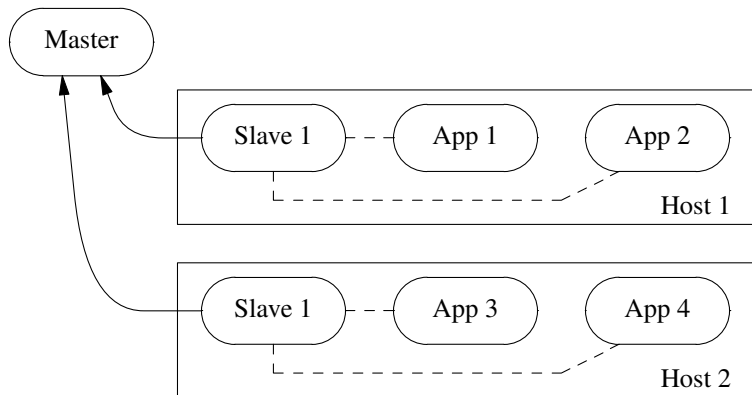


图 9-19

在《分布式系统的工程化开发方法》⁸⁵中谈到了 Zurg 的功能需求：

- 典型的 Master/Slave/Client 结构。
- 一个 Master 进程，兼做 name service。可用冷热备份，或者用 consensus 多点状态同步。如果 Master 意外重启，全部 Slave 都会自动重启。

⁸³ http://en.wikipedia.org/wiki/Google_platform#Software

⁸⁴ Slave 的实现代码见 <http://github.com/chenshuo/muduo-protorpc> 附带的例子。

⁸⁵ <http://blog.csdn.net/Solstice/article/details/5950190>

- 每个节点运行一个 Slave 进程。定期向 Master 汇报该节点的资源使用率，控制其他服务进程的启停，捕获 SIGCHLD 信号，及时知道服务进程（图 9-19 中的 App）意外退出。⁸⁶

到了这一境界，日常的管理运维工作已经不再需要反复执行 ssh，常见任务都可以通过 Zurg 来完成。

部署 只需要向 Master 发一条指令，Master 会命令 Slaves 从指定的地点 rsync 新的可执行文件到本地目录。

进程管理与监控 Zurg 的主要功能就是进程管理和监控，比起一般的开源工具，Zurg 更具备一些优势。由于 Sudoku Solver 是由 Zurg Slave fork(2) 而得的，那么当 Sudoku Solver crash 的时候，Zurg Slave 会立刻收到 SIGCHLD，从而能立刻向管理员报告状态并重启。这比 Monit 的轮询要迅速得多。⁸⁷

为了安全起见，Zurg Slave 在启动可执行文件的时候可以验证其 md5，这样避免错误版本的服务程序运行在生产环境。

Zurg Master 可以提供一个 Web 页面以供查看本机群内各个服务程序是否正常运行，并且提供一个接口（可以是 HTTP）让我们能编写脚本来控制 Zurg Master。

升级 如果要主动重启 Sudoku Solver，可以向 Zurg Master 发出指令，不需要用 ssh & kill。Zurg 会保存每台 host 上服务进程的启动记录，以便事后分析。如果用境界 2 中的手动 /etc/init.d 管理方式，需要到每台机器上收集 log 才知道 Sudoku Solver 什么时候重启过。

另外也可以单独开发 GUI 程序，运行在运维人员的桌面上，重启多台 host 上的 Sudoku Solver 只需要轻点几下鼠标。

配置 零散的配置文件被集中的 Zurg 配置文件取代。

Zurg 配置文件会制定哪些 service 会在哪些 host 上运行，Zurg Master 读取配置文件，然后命令各个 Zurg Slave 启动相应的服务程序。比方说配置文件指定 Sudoku Solver 运行在 host1、host2、host3 上，那么 Zurg 会通知在 host1、host2、host3 上的 Zurg Slave 启动 Sudoku Solver。（当然，每台 host 上的 Zurg Slave 需要由 /etc/init.d 启动，其他的服务程序都由它负责启动。）

⁸⁶ 如果 Slave 意外重启，如何避免重复启动服务？

⁸⁷ 还可以在 fork() 之前做一些手脚，让 Zurg Slave 能更方便地获得 Sudoku Solver 的存活状态。比方说，打开一对 pipe，让子进程继承写端 fd，在父进程中关注读端 fd 的 readable 事件。这样一旦子进程退出，父进程 Zurg Slave 立刻就能读到 EOF，这比用 SIGCHLD signal 更可靠。

更重要的是，服务程序之间的依赖关系在 Zurg 配置文件里直接体现出来。比方说，在 Zurg 配置文件里指明 Web Server 依赖 Sudoku Solver，Web Server 的配置文件由 Zurg Master 生成（可能会用到模板引擎，读入一个 Web Server 的配置模板），其中出现的 Sudoku Solver 的 host:port 由 Zurg Master 自动填上，这样如果把 Sudoku Solver 从 hostA 迁移到 hostB，只需要改一处地方（Zurg 的配置），而 Sudoku Solver 和 Web Solver 的配置都由 Zurg Master 自动生成。这样大大降低了犯错误的机会。

到了这一境界，分布式系统的日常管理已经基本成熟，但在容错与负载均衡方面有较大的提升空间。

目前最大的障碍是 DNS，它限制了快速 failover。比方说，如果 hostA 发生硬件故障，Zurg Master 固然可以在 hostB 上立刻启动 Sudoku Solver，但是如何通知 Web Server 到 hostB 上享用服务呢？修改 DNS entry 的话（把 hostA 的域名解析到 hostB 的 IP），可能要好几分钟才能完成更新，因为 DNS 没有推送机制。

如果思路受限制于 host:port，那么会采取一些看似高级，实则笨拙的高可用（high availability）解决方案。比方说在内核里做做手脚，设法让两台机器共享同一个 IP，然后通过专门的心跳连线来控制哪台 host 对外提供服务，哪台是备用机。如果那台“主机”发生故障，则可以快速（几秒）切换到备用机，因为 hostname 和 IP 地址是相同的，客户端不用重新配置或重启，只要重新连接 TCP 就能完成 failover。如果在错误的道路上走得更远一点，可能还会设法把 TCP 连接一同迁移到备用机，这样客户端甚至不需要断开并重连。

Load balance 也受限于 DNS

如果发现现有的 4 个 Sudoku Solver 不堪重负，又部署了 4 台 Sudoku Solver，如何通知各个 Web Server 把新的 Sudoku Solver 加到连接池里？

有一些 ad hoc 的手段，比方说每个 Web Server 有一个管理接口，可以通过这个接口向它动态地增减 Sudoku Solver 的地址。借助这个管理接口，我们也可以做一些计划中的联机迁移。比方说要主动把某个 Sudoku Solver 从 hostA 迁移到 hostB，我们可以先在 hostB 上启动 Sudoku Solver，然后通过 Web Server 的管理接口把 hostB:9981 添加到 Web Server 的连接池中，再把 hostA:9981 从连接池中删掉，最后停掉 hostA 上的 Sudoku Solver。这对计划中的 Sudoku Solver 升级是可行的，能做到避免中断 Web Server 服务。对于 failover，这种做法似乎稍显不够方便，因为要让 Zurg Master 理解 Web Server 的管理接口，会给系统带来循环依赖。（正常情况下，

Zurg Master 不应该知道或访问它管理的服务程序的接口细节，这样 Sudoku Solver 升级的时候就不用升级 Zurg Master。)

这种做法要求 Web Server 在开发的时候留下适当的维修探查通道，见 §9.5 的推荐做法。

另外一种 ad hoc 的手段，每个 Sudoku Solver 在启动的时候自己主动往某个数据库表里 insert 或 update 本程序的 host:port。Web Server 的配置里写的不是 host:port，而是一条 SELECT 语句，用于找出它依赖的 Sudoku Solver 的 host:port。Web Server 还可以通过数据库触发器来及时获知 Sudoku Solver address list 的变化。这样增加或减少 Sudoku Server 的话，Web Server 几乎可以立刻应对，也不需要通过管理接口来手工增减 Sudoku Solver 地址。数据库在这里扮演了 naming service 的角色，它的可用性直接影响了整个系统的可用性。

境界 3 是黎明前的黑暗，只要统一引入 naming service，抛开 DNS，容错和负载均衡的问题便迎刃而解。

9.8.4 境界 4：机群管理与 naming service 结合

这是业内领先的公司的水平。

前面分析到，使用 Zurg 机群管理软件能大大简化分布式系统的日常运维，但是它也有很大的缺陷——不能实现快速 failover。如果系统规模大到一定程度，机器出故障的频率会显著增加，这时候自动化的快速 failover 是必备的，否则运维人员就会疲于奔命地“救火”。

实现简单而快速的 failover 不需要特殊的编程技巧，也不需要 kernel 动手脚，只要抛弃传统的 DNS 观念，摆脱 host:port 的束缚，采用为分布式系统特制的 naming service 代替 DNS 即可。

naming service 是实现快速 failover 的必备条件。Host A 上的服务 S1 崩溃了，failover 到 Host B 上，如何把新的地址（或端口号）通知给 S1 的使用者？为什么 DNS 不适合？DNS 设计作为静态或缓慢变化的域名解析，DNS 客户端与 DNS 服务器之间采用超时轮询而不是主动通知，不适合快速 failover。DNS 也不能解析端口号⁸⁸。解决办法：实现自己的名字服务，并在程序的配置中使用 service_name 而不是 host:port。例子：Chubby、ZooKeeper、Eureka（<http://techblog.netflix.com/2012/09/eureka.html>）。

naming service 的功能是把一个 service_name 解析成 list of ip:port。比方说，查询 "sudoku_solver"，返回 host1:9981、host2:9981、host3:9981。

⁸⁸ 除非使用不常用的 SRV RR 记录，见 RFC 2782。

naming service 与 DNS 最大的不同在于它能把新的地址信息推送给客户端。比方说, Web Server 订阅了 "sudoku_solver", 每当 sudoku_solver 发生变化, Web Server 就会立刻收到更新。Web Server 不需要轮询, 而是等候通知。

naming service 谁负责更新

在境界 2 中, Sudoku Solver 会自己主动去 naming server 注册。到了境界 3, 由于 Sudoku Solver 是由 Zurg 负责启动的, 那么 Zurg 知道 Sudoku Solver 运行在哪些 hosts 上, 它会主动更新 naming service, 不需要 Sudoku Solver 自己动手。

naming service 的可用性 (availability) 和一致性如何保证

毫无疑问, 一旦采用这种方案, naming service 是系统正常运转的关键, 它的可用性决定了系统的可用性。naming service 绝对不能只 run 在一台服务器上, 为了可靠性, 应该用一组 (通常是 5 台) 服务器同时提供服务。当然, 这需要解决一致性问题。目前实现高可用 naming service 的公认办法是 Paxos 算法, 也有了一些开源的实现 (ZooKeeper、KeySpace、Doozer)。

对程序设计的影响

如果公司的网络库在设计的时候就考虑了 naming service, 那么对程序设计来说是透明的。配置文件里写的不再是 host:port, 而是 service_name, 交给网络库去解析成 ip:port 地址列表。

为什么 muduo 网络库没有封装 DNS 解析

一方面因为 gethostbyname() 和 getaddrinfo() 解析 DNS 是阻塞的 (除非用 UDNS 之类的异步 DNS 库); 另一方面, 因为在大规模分布式系统中 DNS 的作用不大, 我宁愿花时间实现一个 naming service, 并且为它编写 name resolve library。

在境界 3 中, 每个项目组有自己的 hosts, 只运行本项目中的服务程序, 每个服务程序的 TCP 端口可以静态分配 (比如 Sudoku Solver 固定使用 9981 端口), 不用担心端口冲突。如果公司规模继续扩大, 迟早会把 16-bit 的 port 命名空间用完, 这时候给新项目分配端口号将成为问题。

到了境界 4, 这一限制将被打破, 服务程序可以 run 在公司内任何一台 host 上, 也不用担心端口冲突, 因为 Zurg 会选择当前 host 的空闲端口来启动 Sudoku Solver, 并且把选中的端口保存在 naming service 中。这样一来, TCP port 也实现了动态配置, Web Server 完全能自动适应 run 在不同 port 的 Sudoku Solver。

第 10 章

C++ 编译链接模型精要

C++ 从 C 语言¹ 继承了一种古老的编译模型，引发了其他语言中根本不存在的的一些编译方面的问题（比方说“一次定义原则（ODR）²”）。理解这些问题有助于在实际开发中规避各种古怪的错误。

C++ 语言的三大约束是：与 C 兼容、零开销（zero overhead）原则、值语义。§11.7 会具体介绍值语义的话题，下面谈谈第一点“与 C 兼容”。

“与 C 兼容”的含义很丰富，不仅仅是兼容 C 的语法³，更重要的是兼容 C 语言的编译模型与运行模型，也就是说能直接使用 C 语言的头文件和库。比方说对于 connect(2) 这个系统函数⁴，它的头文件和原型如下：

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

C++ 的基本类型的长度和表示（representation）必须和 C 语言一样（int、指针等），准确地说是和编译系统库的 C 语言编译器保持一致。C++ 编译器必须能理解头文件 sys/socket.h 中 struct sockaddr 的定义，生成与 C 编译器完全相同的 layout（包括采用相同的对齐（alignment）算法），并且遵循 C 语言的函数调用约定（参数传递，返回值传递，栈帧管理等等），才能直接调用这个 C 语言库函数。

现代操作系统暴露出的原生接口往往是 C 语言描述的，Windows 的原生 API 接口是 Windows.h 头文件，POSIX 是一堆 C 语言头文件。C++ 兼容 C，从而能在编译的

¹ 本节谈的 C 语言和 C++ 语言指的是现代的常见的实现（没有特别指明时，可认为是 Linux x86-64 的 GCC），并不限于 C 标准或 C++ 标准，因为标准里根本就没有提到“程序库（library）”这个概念。另外本节所提的 C 语言库函数不仅包括 C 标准中的函数，也包括 POSIX 里的常用函数，因为在 Linux 下二者是不分家的，都位于 libc.so。

² http://en.wikipedia.org/wiki/One_Definition_Rule

³ 从兼容语法的角度，Java 和 C# 都可以算是“与 C 兼容”，例如它们的 for 循环写出来都是：

```
for (int i = 0; i < 100; ++i) { /* do something */ }
```

⁴ 本节不区分系统调用与用户态库函数，统称为“系统函数”。

时候直接使用这些头文件，并链接到相应的库上。并在运行的时候直接调用 C 语言的函数库，这省了一道中间层的手续，可算是 C++ 高效的原因之一。

图 10-1 表明了 Linux 上编译一个 C++ 程序的典型过程。其中最耗时间的是 cc1plus 这一步，在一台正在编译 C++ 项目的机器上运行 top(1)，排在首位的往往就是这个进程。

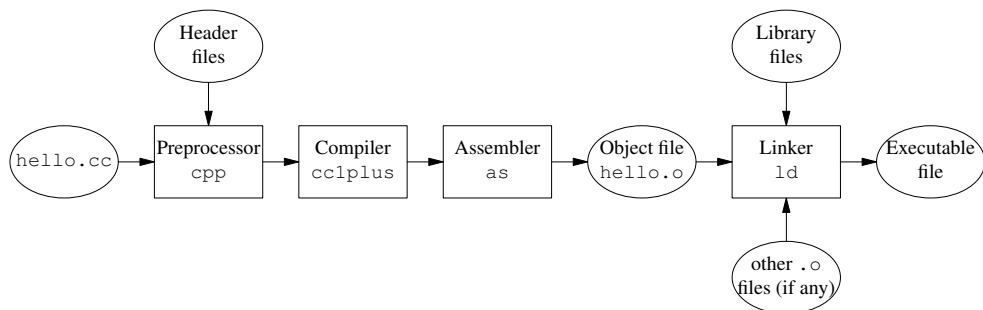


图 10-1

值得指出的是，图 10-1 中各个阶段的界线并不是铁定的。通常 cpp 和 cc1plus 会合并成一个进程；而 cc1plus 和 as 之间既可以以临时文件 (*.s) 为中介，也可以以管道 (pipe) 为中介；对于单一源文件的小程序，往往不必生成 .o 文件。另外，linker 还有一个名字叫做 link editor。

在不同的语境下，“编译”一词有不同的含义。如果笼统地说把 .cc 文件“编译”为可执行文件，那么指的是 preprocessor/compiler/assembler/linker 这四个步骤。如果区分“编译”和“链接”，那么“编译”通常指的是从源文件生成目标文件这几步（即 g++ -c）。如果进一步区分预处理、编译（代码转换）、汇编，那么编译器实际看到的是预处理器完成头文件替换和宏展开之后的源代码⁵。

C++ 至今（包括 C++11）没有模块机制，不能像其他现代编程语言那样用 import 或 using 来引入当前源文件用到的库（含其他 package/module 里的函数或类），而必须用 include 头文件的方式来机械地将库的接口声明以文本替换的方式载入，再重新 parse 一遍。这么做一方面让编译效率奇低，编译器动辄要 parse 几万行预处理之后的源码，哪怕源文件只有几百行；另一方面，也留下了巨大的隐患。部分原因是头文件包含具有传递性，引入不必要的依赖；另一个原因是头文件是在编译时使用，动态库文件是在运行时使用，二者的时间差可能带来不匹配，导致二进制兼容性方面的

⁵ 前面提到，现代编译器通常把预处理和代码转换合并起来，从而让编译器获得更多的信息，调试信息也更丰富。现在的编译器能获知包括宏常量的名字、宏函数等传统上编译器看不到的内容，有的开发环境甚至能单步跟踪宏函数。

问题 (§11.2)。C++ 的设计者 Bjarne Stroustrup 自己很清楚这一点⁶，但这是在“与 C 兼容”的大前提下不得不做出的妥协。

比如有一个简单的小程序，只用了 `printf(3)`，却不得不包含 `stdio.h`，把其他不相关的函数、`struct` 定义、宏、`typedef`、全局变量等等也统统引入到当前命名空间。在预处理的时候会读取近 20 个头文件，预处理之后供编译器 `parse` 的源码有近千行（这还算是短的）。

```
// hello.cc           # 一个简单的源文件
#include <stdio.h>     # 只包含了一个头文件
```

```
int main()
{
    printf("hello preprocessor\n");
}
```

```
$ gcc -E hello.cc |wc # 预处理之后有 942 行
942      2164      17304
```

```
$ strace -f -e open cpp hello.cc -o /dev/null 2>&1 |grep -v ENOENT|awk '{print $3}'
# 省略无关内容。另外我不知道 cpp 有没有直接输出以下内容的命令行选项，只好用笨办法。
```

```
open("hello.cc",
open("/usr/include/stdio.h",
open("/usr/include/features.h",
open("/usr/include/bits/predefs.h",
open("/usr/include/sys/cdefs.h",
open("/usr/include/bits/wordsize.h",
open("/usr/include/gnu/stubs.h",
open("/usr/include/bits/wordsize.h",
open("/usr/include/gnu/stubs-64.h",
open("/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h",
open("/usr/include/bits/types.h",
open("/usr/include/bits/wordsize.h",
open("/usr/include/bits/typesizes.h",
open("/usr/include/libio.h",
open("/usr/include/_G_config.h",
open("/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h",
open("/usr/include/wchar.h",
open("/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stdarg.h",
open("/usr/include/bits/stdio_lim.h",
open("/usr/include/bits/sys_errlist.h",
```

读者若有兴趣，可将其中的 `stdio.h` 替换为 C++ 标准库的头文件 `complex`，看看预处理之后的源代码有多少行，额外包含了哪些头文件。（在我的机器上测试，预处理之后有 21 879 行，包含了近 150 个头文件，包括 `<string>`、`<sstream>` 等大块头。）

值得一提的是，为了兼容 C 语言，C++ 付出了很大的代价。例如要兼容 C 语言的隐式类型转换规则（例如整数类型提升），这让 C++ 的函数重载决议（`overload`

⁶《C++ 语言的设计和演化》的第 18 章“C 语言预处理器”——C++ 必须被摧毁。

resolution) 规则变得无比复杂⁷。另外 class 定义式后面那个分号也不晓得谋杀了多少初学者的时间, 这是为了与 C struct 语法兼容, 因为 C 允许在函数返回类型处定义新 struct 类型, 因此分号是必需的。Bjarne Stroustrup 自己也说“我又不是不懂如何设计出比 C++ 更漂亮的语言⁸”。(由于 C 语言没有函数重载, 也就不存在重载决议, 所以隐式类型转换的危害没有体现在这一方面。)

10.1 C 语言的编译模型及其成因

要想了解 C 语言的编译模型的成因, 我们需要略微回顾一下 Unix 的早期历史⁹。1969 年 Ken Thompson 用汇编语言在一台闲置的 PDP-7 小型机上写出了 Unix 的史前版本¹⁰。值得一提的是, PDP-7 的字长是 18-bit¹¹, 只能按字 (word) 寻址, 不支持今日常见的按 8-bit 字节寻址。假如 C 语言诞生在 PDP-7 上, 计算机软硬件的发展史恐怕要改写。

1970 年 5 月, Ken Thompson 和 Dennis Ritchie 所在的贝尔实验室下订单购买了一台 PDP-11 小型机, 这是 1970 年 1 月刚刚上市的新机型。PDP-11 的字长是 16-bit, 可以按 8-bit 字节寻址, 这可谓一举奠定了今后 C 语言及硬件的发展道路¹²。这台机器的主机 (处理器和内存) 当年夏天就到货了, 但是硬盘直到 1970 年 12 月才到货。

1971 年, Ken Thompson 把原来运行在 PDP-7 上的 Unix 用 PDP-11 汇编靠人力重写了一遍, 运行在这台 PDP-11/20 机器上。这台机器一共只有 24KiB 内存¹³, 其中 16KiB 运行操作系统, 8KiB 运行用户代码¹⁴; 硬盘一共只有 512KiB, 文件大小限制为 64KiB。然后实现了一个文本处理器, 用于排版贝尔实验室的专利申请, 这是购买这台计算机的正经用途。

⁷ C++ 复杂的作用域机制也大大增加了函数重载决议的难度, 基本上只有 C++ 编译器才弄得清楚 (http://en.wikipedia.org/wiki/C%2B%2B#Parsing_and_processing_C.2B.2B_source_code)。

⁸ “Even I knew how to design a prettier language than C++.” — Bjarne Stroustrup

⁹ 主要参考 <http://minnie.tuhs.org/cgi-bin/utree.pl> 和 Dennis M. Ritchie 写的《The Evolution of the Unix Time-sharing System》一文 (<http://cm.bell-labs.com/cm/cs/who/dmr/hist.pdf>)。

¹⁰ Unix 的历史一般从 1970 年算起 (Unix Epoch 是 1970-01-01 00:00:00 UTC), 因此这个只能算“史前”。

¹¹ <http://en.wikipedia.org/wiki/PDP-7> <http://en.wikipedia.org/wiki/PDP-11>

¹² 在 C 语言 20 世纪 70 年代开始流行之后, 高效支持 C 语言就成了 CPU 指令集的设计目标之一, 否则这种 CPU 很难推广。另外, C/Unix/Arpanet 还规范了字节的长度, 在此之前, 字节可以是 6、7、8、9、12 比特, 之后都是 8-bit, 否则就不能与其他系统联网通信 (<http://herbsutter.com/2011/10/12/dennis-ritchie/>)。

¹³ 用的是磁芯存储器 (http://en.wikipedia.org/wiki/Magnetic-core_memory), 因此早期文献常以 core 指代内存。

¹⁴ PDP-11/20 是 PDP-11 系列的第一个型号, 甚至没有内存保护机制, 也就没法区分核心态和用户态。

下面的 Unix 历史多半发生在另外一台内存和硬盘都更大的 PDP-11 机器上，型号可能是 PDP-11/40 或 PDP-11/45。（不同的权威文献说法不一，可能不止一台。）

1972 年是 C 语言历史上最为关键的一年¹⁵，这一年 C 语言加入了预处理，具备了编写大型程序的能力（理由见下文）。到了 1973 年初，C 语言基本定型，主要新特性是支持结构体。此时 C 语言的编译模型已经基本定型，即分为预处理、编译、汇编、链接这四个步骤，沿用至今。

1973 年是 Unix 历史上关键的一年，这一年夏天，二人把 Unix 的内核用 C 语言重写了一遍，完成了用高级语言编写操作系统的伟大创举。（Thompson 在 1972 年就尝试过用 C 重写 Unix 内核，但是当时的 C 语言不支持结构体，因此他放弃了。）

随后，1974 年，Dennis Ritchie 和 Ken Thompson 发表了经典论文《The UNIX Time-Sharing System》¹⁶。除了没有函数原型声明外，1974 年的 C 代码¹⁷读起来跟现在的 C 程序基本无区别。

10.1.1 为什么 C 语言需要预处理

了解了 C 语言的诞生背景，我们可以归纳 PDP-11 上的第一代 C 编译器的硬性约束：内存地址空间只有 16-bit，程序和数据必须挤在这狭小的 64KiB 空间里，可谓捉襟见肘¹⁸。注意，本节提到的 C 语言甚至早于 1978 年的 K&R C，是 20 世纪 70 年代最初几年的原始 C 语言。

编译器没办法在内存里完整地表示单个源文件的抽象语法树¹⁹，更不可能把整个程序（由多个源文件组成）放到内存里，以完成交叉引用（不同源文件的函数之间相互调用，使用外部变量等等）。由于内存限制，编译器必须要能分别编译多个源文件，生成多个目标文件，再设法把这些目标文件组合（链接²⁰）为一个可执行文件。

¹⁵ 《The Development of the C Language》Dennis M. Ritchie. (<http://cm.bell-labs.com/cm/cs/who/dmr/chist.pdf>)。

¹⁶ <http://cm.bell-labs.com/cm/cs/who/dmr/cacm.pdf>：这篇文章至少有三个版本，第一版以单页摘要的形式发表于 1973 年 10 月的第四届 ACM SOSOP 会议上，第二版发表于 1974 年 7 月的 *Communications of the ACM* 期刊上，第三版发表于 1978 年七-八月的 BSTJ 上。此处链接是第三版，内容与 CACM 的原始版本略有出入。

¹⁷ Unix V5 的 C 编译器源码：<http://minnie.tuhs.org/cgi-bin/utree.pl?file=V5/usr/c>。

¹⁸ PDP-11 的物理内存可以有几百 KiB，但是每个进程只能看到 16-bit 的地址空间。PDP-11/45 支持将代码空间和数据空间分离（即哈佛结构，而非冯诺依曼结构），各自有 64KiB。但是直到 1979 年的 Unix V7 才用上这个功能，而此时 C 语言早已定型（http://en.wikipedia.org/wiki/PDP-11_architecture）。

¹⁹ 我怀疑当时的 C 编译器恐怕连整个函数都无法放到内存里，只能放下当前的表达式。

²⁰ 其实链接器的历史比编译器还长，在没有高级语言编译器而只有汇编器的时代，链接器就已经存在。我们可以把多个汇编源文件 assemble 成目标文件，再让链接器来处理外部符号的地址与函数重定位。

在今天看来，C 语言这种支持把一个程序分成多个源文件的“功能”几乎是顺理成章的。但是在当时而言，并不是每个语言都有意地做到这一点。我们以同一时期（1968–1974）Niklaus Wirth 设计的 Pascal 语言为对照。Pascal 语言可以定义函数和结构体，也支持指针，语法也比当时的 C 语言更优美。但它长期没有官方规定²¹的多源文件模块化机制，它要求每个程序（program）必须位于同一个源文件²²，这其实大大限制了它在系统编程方面的用途²³。如果 Pascal 一早就克服这些缺点²⁴，“那么我们今天很可能要把 begin 和 end 直接映射到键盘上。”²⁵

或许是受内存限制，一个可执行程序不能太大，Dennis Ritchie 编写的 PDP-11 C 编译器不是一个可执行文件，而是 7 个可执行文件²⁶：cc、cpp、as、ld、c0、c1、c2²⁷。其中 cc 是个 driver，用于调用另外几个程序。cpp 是预处理器（Unix V7 从 c0 分离出来），当时叫做 compiler control line expander。c0、c1、c2 是 C 编译器的三个阶段（phase）²⁸，c0 的作用是把源程序编译为两个中间文件；c1 把中间文件编译为汇编源代码；c2 是可选的，用于对生成汇编代码做窥孔优化。as 是汇编器，把汇编代码转换为目标文件。ld 是链接器，把目标文件和库文件链接成可执行文件。编译流程见图 10-2。不用 cc，手工编译一个简单程序 prog.c 的过程如下：

```
/lib/cpp prog.c > prog.i      # prog.i 是预处理之后的源代码
/lib/c0 prog.i temp1 temp2    # c0 生成 temp1 和 temp2 这两个中间文件
/lib/c1 temp1 temp2 prog.s    # c1 读入 temp1 和 temp2，生成汇编代码 prog.s
as - prog.s                  # 把 prog.s 汇编为目标文件 a.out。猜猜 a.out 的原意？
ld -n /lib/crt0.o a.out -lc   # 把 a.out 链接为可执行文件
当时的链接器是单向查找未决符号，因此要把 crt0.o 放到 a.out 之前，-lc 必须放到末尾
```

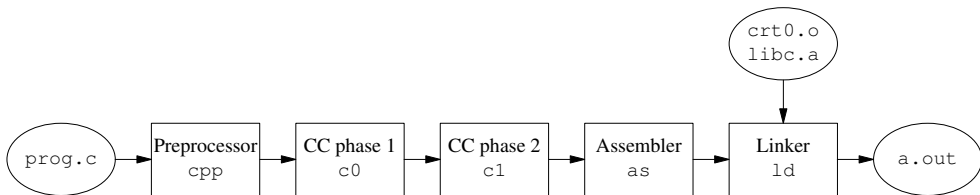


图 10-2

²¹ 《PASCAL - User Manual and Report》Springer-Verlag, 1974.

²² Donald Knuth 写的 TeX 就是一个 20000 多行的单源文件 Pascal 大程序。

²³ Niklaus Wirth 最初的设计目的是让 Pascal 成为结构化编程的教学语言。

²⁴ 《Why Pascal is Not My Favorite Programming Language》Brian W. Kernighan.
(<http://www.lysator.liu.se/c/bwk-on-pascal.html>)

²⁵ 孟岩的《C++ 开源程序库评话》(<http://blog.csdn.net/myan/article/details/679007>)。

²⁶ 《Regenerating System Software》(http://minnie.tuhs.org/PUPS/Setup/v7_regen.html)

²⁷ 在 Unix V5 中 c[012] 的源代码一共有 6100 行，在 Unix V6 中一共有 8000 行。

²⁸ 《A Tour through the UNIX C Compiler》Dennis M. Ritchie. (<http://plan9.bell-labs.com/7thEdMan/v7vol2b.pdf>)

为了能在尽量减少内存使用的情况下实现分离编译，C 语言采用了“隐式函数声明（implicit declaration of function）”的做法。代码在使用前文未定义的函数时，编译器不需要也不检查函数原型²⁹：既不检查参数个数，也不检查参数类型与返回值类型。编译器认为未声明的函数都返回 int，并且能接受任意个数的 int 型参数。而且早期的 C 语言甚至不严格区分指针和 int，而是认为二者可以相互赋值转换。在 C++ 程序员看来，这是毫无安全保障的做法，但是 C 语言就是如此地相信程序员。

举例解释一下什么是“隐式函数声明”。

```
// hello.c
int main()                # 这个程序没有引用任何头文件
{
    printf("hello C.\n"); # 隐式声明 int printf(...);
    return 0;
}

$ gcc hello.c -Wall        # 用 gcc 可以编译运行通过
hello.c: In function 'main':
hello.c:3: warning: implicit declaration of function 'printf'
hello.c:3: warning: incompatible implicit declaration of built-in function 'printf'

$ g++ hello.c -Wall        # 用 g++ 则会报错
hello.c: In function 'int main()':
hello.c:3: error: 'printf' was not declared in this scope
```

如果 C 程序用到了某个没有定义的函数（可能错误拼写了函数名），那么实际造成的是链接错误（undefined reference），而非编译错误。例如：

```
// undefined.c
int main()
{
    helloworld();          # 隐式声明 helloworld
    return 0;
}

$ gcc undefined.c -Wall
undefined.c: In function 'main':
undefined.c:3: warning: implicit declaration of function 'helloworld'
/tmp/ccHUCGat.o: In function 'main':
undefined.c:(.text+0xa): undefined reference to `helloworld'
collect2: ld returned 1 exit status    # 真正报错的是 ld，不是 cc1
```

其实，有了隐式函数声明，我们已经能分别编译多个源文件，然后把它们链接为一个大的可执行文件（此处指的是编译出来有几十 KiB 的程序）。那么为什么还需要头文件和预处理呢？

根据 Eric S. Raymond 在《The Art of Unix Programming》第 17.1.1 节³⁰引用

²⁹ C 语言的函数原型是 20 世纪 80 年代才从 C++ 借用过来的，算是 C++ 对 C 的反哺。

³⁰ http://www.faqs.org/docs/artu/c_evolution.html

Steve Johnson 的话，最早的 Unix 是把内核数据结构（例如 `struct dirent`）打印在手册上，然后每个程序自己在代码中定义 `struct`。例如 Unix V5 的 `ls(1)` 源码³¹中就自行定义了表示目录的结构体。有了预处理和头文件，这些公共信息就可以做成头文件放到 `/usr/include`，然后程序包含用到的头文件即可。减少无谓错误，提高代码的可移植性。

最早的预处理只有两项功能：`#include` 和 `#define`。`#include` 完成文件内容替换，`#define` 只支持定义宏常量，不支持定义宏函数。早期的头文件里只放三样东西：`struct` 定义，外部变量³²的声明，宏常量。这样可以减少各个源文件里的重复代码。

到目前为止，头文件的预处理的作用都还是正面的。在谈头文件与预处理的害处之前，让我把 PDP-11 的 16-bit 地址空间对 C 语言及其编译模型的影响讲完。

10.1.2 C 语言的编译模型

由于不能将整个源文件的语法树保存在内存中，C 语言其实是按“单遍编译（one pass）³³”来设计的。所谓单遍编译，指的是从头到尾扫描一遍源码，一边解析（parse）代码，一边即刻生成目标代码。在单遍编译时，编译器只能看到目前（当前语句/符号之前）已经解析过的代码，看不到之后的代码，而且过眼即忘。这意味着

- C 语言要求结构体必须先定义，才能访问其成员，否则编译器不知道结构体成员的类型和偏移量，就无法立刻生成目标代码。
- 局部变量也必须先定义再使用，因为如果把定义放到后面，编译器在第一次看到一个局部变量时并不知道它的类型和在 `stack` 中的位置，也就无法立刻生成代码，只能报错退出。
- 为了方便编译器分配 `stack` 空间，C 语言要求局部变量只能在语句块的开始处定义。
- 对于外部变量，编译器只需要知道它的类型和名字，不需要知道它的地址，因此需要先声明后使用。在生成的目标代码中，外部变量的地址是个空白，留给链接器去填上。
- 当编译器看到一个函数调用时，按隐式函数声明规则，编译器可以立刻生成调用函数的汇编代码（函数参数入栈、调用、获取返回值），这里唯一尚不能确定的是函数的实际地址，编译器可以留下一个空白给链接器去填。

³¹ <http://minnie.tuhs.org/cgi-bin/utree.pl?file=V5/usr/source/s1/ls.c>（注意 `readdir()` 函数）。

³² 或者叫全局变量，如果不那么学术的话。

³³ http://en.wikipedia.org/wiki/One-pass_compiler

对 C 编译器来说，只需要记住 `struct` 的成员和偏移，知道外部变量的类型，就足以一边解析源代码，一边生成目标代码。因此早期的头文件和预处理恰好满足了编译器的需求。外部符号（函数或变量）的决议（`resolution`）可以留给链接器去做³⁴。

从上面的编译过程可以发现，C 编译器可以做得很小，只使用很少的内存。据我观察，Unix V5 的 C 编译器甚至没有使用动态分配内存，而是用一些全局的栈和数组来帮助处理复杂表达式和语句嵌套，整个编译器的内存消耗是固定的³⁵。（我推测 C 语言不支持在函数内部嵌套定义函数也是受此影响，因为这样一来意味着必须用递归才能解析函数体，编译器的内存消耗就不是一个定值。）

受“不能嵌套”的影响，整个 C 语言的命名空间是平坦的（`flat`），函数和 `struct` 都处于全局命名空间。这其实给 C 程序员带来了不少麻烦，因为每个库都要设法避免自己的函数和 `struct` 与其他库冲突。早期 C 语言甚至不允许在不同 `struct` 中使用相同的成员名称³⁶，因此我们看到一些 `struct` 的名字有前缀，例如 `struct timeval` 的成员是 `tv_sec` 和 `tv_usec`，`struct sockaddr_in` 的成员是 `sin_family`、`sin_port`、`sin_addr`。

讲清楚了 C 语言的编译模型，我们再来看看它对 C++ 的影响（和伤害）。

10.2 C++ 的编译模型

由于要保持与 C 兼容，原本很多在 C 语言中顺理成章或者危害不大的东西继承到了 C++ 里就成了大祸害³⁷。

10.2.1 单遍编译

C++ 也继承了单遍编译。在单遍编译时，编译器只能根据目前看到的代码做出决策，读到后面的代码也不会影响前面做出的决定。这特别影响了名字查找（`name lookup`）和函数重载决议。

³⁴ 链接器的主要作用之一其实就是填空，见 [LLL] 和 [ExpC] 等书的有关章节。

³⁵ 这意味着 Unix V5 的 C 编译器不能处理太复杂的表达式，编译器也确实有对“`Expression overflow`”的错误处理。

³⁶ 即 `struct` 的成员名称是全局的。其实不是不允许，而是相同名字的成员的类型和其在各自 `struct` 内的偏移必须也相同。

³⁷ 前面已经讲过隐式类型转换对函数重载决议的影响。

先说**名字查找**，C++ 中的名字包括类型名、函数名、变量名、typedef 名、template 名等等。比方说对下面这行代码

```
Foo<T> a;    # Foo、T、a 这三个名字都不是 macro
```

如果不知道 Foo、T、a 这三个名字分别代表什么，编译器就无法进行语法分析。根据之前出现的代码不同，上面这行语句至少有三种可能性：

1. Foo 是个 `template<typename X> class Foo;`，T 是 type，那么这句话以 T 为模板类型参数类型具现化了 `Foo<T>` 类型，并定义了变量 a。
2. Foo 是个 `template<int X> class Foo;`，T 是 `const int` 变量，那么这句话以 T 为非类型模板参数具现化了 `Foo<T>` 类型，并定义了变量 a。
3. Foo、T、a 都是 int，这句话是个没啥用的表达式语句。

别忘了 `operator<()` 是可以重载的，这句简单代码还可以表达别的意思^{38 39}。另外一个经典的例子是 `AA BB(CC);`，这句话既可以声明函数，也可以定义变量。

C++ 只能通过解析源码来了解名字的含义，不能像其他语言那样通过直接读取目标代码中的元数据来获得所需信息（函数原型、class 类型定义等等）。这意味着要想准确理解一行 C++ 代码的含义，我们需要通读这行代码之前的所有代码，并理解每个符号（包括操作符）的定义。而头文件的存在使得肉眼观察几乎是不可能的。完全有可能出现一种情况：某人不经意改变了头文件，或者仅仅是改变了源文件中头文件的包含顺序，就改变了代码的含义，破坏了代码的功能。这时能造成编译错误已经是谢天谢地了。

C++ 编译器的符号表至少要保存目前已看到的每个名字的含义，包括 class 的成员定义、已声明的变量、已知的函数原型等，才能正确解析源代码。这还没有考虑 template，编译 template 的难度超乎想象。编译器还要正确处理作用域嵌套引发的名字的含义变化：内层作用域中的名字有可能遮住（shadow）外层作用域中的名字。有些其他语言会对此发出警告，对此我建议用 g++ 的 `-Wshadow` 选项来编译代码。（插一句题外话：muduo 的代码都是 `-Wall -Wextra -Werror -Wconversion -Wshadow` 编译的。）

再说**函数重载决议**，当 C++ 编译器读到一个函数调用语句时，它必须（也只能）从目前已看到的同名函数中选出最佳函数。哪怕后面的代码中出现了更合适的匹配，

³⁸ 有兴趣的话可以读一读陈皓写的《恐怖的 C++ 语言》一文（<http://coolshell.cn/articles/1724.html>）。

³⁹ 更多的例子见 [D&E] 的 6.3.1 节。

也不能影响当前的决定⁴⁰。这意味着如果我们交换两个 namespace 级的函数定义在源代码中的位置，那么有可能改变程序的行为。

比方说对于如下一段代码：

```
void foo(int)
{
    printf("foo(int);\n");
}

void bar()
{
    foo('a'); // 调用 foo(int)
}

void foo(char)
{
    printf("foo(char);\n");
}

int main()
{
    bar();
}
```

如果有人在重构的时候把 void bar() 的定义挪到 void foo(char) 之后，程序的输出就不一样了。

这个例子充分说明实现 C++ 重构工具的难度：重构器对代码的理解必须达到编译器的水准，才能在修改代码时不改变原意。函数的参数可以是个复杂表达式，重构器必须能正确解析表达式的类型才能完成重载决议。比方说 foo(str[0]) 应该调用哪个 foo() 跟 str[0] 的类型有关，而 str 可能是个 std::string，这就要求重构器能正确理解 template 并具现化之。C++ 至今没有像样的重构工具，恐怕正是这个原因。

C++ 编译器必须在内存中保存函数级的语法树，才能正确实施返回值优化 (RVO)⁴¹，否则遇到 return 语句的时候编译器无法判断被返回的这个对象是不是那个可以被优化的 named object⁴²。

其实由于 C++ 新增了不少语言特性，C++ 编译器并不能真正做到像 C 那样过眼即忘的单遍编译。但是 C++ 必须兼容 C 的语意，因此编译器不得不装得好像是单遍编译（准确地说是单遍 parse）一样，哪怕它内部是 multiple pass 的⁴³。

⁴⁰ 对于 class 成员函数有一个例外，编译器总是先扫描一遍 class 定义，再来处理其中的成员函数，因此全部同名成员函数都参与重载决议。

⁴¹ http://en.wikipedia.org/wiki/Return_value_optimization

⁴² Visual C++ 直到 2005 年才实现 RVO ([http://msdn.microsoft.com/en-us/library/ms364057\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms364057(VS.80).aspx))。

⁴³ C++ 允许 forward reference，因此几乎肯定做不到 one pass (http://en.wikipedia.org/wiki/Forward_declaration)。

10.2.2 前向声明

几乎每份 C++ 编码规范^{44 45 46}都会建议尽量使用前向声明来减少编译期依赖，这里我用“单向编译”来解释一下这为什么是可行的，很多时候甚至是必需的。

如果代码里调用了函数 `foo()`，C++ 编译器 `parse` 此处函数调用时，需要生成函数调用的目标代码。为了完成语法检查并生成调用函数的目标代码，编译器需要知道函数的参数个数和类型以及函数的返回值类型，它并不需要知道函数体的实现（除非要做 `inline` 展开）。因此我们通常把函数原型放到头文件里，这样每个包含了此头文件的源文件都可以使用这个函数。这是每个 C/C++ 程序员都明白的事情。

当然，光有函数原型是不够的，程序其中某一个源文件应该定义这个函数，否则会造成链接错误（未定义的符号）。这个定义 `foo()` 函数的源文件通常也会包含 `foo()` 的头文件。但是，假设在定义 `foo()` 函数时把参数类型写错了，会出现什么情况？

```
// in foo.h
void foo(int);           // 原型声明

// in foo.cc
#include "foo.h"

void foo(int, bool)      // 在定义的时候必须把参数列表和返回类型抄一遍。
{                          // 有抄错的可能，也可能将来改了一处，忘了改另一处
    // do something
}
```

编译 `foo.cc` 会有错吗？不会，因为编译器会认为 `foo` 有两个重载。但是链接整个程序会报错：找不到 `void foo(int)` 的定义。你有没有遇到过类似的问题？

这是 C++ 的一种典型缺陷，即一样东西区分声明和定义，代码放到不同的文件中，这就有出现不一致的可能性。C/C++ 里很多稀奇古怪的错误就源自于此，比如 [ExpC] 举的一个经典例子：在一个源文件里声明 `extern char* name`，在另一个源文件里却定义成 `char name[] = "Shuo Chen";`。

对于函数的原型声明和函数体定义而言，这种不一致表现在参数列表和返回类型上，编译器通常能查出参数列表不同，但不一定能查出返回类型不同，见后文 p. 406。也可能参数类型相同，但是顺序调换了。例如原型声明为 `draw(int height, int width)`，定义的时候写成 `draw(int width, int height)`，编译器无法查出此类错误，因为原型声明中的变量名是无用的。

⁴⁴ [EC3] 第 31 条，[CCS] 第 22 条。

⁴⁵ LLVM 编程规范（http://llvm.org/docs/CodingStandards.html#hl_dontinclude）。

⁴⁶ Google 编程规范（http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Header_File_Dependencies）。

其他语言似乎没有这个问题。例如我们不需要在 Java 里使用函数原型声明，一个成员函数的参数列表只需要在代码里出现一次，不存在不一致的可能。Java 编译器也不受“单遍编译”的约束，调整成员函数的顺序不会影响代码语义。Java 也没有笨重过时的头文件包含机制，而是有一套基于 `package` 的模块化机制，陷阱少得多。

如果要写一个库给别人用，那么通常要把接口函数的原型声明放到头文件里。但是在写库的内部实现的时候，如果没有出现函数相互调用⁴⁷的情况，那么我们可以适当组织函数定义的顺序，让基础函数出现在代码的前面，这样就不必前向声明函数原型了。参见云风的一篇博客⁴⁸。

函数原型声明可以看作是对函数的前向声明（`forward declaration`），除此之外我们还常常用到 `class` 的前向声明。

有些时候 `class` 的前向声明是必需的，例如 p. 484 出现的 `Child` 和 `Parent class` 相互指涉的情况⁴⁹。有些时候 `class` 的完整定义是必需的^[CCS, 条款 22]，例如要访问 `class` 的成员，或者要知道 `class` 的大小以便分配空间。其他时候，有 `class` 的前向声明就足够了，编译器只需要知道有这么个名字的 `class`。

对于 `class Foo`，以下几种使用不需要看见其完整定义：

- 定义或声明 `Foo*` 和 `Foo&`，包括用于函数参数、返回类型、局部变量、类成员变量等等。这是因为 C++ 的内存模型是 `flat` 的，`Foo` 的定义无法改变 `Foo` 的指针或引用的含义。
- 声明一个以 `Foo` 为参数或返回类型的函数，如 `Foo bar()` 或 `void bar(Foo f)`，但是，如果代码里调用这个函数就需要知道 `Foo` 的定义，因为编译器要使用 `Foo` 的拷贝构造函数和析构函数，因此至少要看到它们的声明（虽然构造函数没有参数，但是有可能位于 `private` 区）。

`muduo` 代码中大量使用前向声明来减少 `include`，并且避免把内部 `class` 的定义暴露给用户代码。

[CCS] 第 30 条规定不能重载 `&&`、`||`、`,` (逗号) 这三个操作符，Google 的 C++ 编程规范补充规定⁵⁰ 不能重载一元 `operator&`（取址操作符），因为一旦重载 `operator&`，这个 `class` 的就不能用前向声明了。例如：

⁴⁷ http://en.wikipedia.org/wiki/Mutual_recursion

⁴⁸ <http://blog.codingnow.com/2007/06/kiss.html>

⁴⁹ 同一页出现的 Java 代码也没有前向声明，说明 Java 编译器能同时看到多个源文件的代码。

⁵⁰ http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Operator_Overloading

```
class Foo; // 前向声明

void bar(Foo& foo)
{
    Foo* p = &foo; // 这句话是取 foo 的地址，但是如果重载了 &，意思就变了。
}
```

代码的行为跟是否 include Foo 的完整定义有关，等于埋了“定时炸弹”。

10.3 C++ 链接 (linking)

链接 (linking) 这个话题可以单独写一本书 [LLL]，这本书讲“C++ 链接”的有第 4.4 节“静态链接/C++ 相关问题”和第 9.4 节“C++ 与动态链接⁵¹”等章节。

本节重点介绍与 C++ 日常开发相关的链接方面的问题，先以手工编制一本书的目录和交叉索引为例，介绍链接器的基本工作原理^{52 53 54}。假设一个作者写完了十多个章节，你的任务是把这些章节编辑为一本书。每个章节的篇幅不等，从 30 页到 80 页都有，都已经分别排好版打印出来。（已经从源文件编译成了目标文件。）

章节之间有交叉引用，即正文里会出现“请参考 XXX 页的第 YYY 节”等字样。作者在撰写每个章节的时候并不知道当前文字的章节号，当然也不知道当前文字将来会出现在哪一页上。因为他可以随时调整章节顺序、增减文字内容，这些举动会影响最终的章节编号和页码。为了引用其他章节的内容，作者会在文字中放 anchor (L^AT_EX 是 \label)，给需要被引用的文字命名。比方说本章“C++ 编译链接模型精要”的名字是 ch:cppCompilation。（这就好比给全局函数或全局变量起了一个独一无二的名字。）在引用其他章节的编号或页码时，作者在正文中留下一个适当的空白，并注明这里应该填上的某个 anchor 的页码或章节编号 (L^AT_EX 是 \ref{ch:cppCompilation})。

现在你拿到了这十几摞打印的文稿，怎么把它们编辑成一本书呢？你可能会想到下面这两个步骤：先编排页码和章节编号，再解决交叉引用。第一步：

- 1a. 把这些文稿按章的先后顺序叠好，这样就可以统一编制正文页码。
- 1b. 在编制页码的同时，章节号也可以一并确定下来。

在进行 1a 和 1b 这个步骤时，你可以同时顺序记录两张纸：

⁵¹ 不过我对用 C++ 编写动态链接库有自己的看法，见 §11.3 和 §11.4。

⁵² [ExpC] 第 5 章，[CS:APP] 第 7 章，[LLL] 第 4 章。

⁵³ 《Linkers and Loaders》(<http://www.linuxjournal.com/article/6463>)。

⁵⁴ https://events.linuxfoundation.org/images/stories/pdf/lfcs2012_ccoutant.pdf

- 章节的编号、标题和它出现的页码，用于编制目录。
- 遇到 `anchor` 时，记下它的名字和出现的页码、章节号，用于解决交叉引用。

如果按上面的办法来操作，解决交叉引用就不难了。第二步：

2. 再从头翻一遍书稿，遇到空白的交叉应用，就到 `anchor` 索引表里查出它的页码和章节编号，填上空白。

至此，如果一切顺利的话，书籍编辑任务完成。请读者思考，为什么书的正文页码用阿拉伯数字，而前言和目录的页码通常是罗马数字？如果整本书从头到尾连续编排页码，手工处理会遇到什么困难？

在这项工作中最容易出现以下两种意外情况，也正是最常见的两种链接错误。

- 正文中交叉应用找不到对应的 `anchor`，空白填不上咋办？
- 某个 `anchor` 多次定义，该选哪一个填到交叉引用的空白处呢？

上面描述的办法要至少翻两遍全文，有没有办法从头到尾只翻一遍书稿就完成交叉引用呢？如果作者在写书的时候只从前面的章节引用后面的章节，那么是可以做到这一点的。我们在编排页码和章节号的时候顺便阅读全文，遇到新的交叉引用空白就记到一张之上。这张纸记录交叉引用的名字和空白出现的页码。我们知道后面肯定能遇到对应的 `anchor`。在遇到一个 `anchor` 时，去那张纸上看看有没有交叉引用用到它，如果有，就往回翻到空白的页码，把空白填上，回头再继续编制页码和章节号。这样一遍扫下来，章节编号、页码、交叉引用就全部搞定了。

这正是传统 `one-pass` 链接器的工作方式，在使用这种链接器的时候要注意参数顺序，越基础的库越放到后面。如果程序用到了多个 `library`，这些 `library` 之间有依赖（假设不存在循环依赖），那么链接器的参数顺序应该是依赖图的拓扑排序。这样保证每个未决符号都可以在后面出现的库中找到。比如 A、B 两个彼此独立的库同时依赖 C 库，那么链接的顺序是 ABC 或 BAC。

为什么这个规定不是反过来，先列出基础库，再列出应用库呢？原因是前一种做法的内存消耗要小得多。如果先处理基础库，链接器不知道库里哪些符号会被后面的代码用到，因此只能每一个都记住，链接器的内存消耗跟所有库的大小之和成正比。反过来，如果先处理应用库，那么只需要记住目前尚未查到定义的符号就行了。链接器的内存消耗跟程序中外部符号的多少成正比（而且一旦填上空白，就可以忘掉它）。

以上简要介绍了 C 语言的链接模型，C++ 与之相比主要增加了两项内容：

- 函数重载，需要类型安全的链接^[D&E, 第 11.3 节]，即 `name mangling`。⁵⁵
- `vague linkage`^{56 57}，即同一个符号有多份互不冲突的定义。

`name mangling` 的事情一般不需要程序员操心，只要掌握 `extern "C"` 的用法，能和 C 程序库 `interoperate` 就行。何况现在一般的 C 语言库的头文件都会适当使用 `extern "C"`，使之也能用于 C++ 程序。

C 语言通常一个符号在程序中只能有一处定义，否则就会造成重复定义。C++ 则不同，编译器在处理单个源文件的时候并不知道某些符号是否应该在本编译单元定义。为了保险起见，只能每个目标文件生成一份“弱定义”，而依赖链接器去选择一份作为最终的定义，这就是 `vague linkage`。不这么做的话就会出现未定义的符号错误，因为链接器通常不会聪明到反过来调用编译器去生成未定义的符号。为了让这种机制能正确运作，C++ 要求代码满足一次定义原则（ODR），否则代码的行为是随机的，视 `linker` 心情好坏而定。

以下分别简要谈谈这两方面对编程的影响。

10.3.1 函数重载

众所周知，为了实现函数重载，C++ 编译器普遍采用名字改编（`name mangling`）的办法⁵⁸，为每个重载函数生成独一无二的名字，这样在链接的时候就能找到正确的重载版本。比如 `foo.cc` 里定义了两个 `foo()` 重载函数。

```
// foo.cc
int foo(bool x)
{
    return 42;
}

int foo(int x)
{
    return 100;
}

$ g++ -c foo.cc
$ nm foo.o                # foo.o 定义了两个 external linkage 函数
0000000000000000 T _Z3foob
0000000000000010 T _Z3fooi
```

⁵⁵ http://en.wikipedia.org/wiki/Name_mangling

⁵⁶ <http://gcc.gnu.org/onlinedocs/gcc/Vague-Linkage.html>

⁵⁷ <http://sourcery.mentor.com/public/cxx-abi/abi.html#vague>

⁵⁸ 此处以 `g++` 为例，规则见 <http://sourcery.mentor.com/public/cxx-abi/abi.html#mangling>。

```
$ c++filt _Z3foob _Z3fooi # unmangle 这两个函数名
foo(bool)                # 注意, mangled name 里没有返回类型
foo(int)
```

注意普通 non-template 函数的 mangled name 不包含返回类型。记得吗, 返回类型不参与函数重载。

这其实有一个小小的隐患, 也是“C++ 典型缺陷”的一个体现。如果一个源文件用到了重载函数, 但它看到的函数原型声明的返回类型是错的 (违反了 ODR), 链接器无法捕捉这样的错误。

```
// main.cc
void foo(bool);           # 返回类型错误地写成了 void

int main()
{
    foo(true);
}

$ g++ -c main.cc
$ nm main.o               # 目标文件依赖 _Z3foob 这个符号
                           U _Z3foob
0000000000000000 T main

$ g++ main.o foo.o        # 能正常生成 ./a.out
```

对于内置类型, 这应该不会造成实际的影响。但是如果返回类型是 class, 那么就晓得会发生什么了。

10.3.2 inline 函数

inline 函数的方方面面见 [EC3] 第 30 条。由于 inline 函数的关系, C++ 源代码里调用一个函数并不意味着生成的目标代码里也会做一次真正的函数调用 (可能看不到 call 指令)。现在的编译器聪明到可以自动判断一个函数是否适合 inline, 因此 inline 关键字在源文件中往往不是必需的。当然, 在头文件里 inline 还是要的, 为了防止链接器抱怨重复定义 (multiple definition)。现在的 C++ 编译器采用重复代码消除⁵⁹的办法来避免重复定义。也就是说, 如果编译器无法 inline 展开的话, 每个编译单元都会生成 inline 函数的目标代码, 然后链接器会从多份实现中任选一份保留, 其余的则丢弃 (vague linkage)。如果编译器能够展开 inline 函数, 那就不必单独为之生成目标代码了 (除非使用函数指针指向它)。

⁵⁹ 见 [LLL] 第 4.4.1 节。

如何判断一个 C++ 可执行文件是 debug build 还是 release build? 换言之, 如何判断一个可执行文件是 -O0 编译还是 -O2 编译? 我通常的做法是看 class template 的短成员函数有没有被 inline 展开。例如:

```
// vec.cc
#include <vector>
#include <stdio.h>

int main()
{
    std::vector<int> vi;
    printf("%zd\n", vi.size()); # 这里调用了 inline 函数 size()
}

$ g++ -Wall vec.cc      # non-optimized build
$ nm ./a.out |grep size|c++filt
00000000004007ac W std::vector<int, std::allocator<int> >::size() const
// vector<int>::size() 没有 inline 展开, 目标文件中出现了函数 (弱) 定义。

$ g++ -Wall -O2 vec.cc # optimized build
$ nm ./a.out |grep size|c++filt
// 没有输出, 因为 vector<int>::size() 被 inline 展开了。
```

注意, 编译器为我们自动生成的 class 析构函数也是 inline 函数, 有时候我们要故意 out-line, 防止代码膨胀或出现编译错误。以下 Printer 是依据后面 §11.4 介绍的 pimpl 手法实现的公开 class。这个 class 的头文件完全没有暴露 Impl class 的任何细节, 只用到了前向声明。并且有意地把构造函数和析构函数也显式声明了。

```
#include <boost/scoped_ptr.hpp>                                     printer.h

class Printer // : boost::noncopyable
{
public:
    Printer();
    ~Printer(); // make it out-line
    // other member functions

private:
    class Impl; // forward declaration only
    boost::scoped_ptr<Impl> impl_;
};
```

printer.h

在源文件中, 我们可以从容地先定义 Printer::Impl, 然后再定义 Printer 的构造函数和析构函数。

```
#include "printer.h"                                              printer.cc
```

```
class Printer::Impl
{
    // members
};

Printer::Printer()
: impl_(new Impl) // 现在编译器看到了 Impl 的定义，这句话能编译通过。
{ }

Printer::~Printer() // 尽管析构函数是空的，也必须放到这里来定义。否则编译器
{                  // 在将隐式声明的 ~Printer() inline 展开的时候无法看到
                  // Impl::~Impl() 的声明，会报错。见 boost::checked_delete
}
printer.cc
```

在现代的 C++ 系统中，编译和链接的界限更加模糊了。传统 C++ 教材告诉我们，要想编译器能够 `inline` 一个函数，那么这个函数体必须在当前编译单元可见。因此我们通常把公共 `inline` 函数放到头文件中。现在有了 `link time code generation`⁶⁰，编译器不需要看到 `inline` 函数的定义，`inline` 展开可以留给链接器去做。

除了 `inline` 函数，g++ 还有大量的内置函数 (`built-in function`)⁶¹，因此源代码中出现 `memcpy`、`memset`、`strlen`、`sin`、`exp` 之类的“函数调用”不一定真的会调用 `libc` 里的库函数。另外，由于编译器知道这些函数的功能，因此优化起来更充分。例如 `muduo` 日志库就使用了内置 `strchr()` 函数在编译期求出文件的 `basename`。

有意思的是，编译器如何处理 `inline` 函数中的 `static` 变量？这个留给有兴趣的读者去探究吧。

10.3.3 模板

C++ 模板包括函数模板和类模板，与链接相关的话题包括：

- 函数定义，包括具现化后的函数模板、类模板的成员函数、类模板的静态成员函数等。
- 变量定义，包括函数模板的静态数据变量、类模板的静态数据成员、类模板的全局对象等。

模板编译链接的不同之处在于，以上具有 `external linkage` 的对象通常会在多个编译单元被定义。链接器必须进行重复代码消除⁶²，才能正确生成可执行文件。

⁶⁰ 2010 年发布的 `gcc4.5` 开始支持 `-f1to` 选项，落后于 Visual C++ .NET 好多年。

⁶¹ <http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Other-Builtins.html>

⁶² 见 [LLL] 第 4.4.1 节。

template 和 inline 函数会不会导致代码膨胀

假设有一个定长 Buffer 类，其内置 buffer 长度是在编译期确定的，我们可以把它实现为非类型类模板：（完整代码见 muduo/base/LogStream.h 中的 FixedBuffer class）

```
template<int Size>
class Buffer
{
public:
    Buffer() : index_(0) {}

    void append(const void* data, int len)
    {
        ::memcpy(buffer_+index_, data, len);
        index_ += len;
    }

    void clear() { index_ = 0; }
    // other members

private:
    char buffer_[Size]; // Size 是模板参数
    int index_;
};
```

在代码中使用了 Buffer<256> 和 Buffer<1024> 两份具体体：

```
int main()
{
    Buffer<256> b1;
    b1.append("hello", 5); // Buffer<256>::append()
    b1.clear();           // Buffer<256>::clear()

    Buffer<1024> b2;
    b2.append("template", 8); // Buffer<1024>::append()
    b2.clear();             // Buffer<1024>::clear()
}
```

按照 C++ 模板的具现化规则，编译器会为每一个用到的类模板成员函数具现化一份实体。

```
$ g++ buffer.cc
$ nm a.out
00400748 W _ZN6BufferILi1024EE5clearEv      # Buffer<1024>::clear()
004006f2 W _ZN6BufferILi1024EE6appendEPKvi  # Buffer<1024>::append(void const*, int)
004006da W _ZN6BufferILi1024EEC1Ev         # Buffer<1024>::Buffer()
004006c2 W _ZN6BufferILi256EE5clearEv      # Buffer<256>::clear()
0040066c W _ZN6BufferILi256EE6appendEPKvi  # Buffer<256>::append(void const*, int)
00400654 W _ZN6BufferILi256EEC1Ev         # Buffer<256>::Buffer()
```

这样看来真的造成了代码膨胀，但实际情况并不一定如此，如果我们用 -O2 编译一下，会发现编译器把这些短函数都 inline 展开了。

```
$ g++ -O2 buffer.cc
$ nm a.out |c++filt |grep Buffer
# 没有输出, Buffer 的成员函数都被 inline 展开了, 没有生成函数定义。
```

如果我们想限制模板的具现化, 比方说限制 Buffer 只能有 64、256、1024、4096 这几个长度, 除了可以用 `static_assert` 来制造编译期错误, 还可以用下面这个只声明、不定义的办法来制造链接错误。

一般的 C++ 教材会告诉你, 模板的定义要放到头文件中, 否则会有编译错误。如果读者足够细心, 会发现其实所谓的“编译错误”是链接错误。例如

```
// main.cc
template<typename T>
void foo(const T&);    // 只声明而没有定义

template<typename T>
T bar(const T&);       // 只声明而没有定义

int main()
{
    foo(0);
    foo(1.0);
    bar('c');
}

$ g++ main.cc          # 注意是链接器报错, 不是编译器报错
/tmp/cc5SKd58.o: In function `main':
main.cc:(.text+0x17): undefined reference to `void foo<int>(int const&)'
main.cc:(.text+0x31): undefined reference to `void foo<double>(double const&)'
main.cc:(.text+0x41): undefined reference to `char bar<char>(char const&)'
collect2: ld returned 1 exit status
```

那么有办法把模板的实现放到库里, 头文件里只放声明吗? 其实是可以的, 前提是你知道模板会有哪些具现化类型, 并事先显式 (或隐式) 具现化出来。

```
$ g++ -c main.cc      # 可以单独编译为目标文件
$ nm main.o           # 目标文件里引用了未定义的模板函数,
                      # 注意这次函数 mangled name 包含返回类型
                 U _Z3barIcET_RKS0_      # char bar<char>(char const&)
                 U _Z3fooIdEvRKT_        # void foo<double>(double const&)
                 U _Z3fooIiEvRKT_        # void foo<int>(int const&)
0000000000000000 T main

// foobar.cc
template<typename T>
void foo(const T&)
{
}
}
```

```

template<typename T>
T bar(const T& x)
{
    return x;
}

template void foo(const int&);      # 显式具现化
template void foo(const double&);  # 如果漏了这几行，仍然会有链接错误。
template char bar(const char&);

$ g++ -c foobar.cc
$ nm foobar.o                      # foobar.o 包含模板函数的定义
0000000000000000 W _Z3barIcET_RKS0_
0000000000000000 W _Z3fooIdEvRKT_
0000000000000000 W _Z3fooIiEvRKT_

$ g++ main.o foobar.o             # 可以成功生成 a.out

```

对于通用（universal）的模板库，这个办法是行不通的，因为你不可能事先知道客户会用哪些参数类型来具现化你的模板（比方说 `vector<T>` 和 `shared_ptr<T>`）。但是对于某些特殊情况，这可以减少代码膨胀，比方说把 `Buffer<int>` 的构造函数从头文件移到某个源文件，并且只具现化几个固定的长度，这样防止客户代码任意具现化 `Buffer` 模板。

对于 `private` 成员函数模板，我们也不用在头文件中给出定义，因为用户代码不能调用它，也就无法随意具现化它，所以不会造成链接错误。考虑下面这个多功能打印机的例子，`Printer` 既能打印，也能扫描。`PrintRequest` 和 `ScanRequest` 都是由代码生成器生成的 `class`，它们有一些共同的成员，但是没有共同的基类。

```

class PrintRequest
{
public:
    int getUserId() const { return userId_; }
    // other members
private:
    int userId_;
};

class ScanRequest
{
public:
    int getUserId() const { return userId_; }
    // other members
private:
    int userId_;
};

```

Request.h

Request.h

我们写一个 `Printer` class，能同时处理这两种请求，为了避免代码重复，我们打算用一个函数模板来解析 `request` 的公共部分。

```

class PrintRequest;
class ScanRequest;

class Printer : boost::noncopyable // 注意 Printer 不是模板
{
public:
    void onRequest(const PrintRequest&);
    void onRequest(const ScanRequest&);

private:
    template<typename REQ>
    void decodeRequest(const REQ&);

    void processRequest();

    int currentRequestUserId_;
};

```

Printer.h

这个 `decodeRequest` 是模板，但不必把实现暴露在头文件中，因为只有 `onRequest` 会调用它。我们可以把这个成员函数模板的实现放到源文件中。这样的好处之一是 `Printer` 的用户看不到 `decodeRequest` 函数模板的定义，可以加快编译速度。

```

#include "Printer.h"
#include "Request.h"

template<typename REQ>
void Printer::decodeRequest(const REQ& req)
{
    currentRequestUserId_ = req.getUserId();
    // decode other parts
}

// 现在编译器能看到 decodeRequest 的定义，也就能自动具现化它

void Printer::onRequest(const PrintRequest& req)
{
    decodeRequest(req);
    processRequest();
}

void Printer::onRequest(const ScanRequest& req)
{
    decodeRequest(req);
    processRequest();
}

```

Printer.cc

前面展示的几种 `template` 用法一般不会用在通用的模板库中，因此很少有书籍或文章谈到它们。在编写应用程序的时候适当使用模板能减少重复劳动，降低出错的可能，值得了解一下。

另外，C++11 新增了 `extern template` 特性，可以阻止隐式模板具现化。`g++` 很早就支持这个特性，`g++` 的 C++ 标准库就使用了这个办法，使得使用 `std::string` 和 `std::iostream` 的代码不受代码膨胀之苦。

```
// ios.cc
#include <iostream>
#include <string>

using namespace std;

int main()           // 用到了 iostream 和 string 两个大模板
{
    string name;
    cin >> name;
    cout << "Hello, " << name << "\n";
}

$ g++ ios.cc
$ size a.out          # 生成的可执行文件很小
   text    data     bss     dec     hex filename
   2900     648     584    4132    1024 a.out

$ nm a.out |grep ' [TW] ' # 仔细看目标文件，并没有具现化那些巨大的类模板
0000000000400c20 T __libc_csu_fini
0000000000400c30 T __libc_csu_init
0000000000400cf8 T _fini
0000000000400958 T _init
0000000000400a50 T _start
0000000000601288 W data_start
0000000000400b34 T main

$ nm a.out |grep -o ' U .*' # 而是引用了标准库中的实现
U _ZNSt8ios_base4InitC1Ev@@GLIBCXX_3.4 # 这两个是 string 的构造函数与析构函数
U _ZNSt8ios_base4InitD1Ev@@GLIBCXX_3.4
U _ZStlsISt11char_traitsIcESaIcEERSt13basic_ostreamIT_0ES7_RKSbIS4_S5_T1_E
U _ZStlsIcSt11char_traitsIcESaIcEERSt13basic_ostreamIT_0ES7_RKSbIS4_S5_T1_E
U _ZStrsIcSt11char_traitsIcESaIcEERSt13basic_istreamIT_0ES7_RKSbIS4_S5_T1_E
```

这或许能帮助消除一定的模板恐惧吧。

10.3.4 虚函数

在现在的 C++ 实现中，虚函数的动态调用（动态绑定、运行期决议）是通过虚函数表（`vtable`）进行的，每个多态 `class` 都应该有一份 `vtable`。定义或继承了虚函数

的对象中会有一个隐含成员：指向 `vtable` 的指针，即 `vptr`。在构造和析构对象的时候，编译器生成的代码会修改这个 `vptr` 成员，这就要用到 `vtable` 的定义（使用其地址）。因此我们有时看到的链接错误不是抱怨找不到某个虚函数的定义，而是找不到虚函数表的定义。例如：

```
// virt.cc
class Base
{
public:
    virtual ~Base();
    virtual void doIt();
};

int main()
{
    Base* b = new Base;
    b->doIt();
}

$ g++ virt.cc
/tmp/cc8Q7qKi.o: In function `Base::~Base()':
virt.cc:(.text._ZN4BaseC1Ev[Base::~Base()]+0xf):
        undefined reference to `vtable for Base'
collect2: ld returned 1 exit status
```

出现这种错误的根本原因是程序中某个虚函数没有定义，知道了这个方向，查找问题就不难了。

另外，按道理说，一个多态 `class` 的 `vtable` 应该恰好被某一个目标文件定义，这样链接就不会有错。但是 C++ 编译器有时无法判断是否应该在当前编译单元生成 `vtable` 定义⁶³，为了保险起见，只能每个编译单元都生成 `vtable`，交给链接器去消除重复数据⁶⁴。有时我们不希望 `vtable` 导致目标文件膨胀，可以在头文件的 `class` 定义中声明 `out-line` 虚函数⁶⁵。

10.4 工程项目中头文件的使用规则

既然短时间内 C++ 还无法摆脱头文件和预处理，因此我们要深入理解可能存在的陷阱。在实际项目中，有必要规范头文件和预处理的用法，避免它们的危害。一旦为了使用某个 `struct` 或者某个库函数而包含了一个头文件，那么这个头文件中定义的其他名字（`struct`、函数、宏）也被引入当前编译单元，有可能制造麻烦。

⁶³ 就跟“无法 `inline` 的 `inline` 函数”一个道理。

⁶⁴ <http://sourcery.mentor.com/public/cxx-abi/abi.html#vague-vtable>

⁶⁵ http://llvm.org/docs/CodingStandards.html#ll_virtual_anch

10.4.1 头文件的害处

我认为头文件的害处主要体现在以下几方面：

- 传递性。头文件可以再包含其他头文件。前面已经举过例子，一个简单的 `#include <complex>` 展开之后有两万多行代码，一方面造成编译缓慢；另一方面，任何一个头文件改动一点点代码都会需要重新编译所有直接或间接包含它的源文件。因为 build tool 无法有效判断这个改动是否会影响程序语义，保守起见只能把受影响的源文件全部重新编译一遍。因此，合理组织源代码，减少开发时 rebuild 的成本是每个稍具规模项目的必做功课。
- 顺序性。一个源文件可以包含多个头文件。如果头文件内容组织不当，会造成程序的语义跟头文件包含的顺序有关，也跟是否包含某一个头文件有关⁶⁶。通常的做法是把头文件分为几类⁶⁷，然后分别按顺序包含这几类头文件⁶⁸，相同类的头文件按文件名的字母排序。这样一方面源代码比较整洁；另一方面如果两个人同时修改源码，各自想多包含一个头文件，那么造成冲突的可能性较小。一般应该避免每次在 `#include` 列表的末尾添加新的头文件，这样很快代码的依赖关系就无法管理了。
- 差异性。内容差异造成不同源文件看到的头文件不一致，时间差异造成头文件与库文件内容不一致。例如 §12.7 提到不同的编译选项会造成 Visual C++ `std::string` 的大小不一样。也就是说 `<string>` 头文件的内容经过预处理后会有变化，如果两个源文件编译时的宏定义选项不一致，可能造成二进制代码不兼容。这说明整个程序应该用统一的编译选项⁶⁹。如果程序用到了第三方静态库或者动态库，除了拿到头文件和库文件，我们还要拿到当时编译这个库的编译选项，才能安全无误地使用这个程序库。如果程序用到了两个库，但是它们的编译选项有冲突，那麻烦就大了，后面谈库文件组织的时候再来说这个问题和时间差异的问题。

反观现代的编程语言，它们比 C++ 的历史包袱轻多了，模块化做得也比较好。模块化的做法主要有两种：

⁶⁶ 假设有两个源文件，一个包含了 `foo.h`，一个没有，`foo.h` 里定义了特殊的宏、模板特化或者 `struct` 对齐指令，那么这两个源文件中相同代码的行为就可能不一致了。而且这种不一致很难追查。

⁶⁷ 例如分为 C 语言系统头文件、C++ 标准库头文件、C++ 第三方库头文件、本公司的基础库头文件、本项目的头文件。

⁶⁸ 不同的编程规范的建议不一致，我个人是按从特殊到一般的顺序包含头文件，见 `muduo` 源码。

⁶⁹ 另外一个例子是 `g++` 的 `-malign-double` 选项会影响 32-bit 下 `double` 类型的地址对齐，如果两个源码的编译选项不同，造成同一个 `struct/class` 的 layout 不同，那么程序的行为就会很飘逸了。

- 对于解释型语言，`import` 的时候直接把对应模块的源文件解析（`parse`）一遍（不再是简单地把源文件包含进来）。
- 对于编译型语言，编译出来的目标文件（例如 Java 的 `.class` 文件）里直接包含了足够的元数据，`import` 的时候只需要读目标文件的内容，不需要读源文件。

这两种做法都避免了声明与定义不一致的问题，因为在这些语言里声明与定义是一体的。同时这种 `import` 手法也不会引入不想要的名字，大大简化了名字查找的负担（无论是人脑还是编译器），也不用担心 `import` 的顺序不同造成代码功能变化。

10.4.2 头文件的使用规则

几乎每个 C++ 编程规范都会涉及头文件的组织。归纳起来观点如下：

- “将文件间的编译依赖降至最小。” [EC3, 条款 31]
- “将定义式之间的依赖关系降至最小。避免循环依赖。” [CCS, 条款 22]
- “让 `class` 名字、头文件名字、源文件名字直接相关。”⁷⁰ 这样方便源代码的定位。`muduo` 源码遵循这一原则，例如 `TcpClient` `class` 的头文件是 `TcpClient.h`，其成员函数定义在 `TcpClient.cc`。
- “令头文件自给自足。” [CCS, 条款 23] 例如要使用 `muduo` 的 `TcpServer`，可以直接包含 `TcpServer.h`。为了验证 `TcpServer.h` 的自足性（`self-contained`），`TcpServer.cc` 第一个包含的头文件就是它。
- “总是在头文件内写内部 `#include guard`（护套），不要在源文件写外部护套。” [CCS, 条款 24] 这是因为现在的预处理对这种通用做法有特别的优化，GNU `cpp` 在第二次 `#include` 同一个头文件时甚至不会去读这个文件，而是直接跳过⁷¹。
- `#include guard` 用的宏的名字应该包含文件的路径全名（从版本管理器的角度），必要的话还要加上项目名称（如果每个项目有自己的代码仓库）⁷²。
- 如果编写程序库，那么公开的头文件应该表达模块的接口⁷³，必要的时候可以把实现细节放到内部头文件中。`muduo` 的头文件满足这条规则（p. 130）。

遵循以上规则，作为应用程序的作者，一般就不会遇到跟头文件和预处理相关的诡异问题。这里介绍一个查找头文件包含途径的小技巧。比方说有一个程序只包含了

⁷⁰ http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Header_Files

⁷¹ http://gcc.gnu.org/onlinedocs/gcc-4.4.4/cpp/Once_002dOnly-Headers.html

⁷² http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#The_define_Guard

⁷³ http://llvm.org/docs/CodingStandards.html#hl_module

<iostream>, 但是却能使用 `std::string`, 我想知道 <string> 是如何被引入的。办法是在当前目录创建一个 `string` 文件, 然后制造编译错误, 步骤如下:

```
// hello.cc
#include <iostream>

int main()
{
    std::string s = "muduo"; // 奇怪, 明明没有包含 <string> 却能使用 std::string
}

$ cat > string // 创建一个只有一行内容的 string 文件
#error error
^D

$ g++ -M -I . hello.cc // 用 g++ 帮我们查出包含途径, 原来是 locale_classes.h 干的
In file included from /usr/include/c++/4.4/bits/locale_classes.h:42,
                 from /usr/include/c++/4.4/bits/ios_base.h:43,
                 from /usr/include/c++/4.4/ios:43,
                 from /usr/include/c++/4.4/ostream:40,
                 from /usr/include/c++/4.4/iostream:40,
                 from hello.cc:1:
./string:1:2: error: #error error
```

下面我们谈谈在编写和使用库的时候应该注意些什么。

10.5 工程项目中库文件的组织原则

考虑一个稍具规模的公司, 有一个基础库团队, 开发并维护公司的公共 C++ 网络库 `net`; 还有一个团队开发了一套消息中间件, 并提供了一个客户端库 `hub`, `hub` 是基于 `net` 构建的; 最近公司又有另外一个团队开发了一套存储系统, 并提供了一个客户端库 `cab`, `cab` 也是基于 `net` 构建的。公司内部开发的服务端程序可能会用到这些库的一个或几个, 本节主要讨论如何组织这些由不同团队开发的库与应用程序。

在谈具体的 C++ 库文件的组织之前, 先谈一谈更基本的话题: 依赖管理。

假设你负责实现并维护一个关键的网络服务程序 `app`, 经过充分测试之后, `app 1.0` 上线运行, 一切顺利。`app 1.0` 用到了网络库 (`net 1.0`) 和消息中间件的客户端库 (`hub 1.0`), 并且 `hub 1.0` 本身也用到了 `net 1.0`, 依赖关系如图 10-3 所示。

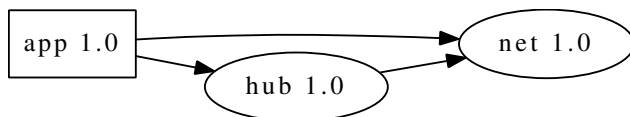


图 10-3

尽管在发布之前 QA 人员 sign-off 的是 app 1.0, 但是我们应该认为他们 sign-off 的是 app 1.0 和它依赖的所有库构成的 bundle。因为 app 的行为跟它用到的库有关, 如果改变其中任何一个库, app 的行为都可能发生变化 (尽管 app 的源码和可执行文件一个字节都没动), 也就可能跟当时充分测试通过的“app 1.0”行为不一致。

周伟明老师在《软件测试实践》的第 1.7.2 节“COM 的可测试性分析”中明确表示, COM “违反了软件设计的基本原理”, 其理由是:

我们假设一个软件包含 n 个不同的 COM 组件, 按照 COM 的设计思想, 每个组件都是可以替换的。假设每个组件都有若干个不同的版本, 记为分别有 $M_1, M_2, \dots, M_n$ 个不同的版本, 那么组成整个软件的所有组件的组合关系有 $M_1 \times M_2 \times \dots \times M_n$ 种, 等于这个软件共有 $\prod_{i=1}^n M_i$ 种二进制版本。如果要将测试做得充分, 这些组合全部都需要进行测试, 否则很难保证没测试到的组合不会有问题。

这至少从理论上说明, 改动程序本身或它依赖的库之后应该重新测试, 否则测试通过的版本和实际运行的版本根本就是两个东西。一旦出了问题, 责任就难理清了。

这个问题对于 C++ 之外的语言也同样存在, 我认为凡是可以在编译之后替换库的语言都需要考虑类似的问题⁷⁴。对于脚本语言来说, 除了库之外, 解释器的版本 (Python 2.5/2.6/2.7) 也会影响程序的行为, 因此有 Python virtualenv 和 Ruby rbenv 这样的工具, 允许一台机器同时安装多个解释器版本。Java 程序的行为除了跟 class path 里的那些 jar 文件有关, 也跟 JVM 的版本有关, 通常我们不能在没有充分测试的情况下升级 JVM 的大版本 (从 1.5 到 1.6)。

除了库和运行环境, 还有一种依赖是对外部进程的依赖, 例如 app 程序依赖某些数据源 (运行在别的机器上的进程), 会在运行的时候通过某种网络协议从这些数据源定期或不定期读取数据。数据源可能会升级, 其行为也可能变化, 如何管理这种依赖就超出本节的范围了。

回到 C++, 首先谈编译器版本之间的兼容性。截至 g++ 4.4, Linux 目前已有四个互不兼容的 ABI⁷⁵ 版本, 编译出来的库互不通用:

⁷⁴ http://en.wikipedia.org/wiki/Dependency_hell

⁷⁵ 这个指的是编译器的 ABI, 跟第 11 章提到的程序库的 ABI 不是一回事。另外本处暂不考虑将来 C++11 引起的 ABI 变化 (<http://gcc.gnu.org/onlinedocs/gcc/Compatibility.html>)。

- 1. gcc 3.0 之前的版本，例如 2.95.3
- 2. gcc 3.0/3.1⁷⁶
- 3. gcc 3.2/3.3⁷⁷
- 4. gcc 3.4 ~ 4.4⁷⁸

现在看来，这其实影响不大，因为估计没有谁还在用 g++ 3.x 来编译新的代码。

另外一个需要考虑的是 C++ 标准库（libstdc++）的版本与 C 标准库（glibc）的版本。C++ 标准库的版本跟 C++ 编译器直接关联⁷⁹，我想一般会有人去替换系统的 libstdc++。C 标准库的版本跟 Linux 操作系统的版本直接相关，见表 10-1。一般也不会有人单独升级 glibc，因为这基本上意味着需要重新编译用户态的所有代码。另外，为了稳妥起见，通常建议用 Linux 发行版自带的那个 gcc 版本来编译你的代码。因为这个版本的 gcc 是 Linux 发行版主要支持的编译器版本，当前 kernel 和用户态的其他程序也基本是它编译的，如果它有什么问题的话，早就被人发现了。

根据以上分析，一旦选定了生产环境中操作系统的版本，另外三样东西的版本就确定了。我们暂且认为生产环境中运行 app 1.0 的机器的 Linux 操作系统版本、libstdc++ 版本、glibc 版本是统一的⁸⁰，而且 C++ 应用程序和库的代码都是用操作系统原生的 g++ 来编译的。表 10-1 列出了几大主流 Linux 发行版的版本配置。

表 10-1

Distro	Kernel	gcc	glibc
RHEL 6	2.6.32	4.4.6	2.12
RHEL 5	2.6.18	4.1.2	2.5
RHEL 4	2.6.9	3.4.6	2.3.4
Debian 6.0	2.6.32	4.4.5	2.11.2
Debian 5.0	2.6.26	4.3.2	2.7
Debian 4.0	2.6.18	4.1.1	2.3.6
Ubuntu 10.04 LTS	2.6.32	4.4.3	2.11.1
Ubuntu 8.04 LTS	2.6.24	4.2.3	2.7
Ubuntu 6.04 LTS	2.6.15	4.0.3	2.3.6

⁷⁶ <http://ols.fedoraproject.org/GCC/Reprints-2003/nathan-gccsummit.pdf>
⁷⁷ <http://gcc.gnu.org/gcc-3.2/c++-abi.html>
⁷⁸ <http://lwn.net/Articles/142828/> http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html
⁷⁹ <http://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html>
⁸⁰ 升级操作系统时这三个都会一起变，那时候程序几乎肯定要重新测试并重新部署上线。

这样一来，我们就可以在 C++ 编译器版本、C++ 标准库版本、C 标准库版本均固定的情况下来讨论应用程序与库的组织⁸¹。进一步说，这里讨论的是公司内部实现的库，而不是操作系统自带的编译好的库（libz、libssl、libcurl 等等）。后面这些库可以通过操作系统的 package 管理机制来统一部署，确保每台机器的环境相同。

Linux 的共享库（shared library）比 Windows 的动态链接库在 C++ 编程方面要好用得多了，对应用程序来说基本可算是透明的，跟使用静态库无区别。主要体现在：

- 一致的内存管理。Linux 动态库与应用程序共享同一个 heap，因此动态库分配的内存可以交给应用程序去释放⁸²，反之亦可。
- 一致的初始化。动态库里的静态对象（全局对象、namespace 级的对象等等）的初始化和程序其他地方的静态对象一样，不用特别区分对象的位置。
- 在动态库的接口中可以放心地使用 class、STL、boost（如果版本相同）。
- 没有 dllimport/dllexport 的累赘。直接 include 头文件就能使用。
- DLL Hell^{83 84} 的问题也小得多，因为 Linux 允许多个版本的动态库并存，而且每个符号可以有多个版本⁸⁵。

DLL hell 指的是安装新的软件的时候更新了某个公用的 DLL，破坏了其他已有软件的功能。例如安装 xyz 1.0 会把 net 库升级为 1.1 版，覆盖了原来 app 1.0 和 hub 1.0 依赖的 net 1.0，这有潜在的风险（图 10-4）。

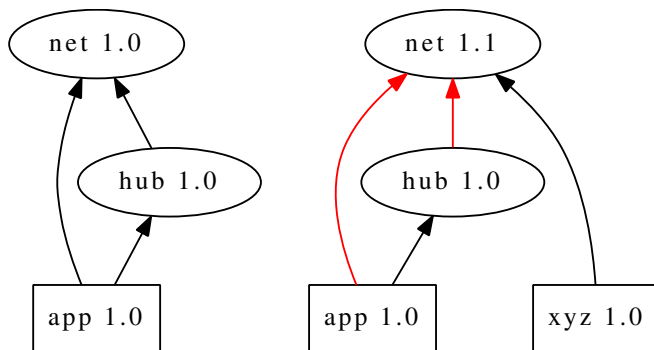


图 10-4

⁸¹ 注意几大主流 Linux 发行版不约而同地在最新稳定版中选择了 Kernel 2.6.32、g++4.4，这也是 muduo 选择 kernel 2.6.32、g++4.4 为首要支持平台的原因。

⁸² 例如 libreadline 的 readline(3) 返回的 char* 指针必须由调用方用 free(3) 释放。

⁸³ http://en.wikipedia.org/wiki/DLL_Hell

⁸⁴ <http://www.desaware.com/tech/dllhell.aspx>

⁸⁵ 见 [LLL] 第 8 章“Linux 共享库的组织”。

现在 Windows 7 里有 side-by-side assembly，基本解决了 DLL hell 问题，代价是系统里有一个巨大的且不断增长的 WinSxS 目录。

一个 C++ 库的发布方式有三种：动态库（.so）、静态库（.a）、源码库（.cc）⁸⁶。表 10-2 简单总结了一些基本特性。

表 10-2

	动态库	静态库	源码库
库的发布方式	头文件 + .so 文件	头文件 + .a 文件	头文件 + .cc 文件
程序编译时间	短	短	长
查询依赖	ldd 查询	编译期信息	编译期信息
部署	可执行文件 + 动态库	单一可执行文件	单一可执行文件
主要时间差	编译时 ⇔ 运行时	编译库 ⇔ 编译应用程序	无

本节谈动态库只包括编译时就链接动态库的那种常规用法，不包括运行期动态加载（dlopen()）的用法。

作为应用程序的作者，如果要在多台 Linux 机器上运行这个程序，我们先要把它部署（deploy）到那些机器上⁸⁷。如果程序只依赖操作系统本身提供的库（包括可以通过 package 管理软件安装的第三方库），那么只要把可执行文件拷贝到目标机器上就能运行。这是静态库和源码库在分布式环境下的突出优点之一。

相反，如果依赖公司内部实现的动态库，这些库必须事先（或者同时）部署到这些机器上，应用程序才能正常运行。这立刻就会面临运维方面的挑战：部署动态库的工作由谁（库的作者还是应用程序的作者）来做呢？另外一个相关的问题是，如果动态库的作者修正了 bug，他可以自主更新所有机器上的库吗？

我们暂且认为库的作者可以独立地部署并更新动态库，并且影响到使用这个库的应用程序⁸⁸。否则的话，如果每个程序都把自己用到的动态库和应用程序一起打包发布，库的作者不负责库的更新，那么这和使用静态库就没有区别了，还不如直接静态链接。

无论哪种方式，我们都必须保证应用程序之间的独立性，也就是让动态库的多个大版本能够并存。例如部署 app 1.0 和 xyz 1.0 之后的依赖关系如图 10-5 所示。

⁸⁶ header-only 的库也算是源码库。
⁸⁷ 如果处于测试目的，多台机器可以从某个网络文件系统启动可执行文件。但是在生产环境中，一般要把可执行文件放到本地文件系统，以减少依赖、增强可用性。
⁸⁸ 具体地说这跟动态库的版本规划有关，比如 net 1.1.1 升级到 net 1.1.2 只会影响原来使用 net 1.1 系列的应用程序，不影响使用 net 1.0 的应用程序。

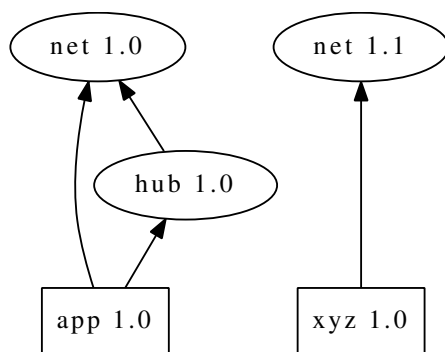


图 10-5

按照传统的观点，动态库比静态库节省磁盘空间和内存空间⁸⁹，并且具备动态更新的能力（可以 hot fix bug⁹⁰），似乎动态库应该是目前的首选⁹¹。但是正是这种动态更新⁹²的能力让动态库成了烫手的山芋。

10.5.1 动态库是有害的

Jeffrey Richter 对动态库的本质问题有精辟的论述⁹³：

一旦替换了某个应用程序用到的动态库，先前运行正常的这个程序使用的将不再是当初 build 和测试时的代码。结果是程序的行为变得不可预期。

怎样在 fix bug 和增加 feature 的同时，还能保证不会损坏现有的应用程序？我（Jeffrey Richter）曾经对这个问题思考了很久，并且得出了一个结论——那就是这是不可能的。

作为库的作者，你肯定不希望更新部署一个看似有益无害的 bug fix 之后，星期一早上被应用程序的维护者的电话吵醒，说程序不能启动（新的库破坏了二进制兼容性）或者出现了不符合预期的行为。

⁸⁹ 对于系统库或许真的如此，但是对于我们自己写的业务库则不一定有多大实际的效果。假设一台服务器上运行 10 个不同的服务程序，每个程序的可执行文件大小是 100MB（当然这是非常夸张的估算），那么一共用了 1GB 的内存来装载代码，相比服务器动辄几十 GB 的内存来说简直是九牛一毛。

⁹⁰ 当然头文件里 inline 函数的 bug 不能通过发布新的库文件来修正，而必须重新编译可执行文件。

⁹¹ 20 世纪 90 年代出版的《C 专家编程》[ExpC] 就大力推崇动态库，仿佛它是灵丹妙药一般。

⁹² 当然我们不能原地（in-place）覆盖更新正在使用的动态库或可执行文件，这会让进程在一段时间之后因 SIGBUS 而崩溃。

⁹³ 《Microsoft .NET 框架程序设计（修订版）》第 3 章，李建忠译。

作为应用程序的作者，你也肯定不希望星期一大早被运维的同事吵醒，说你负责的某个服务进程无法启动或者行为异常。经排查，发现只有某一个动态库的版本与上星期不同。你该朝谁发火呢？

既然双方都不想过这种提心吊胆的日子，那为什么还要用动态库呢？

那么有没有可能在发布动态库的 **bug fix** 之前充分测试所有受影响的应用程序呢？这会遇到一个两难命题：一个动态库的使用面窄，只有两三个程序用到它，测试的成本较低，那么它作为动态库的优势就不明显。相反，一个动态库的使用面宽，有几十个程序用到它，动态库的“优势”明显，测试和更新的成本也相应很高（或许高到足以抵消它的“优势”）。有一种做法是把动态库的更新先发布到 QA 环境，正常运行一段时间之后再发布到生产环境，这么做也有另外的问题：你在测试下一版 **app 1.1** 的时候，该用 QA 环境的动态库版本还是用生产环境的动态库版本？如果程序在编译测试之后行为还会改变，这是不是在让 QA 白费力气？

总之，一旦动态库可能频繁更新⁹⁴，我没有发现一个完美的使用动态库的办法。在决定使用动态库之前，我建议至少要熟悉它的各种陷阱。参考资料如下：

- <http://harmful.cat-v.org/software/dynamic-linking/>
- 《A Quick Tour of Compiling, Linking, Loading, and Handling Libraries on Unix》
(<http://ref.web.cern.ch/ref/CERN/CNL/2001/003/shared-lib/Pr/>)。
- 《How to write shared libraries》(<http://www.akkadia.org/drepper/dsohowto.pdf>)。
- 《Good Practices in Library Design, Implementation, and Maintenance》
(<http://www.akkadia.org/drepper/goodpractice.pdf>)。
- 《Solaris Linker and Libraries Guide》(<http://docs.oracle.com/cd/E19963-01/html/819-0690/>)。
- 《Shared Libraries in SunOS》(<http://www.cs.cornell.edu/courses/cs414/2004fa/sharedlib.pdf>)。

10.5.2 静态库也好不到哪儿去

静态库相比动态库主要有几点好处（http://en.wikipedia.org/wiki/Static_library）：

- 依赖管理在编译期决定，不用担心日后它用的库会变。同理，调试 **core dump** 不会遇到库更新导致 **debug** 符号失效的情况。
- 运行速度可能更快，因为没有 **PLT**（过程查找表），函数调用的开销更小。
- 发布方便，只要把单个可执行文件拷贝到模板机器上。

⁹⁴ “频繁”指的是一两个月一次，因此如果一个应用程序使用了 5 个独立更新的这种动态库，那么几乎每周都会有库更新。

静态库的一个小缺点是链接比动态库慢，有的公司甚至专门开发了针对大型 C++ 程序的链接器⁹⁵。

静态库的作者把源文件编译成 .a 库文件，连同头文件一起打包发布。应用程序的作者用库的头文件编译自己的代码，并链接到 .a 库文件，得到可执行文件。这里有一个编译的时间差：编译库文件比编译可执行文件要早，这就可能造成编译应用程序时看到的头文件与编译静态库时不一样。

比方说编译 net 1.1 时用的是 boost 1.34，但是编译 xyz 这个应用程序的时候用的是 boost 1.40，见图 10-6。这种不一致有可能导致编译错误，或者更糟糕地导致不可预期的运行错误。比方说 net 库以 boost::function 提供回调，但是 boost 1.36 去掉了一个模板类型参数⁹⁶，造成 xyz 1.0 用 boost 1.40 的话就与 net 1.1 不兼容。

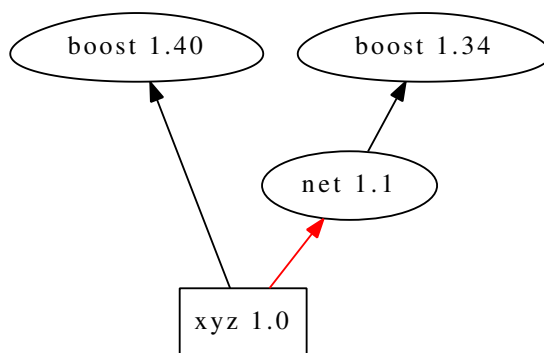


图 10-6

这说明应用程序在使用静态库的时候必须要采用完全相同的开发环境（更底层的库、编译器版本、编译器选项）。但是万一两个静态库的依赖有冲突怎么办？

静态库把库之间的版本依赖完全放到编译期，这比动态库要省心得多，但是仍然不是一件容易的事情。下面略举几种可能遇到的情况：

- 迫使升级高版本。假设一开始应用程序 app 1.0 依赖 net 1.0 和 hub 1.0，一切正常，如图 10-7（左图）所示。在开发 app 1.1 的时候，我们要用到 net 1.1 的功能。但是 hub 1.0 仍然依赖 net 1.0，hub 库的作者暂时没有升级到 net 1.1 的打算。如果不小心的话，就会造成 hub 1.0 链接到 net 1.1，如图 10-7（右图）所示。这就跟编译 hub 1.0 的环境不同了，hub 1.0 的行为不再是经过充分测试的。

⁹⁵ <http://research.google.com/pubs/pub34417.html>

⁹⁶ <http://www.boost.org/doc/html/function/history.html>

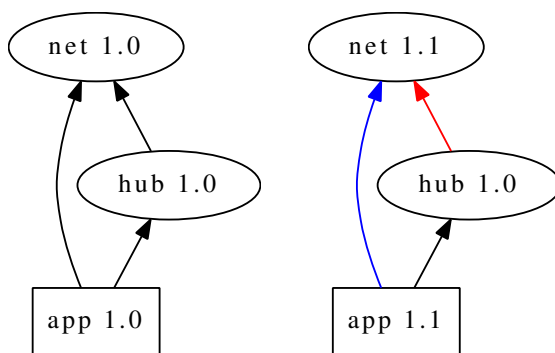


图 10-7

- 重复链接。如果 Makefile 编写不当，有可能出现 hub 1.0 继续链接到 net 1.0，而应用程序则链接到 net 1.1 的情况，如图 10-8（左图）所示。这时如果 net 库里有 internal linkage 的静态变量，可能造成奇怪的行为，因为同一个变量现在有了两个实体，违背了 ODR。一个具体的例子见云风的博客⁹⁷。
- 版本冲突。比方说 app 升级到 1.2 版，想加入一个库 cab 1.0，但是 cab 1.0 依赖 net 1.2，如图 10-8（右图）所示。这时我们的问题是，如果用 net 1.1，则不满足 cab 1.0 的需求；如果用 net 1.2，则不满足 hub 1.1 的需求。那该怎么办？

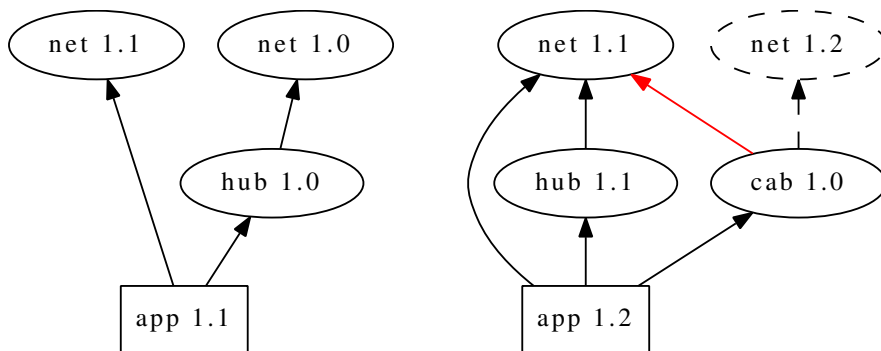


图 10-8

可见静态库的版本管理并不如想象中那么简单。如果一个应用程序用到了三四个公司内部的静态库（见图 10-9），那么协调库之间的版本要花一番脑筋，单独升级任何一个库都可能破坏它原本的依赖。

⁹⁷ http://blog.codingnow.com/2012/01/lua_link_bug.html

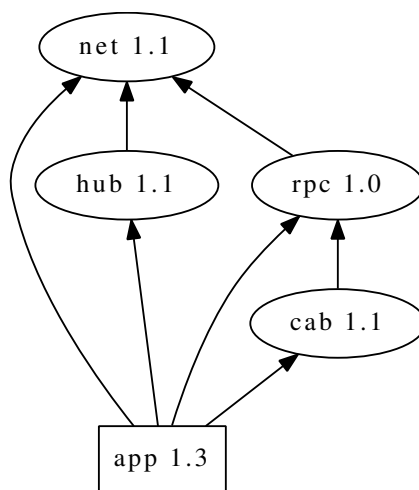


图 10-9

静态库的演化也比较费事。到目前为止我们认为公司没有历史包袱，所有的机器都是 2009 年前后买的，运行的是 Ubuntu 8.04 LTS，软件版本是 g++ 4.2、glibc 2.7、boost 1.34 等等，C++ 程序和库也都是在这个统一的环境下开发的。现在到了 2012 年，线上服务器已服役满 3 年，进入换代周期。新购买的机器打算升级到 Ubuntu 10.04 LTS，因为新内核的驱动程序对新硬件支持更好，而且 8.04 版还有一年多就停止支持了。这样同时升级了内核、gcc 4.4、glibc 2.11、boost 1.40。

这就要求静态库的作者得为新系统重新编译并发布新的库文件。为了避免混淆，我们不得不为库加上后缀名，以标明环境和依赖。假设目前有 net 1.0、net 1.1、net 1.2、hub 1.0、hub 1.1、cab 1.0 等现役的库，那么需要发布多个版本的静态库：

- net1.0_boost1.34_gcc42
- net1.0_boost1.40_gcc44
- net1.1_boost1.34_gcc42
- net1.1_boost1.40_gcc44
- net1.2_boost1.34_gcc42
- net1.2_boost1.40_gcc44
- hub1.0_net1.0_boost1.34_gcc42
- hub1.0_net1.0_boost1.40_gcc44
- hub1.1_net1.1_boost1.34_gcc42
- hub1.1_net1.1_boost1.40_gcc44
- cab1.0_net1.2_boost1.34_gcc42
- cab1.0_net1.2_boost1.40_gcc44

这种组合爆炸式的增长让人措手不及，因为任何一个底层库新增一个变体 (variant)，所有依赖它的高层库都要为之编译一个版本。

如果这些库打算支持 C++11，那么上面这个列表还会长 50%，因为 g++ 为 C++11 修改了 ABI，即使用 `--std=c++0x` 参数编译出来的库文件不能与旧的 C++ 库混用。

要想摆脱这个困境，我目前能想到的办法是使用源码库，即每个应用程序都从头编译所需的库，把时间差减到最小。

10.5.3 源码编译是王道

每个应用程序自己选择要用到的库，并自行编译为单个可执行文件。彻底避免头文件与库文件之间的时间差，确保整个项目的源文件采用相同的编译选项，也不用为库的版本搭配操心。这么做的缺点是编译时间很长，因为把各个库的编译任务从库文件的作者转嫁到了每个应用程序的作者。

另外，最好能和源码版本工具配合，让应用程序只需指定用哪个库，build 工具能自动帮我们 check out 库的源码。这样库的作者只需要维护少数几个 branch，发布库的时候不需要把头文件和库文件打包供人下载，只要 push 到特定的 branch 就行。而且这个 build 工具最好还能解析库的 Makefile (或等价的 build script)，自动帮我们解决库的传递性依赖^{98 99}，就像 Apache Ivy 能做的那样。

在目前看到的开源 build 工具里，最接近这一点的是 Chromium 的 gyp 和腾讯的 typhoon-blade¹⁰⁰，其他如 SCons、CMake、Premake、Waf 等等工具仍然是以库的思路来搭建项目。

总结

由于 C++ 的头文件与源文件分离，并且目标文件里没有足够的元数据供编译器使用，因此必须同时提供库文件和头文件。也就是说要想使用一个已经编译好的 C/C++ 库 (无论是静态库还是动态库)，我们需要两样东西，一是头文件 (.h)，二是库文件 (.a 或 .so)，这就存在了这两样东西不匹配的可能。这是造就 C++ 简陋脆弱的模块机制的根本原因。C++ 库之间的依赖管理远比其他现代语言复杂，在编写程序库和应用程序时，要熟悉各种机制的优缺点，采用开发及维护成本较低的方式来组织和发布库。

⁹⁸ <http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html>

⁹⁹ <http://www.youtube.com/watch?v=2qv3fcXW1mg>

¹⁰⁰ <http://code.google.com/p/typhoon-blade/>

第 11 章

反思 C++ 面向对象与虚函数

C++ 的面向对象语言设施相比其他现代语言可算得上“简陋”，而且与语言的其他部分（better C、数据抽象、泛型）融合度较差（见电子工业出版社出版的《C++ Primer（第 4 版）（评注版）》第 15 章）。在 C++ 中进行面向对象编程会遇到其他语言中不存在的问题，其本质原因是 C++ class 是值语义，而非对象语义。

11.1 朴实的 C++ 设计

去年 8 月¹入职，培训了 4 个月，12 月进入现在这个部门，到现在工作正好一年了。工作内容是软件开发，具体地说，用 C++ 开发一个网络应用（TCP not Web），这是我们的外汇交易系统的一个部件。这半年来，和一两位同事合作把原有的一个 C++ 程序重写了一遍，并增加了很多新功能，重写后的代码不长，不到 15 000 行²，代码质量与性能大大提高。实际上，重写只花了三个月，9 月我们交付了第一个版本，实现了原来的主要功能，吞吐量提高 4 倍。后面这三个月我们在增加新功能，并准备交付第二个版本。这个项目让我对 C++ 的使用有了新的体会，那就是“实用当头，朴实为贵，好用才是王道”。

C++ 是一门（最）复杂的编程语言，语言虽复杂，不代表一定要用复杂的方式来使用它。对于一个金融交易系统，正确性是首要的，价格/数量/交割日期弄错了就会赔钱。在编写代码时，我们特别注意把代码写得尽量简单直白，让人一看就懂。为了控制代码的复杂度，我们采用了基于对象的风格，也就是具体类加全局函数，把 C++ 程序写得如 C 语言一般清晰，同时使用一些 C++ 特性和库来减少代码。

项目中基本没有用到面向对象，或者说没有用到继承和多态的那种面向对象（不一定非得有基类和派生类的设计才是好设计）。引入基类和派生类，或许能带来灵活

¹ 本节内容写于 2008 年底，“去年”指的是 2007 年。

² 经过多年演化，2012 年的代码量是 23 000 行。期间交付了 20 多个大小版本，有两三次重大功能更新。

性，但是代码就不如原来透彻了。在不需要这种灵活性的场合，为什么要付出这样的代价呢？我宁愿花一天时间把几千行 C 代码看懂，也不愿在几十个类组成的继承体系里绕来绕去浪费脑力。定义并使用清晰一致的接口很重要，但“接口”不一定非得是抽象基类，一个类的成员函数就是它的接口。如果看头文件就能明白这个类在干什么、该怎么用固然很好，但如果不明白，打开实现文件，东西都在那儿摆着呢，一望而知。没必要非得用个抽象的接口类把使用者和实现隔开，再把实现隐藏起来，这除了让查找并理解代码变麻烦之外没有任何好处。一个进程内部的解耦意义不大³；相反，函数调用是最直接有效的通信方式。或许采用接口类/实现类的一个可能的好处是依赖注入，便于单元测试。经过权衡比较，我们发现针对各个类写测试的意义不大。另外，如果用白盒测试，那么功能代码和测试代码就得同步更新，会增加不少工作量，碍手碍脚。

程序里边有一处用到了继承，因为它能简化设计。这是一个 `strategy`，涉及一个基类和三四个派生类，所有的类都没有数据成员，只有虚函数。这几个类的代码加起来不到 200 行。这个设计不是一开始就有的，而是在项目进行了一大半的时候，我们发现代码里有若干处针对请求类型的 `switch-case`，于是提炼出了一个 `strategy`，把好几处 `switch-case` 替换为了 `strategy` 对象的虚函数调用，从而简化了代码。这里我们是把 OO 纯粹当做函数指针表来用的。

程序里还有几处用了模板，甚至为了简化与第三方库的交互而动用了 `type traits`，这都是为了简化代码，少敲键盘。这些代码都藏在一个角落里，对外只暴露出一个全局函数的接口，使用者不会被其困扰。

项目里，我们唯一仰赖的 C++ 特性是确定性析构，即一个对象在离开其作用域之后会保证调用析构函数。我们利用这点大大简化了代码，并确保资源和内存的回收。在我看来，确定性析构是 C++ 区别其他主流开发语言（Java/C#/C/动态脚本语言）的最主要特性。

为了确保正确性，我们另外用 Java 写了一个测试夹具（`test harness`）来测试我们这个 C++ 程序。这个测试夹具模拟了所有与我们这个 C++ 程序打交道的其他程序，能够测试各种正常或异常的情况。基本上任何代码改动和 `bug` 修复都在这个夹具中有体现。如果要新加一个功能，会有对应的测试用例来验证其行为。如果发现了一个 `bug`，先往夹具里加一个或几个能复现 `bug` 的测试用例，然后修复代码，让测试通过。我们积累了几百个测试用例，这些用例表示我们对程序行为的预期，是一份

³ 其实，有人一句话道破真相：“但凡你在某个地方切断联系，那么你必然会在另一个地方重新产生联系。”（<http://www.iteye.com/topic/947017>）

可以运行的文档。每次代码改动提交之前，我们都会执行一遍测试，以防低级错误发生。（见本书 §9.7 的详细论述和 §7.12 的例子。）

我们让每个类有明确的职责范围，一个类代表一个概念，不能像个杂货铺一样什么都装。在增加或修改功能的时候，仔细考虑在哪儿下手才最合理。必要时可以动大手脚，而不是每次都选择最简单的修补方式，那样只会使代码越来越臭，积重难返，重蹈上一个版本的覆辙。有时我们会提炼出一个新的类，把原来分散在多个类里的代码集中到一起，从而优化结构。我们有测试夹具保障，并不担心修改会破坏什么。

设计不是一开始就形成的，而是随着项目进展逐步演化出来的。我们的设计是基于类的，而不是基于类的继承体系的。我们是在写应用，不是在写框架，在 C++ 里用那么多继承对我们没好处。一开始我们只有三四个类，实现了基本的报价功能，然后增加了一个类，实现了下单功能。这时我们把报价和下单的共同数据结构提炼成一个新的类，作为原来两个类的成员（而不是基类！），并把解析客户输入的代码移到这个类里。我们的原则是，可以有特别简单的类，但不宜有特别复杂的类，更不能有“大怪兽”。一个类太大，我们就看看能不能把它拆成两个，把责任分开。两个类有共同的代码逻辑，我们会考虑提炼出一个工具类来用，输入数据的验证就是这么提炼出来的一个类。勿以善小而不为，应始终让代码保持清晰易懂。

让代码保持清晰，给我们带来了显而易见的好处。错误更容易暴露，在发布前每多修复一个错误，发布后就少一次半夜被从被窝里叫醒查错的机会。

不要因为某个技术流行而去用它，除非它确实能降低程序的复杂性。毕竟，软件开发的首要技术使命是控制复杂度⁴，防止脑袋爆掉。对于继承要特别小心，这条“贼船”上去就下不来，除非你是继承 `boost::noncopyable`。在讲解面向对象的书里，总会举一些用继承的精巧的例子，比如矩形、正方形、圆形继承自形状，飞机和麻雀继承自“能飞的”，这不意味着继承处处适用。我认为在 C++ 这样需要自己管理内存和对象生命期的语言里，大规模使用面向对象、继承、多态多是自讨苦吃。还不如用 C 语言的思路来设计，在局部用一用继承来代替函数指针表。而 GoF 的《设计模式》与其说是常见问题的解决方案，不如说是绕过（work around）C++ 语言限制的技巧。当然，也是一些人挂在嘴边用来忽悠别人或麻痹自己的灵丹妙药。

11.2 程序库的二进制兼容性

本节主要讨论 Linux x86/x86-64 平台，偶尔会举 Windows 作为反面教材。

⁴《代码大全（第2版）》[CC2e]第5.2节。

C++ 程序员有不同的角色，比如有主要编写应用程序的（`application`），也有主要编写程序库的（`library`），有的程序员或许还身兼多职。如果公司的规模比较大，会出现更细致和明确的分工。比如有的团队专门负责一两个公用的 `library`；有的团队负责某个 `application`，并使用了前一个团队的 `library`。

举一个具体的例子。假设你负责一个图形库，这个图形库功能强大，且经过了充分测试，于是在公司内慢慢推广开来。目前已经有二三十个内部项目用到了你的图形库，大家日子过得挺好。前几天，公司新买了一批大屏幕显示器（分辨率为 2560×1600 像素），不巧你的图形库不能支持这么高的分辨率。（这其实不怪你，因为在你当年编写这个库的时候，市面上显示器的最高分辨率是 1920×1200 像素。）

结果用到了你的图形库的应用程序在 2560×1600 分辨率下不能正常工作，你该怎么办？你可以发布一个新版的图形库，并要求那二三十个项目组用你的新库重新编译他们的程序，然后让他们重新发布应用程序。或者，你提供一个新的库文件，直接替换现有的库文件，应用程序的可执行文件保持不变。

这两种做法各有优劣。第一种做法声势浩大，凡是用到你的库的团队都要经历一个 `release cycle`。后一种做法似乎节省人力，但是有风险：如果新的库文件和原有的应用程序可执行文件不兼容怎么办？

所以，作为 C++ 程序员，只要工作涉及二进制的程序库（特别是动态库），都需要了解二进制兼容性方面的知识。

C/C++ 的二进制兼容性（`binary compatibility`）有多重含义，本文主要在“库文件单独升级，现有可执行文件是否受影响”这个意义下讨论，我称之为 `library`（主要是 `shared library`，即动态链接库）的 `ABI`（`application binary interface`）。至于编译器与操作系统的 `ABI` 见第 10 章。

11.2.1 什么是二进制兼容性

在解释这个定义之前，先看看 Unix 和 C 语言的一个历史问题：`open()` 的 `flags` 参数的取值。`open(2)` 函数的原型如下，其中 `flags` 的取值有三个：`O_RDONLY`、`O_WRONLY`、`O_RDWR`。

```
int open(const char *pathname, int flags);
```

与人们通常的直觉相反，这几个常数值不满足按位或（`bitwise-OR`）的关系，即 $(O_RDONLY \mid O_WRONLY) \neq O_RDWR$ 。如果你想以读写方式打开文件，必须用 `O_RDWR`，而不能用 $(O_RDONLY \mid O_WRONLY)$ 。为什么？因为 `O_RDONLY`、`O_WRONLY`、`O_RDWR` 的值分别是 0、1、2。它们不满足按位或。

那么为什么 Unix/C 语言从诞生到现在一直没有纠正这个小小的缺陷？比方说把 `O_RDONLY`、`O_WRONLY`、`O_RDWR` 分别定义为 1、2、3，这样 `(O_RDONLY | O_WRONLY) == O_RDWR`，符合直觉。而且这三个值都是宏定义，也不需要修改现有的源代码，只需要改改系统的头文件就行了。

这么做会破坏二进制兼容性。对于已经编译好的可执行文件，它调用 `open(2)` 的参数是写死的，更改头文件并不能影响已经编译好的可执行文件。比方说这个可执行文件会调用 `open(path, 1)` 来写文件，而在新规定中，这表示读文件，程序就错乱了。

以上这个例子说明，如果以 `shared library` 方式提供函数库，那么头文件和库文件不能轻易修改，否则容易破坏已有的二进制可执行文件，或者其他用到这个 `shared library` 的 `library`。

操作系统的 `system call` 可以看成 `Kernel` 与 `User space` 的 `interface`，`kernel` 在这个意义下也可以当成 `shared library`，你可以把内核从 2.6.30 升级到 2.6.35，而不需要重新编译所有用户态的程序。

本章所指的“二进制兼容性”是在升级（也可能是 `bug fix`）库文件的时候，不必重新编译使用了这个库的可执行文件或其他库文件，并且程序的功能不被破坏。见 `QT FAQ`⁵ 的有关条款。

在 Windows 有臭名昭著的 `DLL Hell` 问题，比如 `MFC` 有一堆 `DLL`：`mfc40.dll`、`mfc42.dll`、`mfc71.dll`、`mfc80.dll`、`mfc90.dll` 等，这其实是动态链接库的本质问题，怪不到 `MFC` 头上。

11.2.2 有哪些情况会破坏库的 ABI

到底如何判断一个改动是不是二进制兼容呢？这跟 C++ 的实现方式直接相关，虽然 C++ 标准没有规定 C++ 的 ABI，但是几乎所有主流平台都有明文或事实上的 ABI 标准。比方说 `ARM` 有 `EABI`，`Intel Itanium` 有 `Itanium ABI`⁶，`x86-64` 有仿 `Itanium` 的 ABI，`SPARC` 和 `MIPS` 也都有明文规定的 ABI，等等。`x86` 是个例外，它只有事实上的 ABI，比如 Windows 就是 `Visual C++`，Linux 是 `G++`（`G++` 的 ABI 还有多个版本，目前最新的是 `G++ 3.4` 的版本），`Intel` 的 C++ 编译器也得按照 `Visual C++` 或 `G++` 的 ABI 来生成代码，否则就不能与系统的其他部件兼容。

⁵ http://developer.qt.nokia.com/faq/answer/you_frequently_say_that_you_cannot_add_this_or_that_feature_because_it_would

⁶ <http://www.codesourcery.com/public/cxx-abi/abi.html>

C++ 编译器 ABI 的主要内容包括以下几个方面：

- 函数参数传递的方式，比如 x86-64 用寄存器来传函数的前 4 个整数参数；
- 虚函数的调用方式，通常是 vptr/vtbl 机制，然后用 vtbl[offset] 来调用；
- struct 和 class 的内存布局，通过偏移量来访问数据成员；
- name mangling；
- RTTI 和异常处理的实现（以下本文不考虑异常处理）。

C/C++ 通过头文件暴露出动态库的使用方法（主要是函数调用和对象布局），这个“使用方法”主要是给编译器看的，编译器会据此生成二进制代码，然后在运行的时候通过装载器（loader）把可执行文件和动态库绑到一起。如何判断一个改动是不是二进制兼容，主要就是看头文件暴露的这份“使用说明”能否与新版本的动态库的实际使用方法兼容。因为新的库必然有新的头文件，但是现有的二进制可执行文件还是按旧的头文件中的“使用说明”来调用动态库。

先说修改动态库导致二进制不兼容的例子。比如原来动态库里定义了 non-virtual 函数 void foo(int)，新版的库把参数改成了 double。那么现有的可执行文件就无法启动，会发生 undefined symbol 错误，因为这两个函数的 mangled name 不同。但是对于 virtual 函数 foo(int)，修改其参数类型并不会导致加载错误，而是会发生诡异的运行时错误。因为虚函数的决议（resolution）是靠偏移量，并不是靠符号名。

再举一些源代码兼容但是二进制代码不兼容的例子：

- 给函数增加默认参数，现有的可执行文件无法传这个额外的参数。
- 增加虚函数，会造成 vtbl 里的排列变化。（不要考虑“只在末尾增加”这种取巧行为，因为你的 class 可能已被继承。）
- 增加默认模板类型参数，比方说 Foo<T> 改为 Foo<T, Alloc=alloc<T> >，这会改变 name mangling。
- 改变 enum 的值，把 enum Color { Red = 3 }; 改为 Red = 4。这会造成错位。当然，由于 enum 自动排列取值，添加 enum 项也是不安全的（在末尾添加除外）。

给 class Bar 增加数据成员，造成 sizeof(Bar) 变大，以及内部数据成员的 offset 变化，这是不是安全的？通常不是安全的，但也有例外。

- 如果客户代码里有 new Bar，那么肯定不安全，因为 new 的字节数不够装下新 Bar 对象。相反，如果 library 通过 factory 返回 Bar*（并通过 factory 来销毁对象）或者直接返回 shared_ptr<Bar>，客户端不需要用到 sizeof(Bar)，那么可能是安全的。

- 如果客户代码里有 `Bar* pBar; pBar->memberA = xx;`, 那么肯定不安全, 因为 `memberA` 的新 `Bar` 的偏移可能会变。相反, 如果只通过成员函数来访问对象的数据成员, 客户端不需要用到 `data member` 的 `offsets`, 那么可能是安全的。
- 如果客户调用 `pBar->setMemberA(xx);`, 而 `Bar::setMemberA()` 是个 `inline` 函数, 那么肯定不安全, 因为偏移量已经被 `inline` 到客户的二进制代码里了。如果 `setMemberA()` 是“outline”函数, 其实现位于 `shared library` 中, 会随着 `Bar` 的更新而更新, 那么可能是安全的。

那么只使用 `header-only` 的库文件是不是安全呢? 不一定。如果你的程序用了 `boost 1.36.0`, 而你依赖的某个 `library` 在编译的时候用的是 `1.33.1`, 那么你的程序和这个 `library` 就不能正常工作。因为 `1.36.0` 和 `1.33.1` 的 `boost::function` 的模板参数类型的个数不一样, 后者多了一个 `allocator`。

这里有一份黑名单, 列在这里的肯定是二进制不兼容的, 没有列出的也可能是二进制不兼容的, 见 KDE 的文档⁷。

11.2.3 哪些做法多半是安全的

前面我说“不能轻易修改”, 暗示有些改动多半是安全的, 这里有一份白名单, 欢迎添加更多内容。

只要库改动不影响现有的可执行文件的二进制代码的正确性, 那么就是安全的, 我们可以先部署新的库, 让现有的二进制程序受益。

- 增加新的 `class`。
- 增加 `non-virtual` 成员函数或 `static` 成员函数。
- 修改数据成员的名称, 因为生产的二进制代码是按偏移量来访问的。当然, 这会造成源码级的不兼容。
- 还有很多, 不一一列举了。

11.2.4 反面教材: COM

在 C++ 中以虚函数作为接口基本上就跟二进制兼容性说“bye-bye”了。具体地说, 以只包含虚函数的 `class` (称为 `interface class`) 作为程序库的接口, 这样的接口是僵硬的, 一旦发布, 无法修改。

⁷ http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C%2B%2B

另外，Windows 下，Visual C++ 编译的时候要选择 Release 或 Debug 模式，而且 Debug 模式编译出来的 library 通常不能在 Release binary 中使用（反之亦然），这也是因为两种模式下的 CRT 二进制不兼容（主要是内存分配方面，Debug 有自己的簿记（bookkeeping））。Linux 就没有这个麻烦，可以混用。

11.2.5 解决办法

采用静态链接

这里的静态链接不是指使用静态库（.a），而是指完全从源码编译出可执行文件（§10.5.3）。在分布式系统里，采用静态链接也带来部署上的好处，只要把可执行文件放到机器上就能运行，不用考虑它依赖的 libraries。目前 muduo 就是采用静态链接。

通过动态库的版本管理来控制兼容性

这需要非常小心地检查每次改动的二进制兼容性并做好发布计划，比如 1.0.x 版本系列之间做到二进制兼容，1.1.x 版本系列之间做到二进制兼容，而 1.0.x 和 1.1.x 不必二进制兼容。《程序员的自我修养》[LLL] 讲了 .so 文件的命名与二进制兼容性相关的话题，值得一读。

用 pimpl 技法，编译器防火墙

在头文件中只暴露 non-virtual 接口，并且 class 的大小固定为 sizeof(Impl*)，这样可以随意更新库文件而不影响可执行文件。具体做法见 §11.4。当然，这么做又多了一道间接性，可能有一定的性能损失。另见《Exceptional C++》的有关条款和《C++ 编程规范》[CCS，条款 43]。

11.3 避免使用虚函数作为库的接口

作为 C++ 动态库的作者，应当避免使用虚函数作为库的接口。这么做会给保持二进制兼容性带来很大麻烦，不得不增加很多不必要的 interfaces，最终重蹈 COM 的覆辙。

本节主要讨论 Linux x86/x86-64 平台，下面会继续举 Windows/COM 作为反面教材。本节是 §11.2 “程序库的二进制兼容性”的延续，在初次发表 §11.2 内容的时候，我原本以为大家都对“以 C++ 虚函数作为接口”的害处达成了共识，因此就写得比较简略，但现在看来情况并非如此，我还得展开谈一谈。

Linux 多线程服务端编程：使用 muduo C++ 网络库

“接口”有广义和狭义之分，本节用中文“接口”表示广义的接口，即一个库的代码界面；用英文 `interface` 表示狭义的接口，即只包含 `virtual function` 的 `class`，这种 `class` 通常没有 `data member`，在 Java 里有一个专门的关键词 `interface` 来表示它。

11.3.1 C++ 程序库的作者的生存环境

假设你是一个 `shared library` 的维护者，你的 `library` 被公司另外的两三个团队使用了。你发现了一个安全漏洞，或者某个会导致 `crash` 的 `bug` 需要紧急修复，那么你修复之后，能不能直接部署 `library` 的二进制文件？有没有破坏二进制兼容性？会不会破坏别人团队已经编译好的投入生成环境的可执行文件？是不是要强迫别的团队重新编译链接，把可执行文件也发布新版本？会不会打乱别人的 `release cycle`？这些都是工程开发中经常要遇到的问题。

如果你打算新写一个 C++ `library`，那么通常要做以下几个决策：

- 以什么方式发布？动态库还是静态库？（本节不考虑源代码发布这种情况，这其实和静态库类似。）
- 以什么方式暴露库的接口？可选的做法有：以全局（含 `namespace` 级别）函数为接口、以 `class` 的 `non-virtual` 成员函数为接口、以 `virtual` 函数为接口。

Java 程序员不需要考虑这么多，直接写 `class` 成员函数就行，最多考虑一下要不要给 `method` 或 `class` 标上 `final`。也不必考虑什么动态库、静态库，都是 `.jar` 文件。

在作出上面两个决策之前，我们考虑两个基本假设：

- 代码会有 `bug`，库也不例外。将来可能会发布 `bug fixes`。
- 会有新的功能需求。写代码不是一锤子买卖，总是会有新的需求冒出来，需要程序员往库里增加东西。这是好事情，让程序员不丢饭碗。

也就是说，在设计库的时候必须要考虑将来如何升级。如果你的代码第一次发布的时候就已经做到完美，将来不需要任何修改，那么怎么做都行，也就不必继续阅读本节内容了。

基于以上两个基本假设来做决定。第一个决定很好做，如果需要 `hot fix`，那么只能用动态库；否则，在分布式系统中使用静态库更容易部署，这在前面已经谈过。“动态库比静态库节约内存”这种优势在今天看来已不太重要。

下面假定你或者你的老板选择以动态库方式发布，即发布 `.so` 或 `.dll` 文件，来看看第二个决定怎么做。再说一句，如果你能够以静态库方式发布，后面的麻烦都不会遇到。

第二个决定不那么容易做，关键问题是，要选择一种可扩展的（`extensible`）接口风格，让库的升级变得更轻松。“升级”有两层意思：

- 对于 `bug fix only` 的升级，二进制库文件的替换应该兼容现有的二进制可执行文件。二进制兼容性方面的问题已经在前面谈过，这里从略。
- 对于新增功能的升级，应该对客户代码友好。升级库之后，客户端使用新功能的代价应该比较小。只需要包含新的头文件（这一步可以省略，如果新功能已经加入原有的头文件中），然后编写新代码即可。而且，不要在客户代码中留下垃圾，后面我们会谈到什么是垃圾。

在讨论虚函数接口的弊端之前，我们先看看虚函数做接口的常见用法。

11.3.2 虚函数作为库的接口的两大用途

虚函数作为接口大致有这么两种用法：

- **调用**，也就是库提供一个什么功能（比如绘图 `Graphics`），以虚函数为接口方式暴露给客户端代码。客户端代码一般不需要继承这个 `interface`，而是直接调用其 `member function`。这么做据说是有利于接口和实现分离，我认为纯属多此一举、自欺欺人。
- **回调**，也就是事件通知，比如网络库的“连接建立”、“数据到达”、“连接断开”等等。客户端代码一般会继承这个 `interface`，然后把对象实体注册到库里边，等库来回调自己。一般来说客户端不会自己去调用这些 `member function`，除非是为了写单元测试模拟库的行为。
- **混合**，一个 `class` 既可以被客户端代码继承用作回调，又可以被客户端直接调用。说实话我没看出这么做的好处，但实际中某些面向对象的 C++ 库就是这么设计的。

对于“回调”方式，现代 C++ 有更好的做法，即 `boost::function + boost::bind`。`muduo` 的回调即采用这种新方法（§11.5）。以下不考虑以虚函数为回调的过时做法。

对于“调用”方式，这里举一个虚构的图形库来说明问题。这个库的功能是画线、画矩形、画圆弧：

```
struct Point
{
    int x;
    int y;
};
```

Linux 多线程服务端编程：使用 `muduo` C++ 网络库

```
class Graphics
{
    virtual void drawLine(int x0, int y0, int x1, int y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
    virtual void drawRectangle(Point p0, Point p1);

    virtual void drawArc(int x, int y, int r);
    virtual void drawArc(Point p, int r);
};
```

这里略去了很多与本文主题无关的细节，比如 Graphics 的构造与析构、draw*() 函数应该是 public、Graphics 应该不允许复制，还比如 Graphics 可能会用 pure virtual functions 等等，这些都不影响本文的讨论。

这个 Graphics 库的使用很简单，客户端看起来是这个样子。

```
Graphics* g = getGraphics();
g->drawLine(0, 0, 100, 200);
releaseGraphics(g);
```

似乎一切都很好，阳光明媚，符合“面向对象的原则”，但是一旦考虑升级，前景立刻变得昏暗。

11.3.3 虚函数作为接口的弊端

以虚函数作为接口在二进制兼容性方面有本质困难：“一旦发布，不能修改”。

假如我需要给 Graphics 增加几个绘图函数，同时保持二进制兼容性。这几个新函数的坐标以浮点数表示，我理想中的新接口是：

```
--- old/graphics.h 2011-03-12 13:12:44.000000000 +0800
+++ new/graphics.h 2011-03-12 13:13:30.000000000 +0800
class Graphics
{
    virtual void drawLine(int x0, int y0, int x1, int y1);
+   virtual void drawLine(double x0, double y0, double x1, double y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
+   virtual void drawRectangle(double x0, double y0, double x1, double y1);
    virtual void drawRectangle(Point p0, Point p1);

    virtual void drawArc(int x, int y, int r);
+   virtual void drawArc(double x, double y, double r);
    virtual void drawArc(Point p, int r);
};
```


受 C++ 二进制兼容性方面的限制，我们不能这么做。其本质问题在于 C++ 以 `vtable[offset]` 方式实现虚函数调用，而 `offset` 又是根据虚函数声明的位置隐式确定的，这造成了脆弱性。我增加了 `drawLine(double x0, double y0, double x1, double y1)`，造成 `vtable` 的排列发生了变化，现有的二进制可执行文件无法再用旧的 `offset` 调用到正确的函数。

怎么办呢？有一种危险且丑陋的做法，即把新的虚函数放到 `interface` 的末尾：

```
--- old/graphics.h 2011-03-12 13:12:44.000000000 +0800
+++ new/graphics.h 2011-03-12 13:58:22.000000000 +0800
class Graphics
{
    virtual void drawLine(int x0, int y0, int x1, int y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
    virtual void drawRectangle(Point p0, Point p1);

    virtual void drawArc(int x, int y, int r);
    virtual void drawArc(Point p, int r);
+
+ virtual void drawLine(double x0, double y0, double x1, double y1);
+ virtual void drawRectangle(double x0, double y0, double x1, double y1);
+ virtual void drawArc(double x, double y, double r);
};
```

这么做很丑陋，因为新的 `drawLine(double x0, double y0, double x1, double y1)` 函数没有和原来的 `drawLine()` 函数待在一起，造成了阅读上的不便。这么做同时很危险，因为 `Graphics` 如果被继承，那么新增虚函数会改变派生类中的 `vtable offset` 变化，同样不是二进制兼容的。

另外有两种似乎安全的做法，这也是 COM 采用的办法：

1. 通过链式继承来扩展现有的 `interface`，例如从 `Graphics` 派生出 `Graphics2`。

```
--- graphics.h 2011-03-12 13:12:44.000000000 +0800
+++ graphics2.h 2011-03-12 13:58:35.000000000 +0800
class Graphics
{
    virtual void drawLine(int x0, int y0, int x1, int y1);
    virtual void drawLine(Point p0, Point p1);

    virtual void drawRectangle(int x0, int y0, int x1, int y1);
    virtual void drawRectangle(Point p0, Point p1);

    virtual void drawArc(int x, int y, int r);
    virtual void drawArc(Point p, int r);
};
```

```

+
+class Graphics2 : public Graphics
+{
+  using Graphics::drawLine;
+  using Graphics::drawRectangle;
+  using Graphics::drawArc;
+
+  // added in version 2
+  virtual void drawLine(double x0, double y0, double x1, double y1);
+  virtual void drawRectangle(double x0, double y0, double x1, double y1);
+  virtual void drawArc(double x, double y, double r);
+};

```

将来如果继续增加功能，那么还会有 `class Graphics3 : public Graphics2`，以及 `class Graphics4 : public Graphics3` 等等。这么做和前面的做法一样丑陋，因为新的 `drawLine(double x0, double y0, double x1, double y1)` 函数位于派生 `Graphics2` interface 中，没有和原来的 `drawLine()` 函数待在一起，造成了割裂。

2. 通过多重继承来扩展现有的 interface，例如定义一个与 `Graphics` class 有同样成员的 `Graphics2`，再让实现同时继承这两个 interface。

```

--- graphics.h 2011-03-12 13:12:44.000000000 +0800
+++ graphics2.h 2011-03-12 13:16:45.000000000 +0800

class Graphics
{
  virtual void drawLine(int x0, int y0, int x1, int y1);
  virtual void drawLine(Point p0, Point p1);

  virtual void drawRectangle(int x0, int y0, int x1, int y1);
  virtual void drawRectangle(Point p0, Point p1);

  virtual void drawArc(int x, int y, int r);
  virtual void drawArc(Point p, int r);
};
+
+class Graphics2
+{
+  virtual void drawLine(int x0, int y0, int x1, int y1);
+  virtual void drawLine(double x0, double y0, double x1, double y1);
+  virtual void drawLine(Point p0, Point p1);
+
+  virtual void drawRectangle(int x0, int y0, int x1, int y1);
+  virtual void drawRectangle(double x0, double y0, double x1, double y1);
+  virtual void drawRectangle(Point p0, Point p1);
+
+  virtual void drawArc(int x, int y, int r);
+  virtual void drawArc(double x, double y, double r);
+  virtual void drawArc(Point p, int r);
+};

```

```
+// 在实现中采用多重接口继承
+class GraphicsImpl : public Graphics, // version 1
+                    public Graphics2, // version 2
+{
+ // ...
+};
```

这种带版本的 interface 的做法在 COM 使用者的眼中看起来是很正常的（比如 IXMLDOMDocument、IXMLDOMDocument2、IXMLDOMDocument3，又比如 ITaskbarList、ITaskbarList2、ITaskbarList3、ITaskbarList4 等等），这解决了二进制兼容性的问题，客户端源代码也不受影响。

在我看来带版本的 interface 实在是很丑陋，因为每次改动都引入了新的 interface class，会造成日后客户端代码难以管理。比如，如果新版应用程序的代码使用了 Graphics3 的功能，要不要把现有代码中出现的 Graphics2 都替换掉？

- 如果不替换，一个程序同时依赖多个版本的 Graphics，一直背着历史包袱。依赖的 Graphics 版本越积越多，将来如何管理得过来？
- 如果要替换，为什么不相干的代码（现有的运行得好好的使用 Graphics2 的代码）也会因为别处用到了 Graphics3 而被修改？

这种两难境地纯粹是“以虚函数为库的接口”造成的。如果我们能直接原地扩充 class Graphics，就不会有这些麻烦事，见 §11.4 “动态库接口的推荐做法”。

11.3.4 假如 Linux 系统调用以 COM 接口方式实现

或许上面这个 Graphics 的例子太简单，没有让“以虚函数为接口”的缺点充分暴露出来，下面让我们看一个真实的案例：Linux Kernel。

Linux kernel 从 0.01 的 67 个系统调用⁸发展到 2.6.37 的 340 个系统调用⁹，kernel interface 一直在扩充，而且保持良好的兼容性，它保持兼容性的办法很土，就是给每个 system call 赋予一个终身不变的数字代号，等于把虚函数表的排列固定下来。打开脚注中的两个链接，你就能看到 fork() 在 Linux 0.01 和 Linux 2.6.37 里的代号都是 2。（系统调用的编号跟硬件平台有关，这里我们看的是 x86 32-bit 平台。）

试想假如 Linux 当初选择用 COM 接口的链式继承风格来描述，将会是怎样一种壮观的景象？为了避免扰乱视线，请移步观看近百层继承的代码¹⁰。

⁸ <http://lxr.linux.no/linux-old+v0.01/include/unistd.h#L60>

⁹ http://lxr.linux.no/linux+v2.6.37.3/arch/x86/include/asm/unistd_32.h

¹⁰ <https://gist.github.com/867174> （先后关系与版本号不一定 100% 准确，我是用 git blame 去查的，现在列出的代码只从 0.01 到 2.5.31，相信已经足以展现 COM 接口方式的弊端。）

不要误认为“接口一旦发布就不能更改”是天经地义的，那不过是“以 C++ 虚函数为接口”的固有弊端，如果跳出这个框框去思考，其实 C++ 库的接口很容易做得更好。为什么不能改？还不是因为用了 C++ 虚函数作为接口。Java 的 interface 可以添加新函数，C 语言的库也可以添加新的全局函数，C++ class 也可以添加新 non-virtual 成员函数和 namespace 级别的 non-member 函数，这些都不需要继承出新 interface 就能扩充原有接口。偏偏 COM 的 interface 不能原地扩充，只能通过继承来 workaround，产生一堆带版本的 interfaces。有人说 COM 是二进制兼容性的正面例子，某深不以为然。COM 确实以一种最丑陋的方式做到了“二进制兼容”。脆弱和僵硬就是以 C++ 虚函数为接口的宿命。

相反，Linux 系统调用的编号以编译期常数方式固定下来，万年不变，轻而易举地解决了这个问题。在其他面向对象语言（Java/C#）中，我也没有见过每改动一次就给 interface 递增版本号号的诡异做法。

还是应了《The Zen of Python》中的那句话：“Explicit is better than implicit, Flat is better than nested.”

11.3.5 Java 是如何应对的

Java 实际上把 C/C++ 的 linking 这一步骤推迟到 class loading 的时候来做。就不存在“不能增加虚函数”，“不能修改 data member”等问题。在 Java 中用面向 interface 编程远比 C++ 更通用和自然，也没有上面提到的“僵硬的接口”问题。

11.4 动态库接口的推荐做法

取决于动态库的使用范围，有两类做法。

其一，如果动态库的使用范围比较窄，比如本团队内部的两三个程序在用，用户都是受控的，要发布新版本也更容易协调，那么不用太费事，只要做好发布的版本管理就行了。再在可执行文件中使用 rpath 把库的完整路径确定下来。

比如现在 Graphics 库发布了 1.1.0 和 1.2.0 两个版本，这两个版本可以不必是二进制兼容的。用户的代码从 1.1.0 升级到 1.2.0 的时候要重新编译一下，反正他们要用新功能都是要重新编译代码的。如果要原地打补丁，那么 1.1.1 应该和 1.1.0 二进制兼容，而 1.2.1 应该和 1.2.0 兼容。如果要加入新的功能，而新的功能与 1.2.0 不兼容，那么应该发布到 1.3.0 版本。

为了便于检查二进制兼容性，可考虑把库的代码的暴露情况分辨清楚。`muduo` 的头文件和 `class` 就有意识地分为用户可见和用户不可见两部分 (§6.3)。对于用户可见的部分，升级时要注意二进制兼容性，选用合理的版本号；对于用户不可见的部分，在升级库的时候就不必在意。另外 `muduo` 本身设计来是以源文件方式发布的，在二进制兼容性方面没有做太多的考虑。

其二，如果库的使用范围很广，用户很多，各家的 `release cycle` 不尽相同，那么推荐 `pimpl` 技法^[CCS, 条款 43]，并考虑多采用 `non-member non-friend function in namespace` ^{[EC3, 条款 23] [CCS, 条款 44 和 57]} 作为接口。这里以前面的 `Graphics` 为例，说明 `pimpl` 的基本手法。

1. 暴露的接口里边不要有虚函数，要显式声明构造函数、析构函数，并且不能 `inline`，原因见 §10.3.2。另外 `sizeof(Graphics) == sizeof(Graphics::Impl*)`。

```

class Graphics
{
public:
    Graphics(); // outline ctor
    ~Graphics(); // outline dtor

    void drawLine(int x0, int y0, int x1, int y1);
    void drawLine(Point p0, Point p1);

    void drawRectangle(int x0, int y0, int x1, int y1);
    void drawRectangle(Point p0, Point p1);

    void drawArc(int x, int y, int r);
    void drawArc(Point p, int r);

private:
    class Impl; // 头文件只放声明
    boost::scoped_ptr<Impl> impl;
};

```

graphics.h

2. 在库的实现中把调用转发（`forward`）给实现 `Graphics::Impl`，这部分代码位于 `.so/.dll` 中，随库的升级一起变化。

```

#include <graphics.h>

class Graphics::Impl
{
public:
    void drawLine(int x0, int y0, int x1, int y1);
    void drawLine(Point p0, Point p1);

```

graphics.cc

```

    void drawRectangle(int x0, int y0, int x1, int y1);
    void drawRectangle(Point p0, Point p1);

    void drawArc(int x, int y, int r);
    void drawArc(Point p, int r);
};

Graphics::Graphics()
    : impl(new Impl)
{
}

Graphics::~Graphics()
{
}

void Graphics::drawLine(int x0, int y0, int x1, int y1)
{
    impl->drawLine(x0, y0, x1, y1);
}

void Graphics::drawLine(Point p0, Point p1)
{
    impl->drawLine(p0, p1);
}

// ...

```

graphics.cc

3. 如果要加入新的功能，不必通过继承来扩展，可以原地修改，且很容易保持二进制兼容性。先动头文件：

```

--- old/graphics.h      2011-03-12 15:34:06.000000000 +0800
+++ new/graphics.h      2011-03-12 15:14:12.000000000 +0800

class Graphics
{
public:
    Graphics(); // outline ctor
    ~Graphics(); // outline dtor

    void drawLine(int x0, int y0, int x1, int y1);
+ void drawLine(double x0, double y0, double x1, double y1);
    void drawLine(Point p0, Point p1);

    void drawRectangle(int x0, int y0, int x1, int y1);
+ void drawRectangle(double x0, double y0, double x1, double y1);
    void drawRectangle(Point p0, Point p1);

    void drawArc(int x, int y, int r);
+ void drawArc(double x, double y, double r);
    void drawArc(Point p, int r);
}

```

```
private:
    class Impl;
    boost::scoped_ptr<Impl> impl;
};
```

然后在实现文件里增加 forward，这么做不会破坏二进制兼容性，因为增加 non-virtual 函数不影响现有的可执行文件。

```
--- old/graphics.cc    2011-03-12 15:15:20.000000000 +0800
+++ new/graphics.cc    2011-03-12 15:15:26.000000000 +0800
@@ -1,35 +1,43 @@
#include <graphics.h>

class Graphics::Impl
{
public:
    void drawLine(int x0, int y0, int x1, int y1);
+ void drawLine(double x0, double y0, double x1, double y1);
    void drawLine(Point p0, Point p1);

    void drawRectangle(int x0, int y0, int x1, int y1);
+ void drawRectangle(double x0, double y0, double x1, double y1);
    void drawRectangle(Point p0, Point p1);

    void drawArc(int x, int y, int r);
+ void drawArc(double x, double y, double r);
    void drawArc(Point p, int r);
};

Graphics::Graphics()
: impl(new Impl)
{
}

Graphics::~Graphics()
{
}

void Graphics::drawLine(int x0, int y0, int x1, int y1)
{
    impl->drawLine(x0, y0, x1, y1);
}

+void Graphics::drawLine(double x0, double y0, double x1, double y1)
+{
+    impl->drawLine(x0, y0, x1, y1);
+}
+
+void Graphics::drawLine(Point p0, Point p1)
+{
+    impl->drawLine(p0, p1);
+}
```

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，似乎有不只一个分类适合它。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 (const 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，覆盖 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

```
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

network library

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

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

user code

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));
}
```

11.6 iostream 的用途与局限

本节主要考虑 x86 Linux 平台，不考虑跨平台的可移植性，也不考虑国际化 (i18n)，但是要考虑 32-bit 和 64-bit 的兼容性。本节以 `stdio` 指代 C 语言的 `scanf/printf` 系列格式化输入输出函数。本节提及的“C 语言”（包括库函数和线程安全性），指的是 Linux 下 `gcc + glibc` 这一套编译器和库的具体实现，也可以认为是符合 POSIX.1-2001 的实现。本节要注意区分“编程初学者”和“C++ 初学者”，二者含义不同。

C++ `iostream` 的主要作用是让初学者有一个方便的命令行输入输出试验环境，在真实的项目中很少用到 `iostream`，因此不必把精力花在深究 `iostream` 的格式化与 `manipulator`（格式操控符）上。`iostream` 的设计初衷是提供一个可扩展的类型安全的 IO 机制，但是后来莫名其妙地加入了 `locale` 和 `facet` 等累赘。其整个设计复杂不堪，多重 + 虚拟继承的结构也很“巴洛克”，性能方面几无亮点。`iostream` 在实际项目中的用处非常有限，为此投入过多的学习精力实在不值。

11.6.1 stdio 格式化输入输出的缺点

对编程初学者不友好

看看下面这段简单的输入输出代码，这是 C 语言教学的基本示例。

```
#include <stdio.h>

int main()
{
    int i;
    short s;
    float f;
    double d;
    char name[80];

    scanf("%d %hd %f %lf %s", &i, &s, &f, &d, name);
    printf("%d %d %f %f %s\n", i, s, f, d, name);
}
```

注意到其中

- 输入和输出用的格式字符串不一样。输入 `short` 要用 `%hd`，输出用 `%d`；输入 `double` 要用 `%lf`，输出用 `%f`。
- 输入的参数不统一。对于 `i`、`s`、`f`、`d` 等变量，在传入 `scanf()` 的时候要取地址 (`&`)；而对于字符数组 `name`，则不用取地址。读者可以试一试如何用几句话向

刚开始学编程的初学者解释上面两条背后的原因（涉及传递函数不定参数时的类型转换、函数调用栈的内存布局、指针的意义、字符数组退化为字符指针等等）。如果一开始解释不清，只好告诉初学者“这是规定”，弄得人一头雾水。

- 缓冲区溢出的危险。上面的例子在读入 `name` 的时候没有指定大小，这是用 C 语言编程的安全漏洞的主要来源。应该在一开始就强调正确的做法，避免养成错误的习惯。

正确而安全的做法如下所示：²⁰

```
int main()
{
    const int max_name = 80;
    char name[max_name];

    char fmt[10];
    sprintf(fmt, "%%ds", max_name - 1);
    scanf(fmt, name);
    printf("%s\n", name);
}
```

这个动态构造格式化字符串的做法恐怕更难向初学者解释。

安全性 (security)

C 语言的安全性问题近十几年来引起了广泛的注意，C99 增加了 `snprintf()` 等能够指定输出缓冲区大小的函数，输出方面的安全性问题已经得到解决；输入方面似乎没有太大进展，还要靠程序员自己动手。

考虑一个简单的编程任务：从文件或标准输入读入一行字符串，行的长度不确定。我发现竟然没有哪个 C 语言标准库函数能完成这个任务，除非自己动手。

首先，`gets()` 是错误的，因为不能指定缓冲区的长度。

其次，`fgets()` 也有问题。它能指定缓冲区的长度，所以是安全的。但是程序必须预设一个长度的最大值，这不满足题目要求“行的长度不确定”。另外，程序无法判断 `fgets()` 到底读了多少个字节。为什么？考虑一个文件的内容是 9 个字节的字符串 `"Chen\000Shuo"`，注意中间出现了 `'\0'` 字符，如果用 `fgets()` 来读取，客户端如何知道 `"\000Shuo"` 也是输入的一部分？毕竟 `strlen()` 只返回 4，而且整个字符串里没有 `'\n'` 字符。

最后，可以用 `glibc` 定义的 `getline(3)` 函数来读取不定长的“行”。这个函数能正确处理各种情况，不过它返回的是 `malloc()` 分配的内存，要求调用端自己 `free()`。

²⁰ http://www.stoustrup.com/new_learning.pdf

类型安全 (type-safety)

如果 `printf()` 的整数参数类型是 `int`、`long` 等内置类型，那么 `printf()` 的格式化字符串很容易写。但是如果参数类型是系统头文件里 `typedef` 的类型呢？

如果你想在程序中用 `printf()` 来打印日志，你能一眼看出下面这些类型该用 `"%d"`、`"%ld"`、`"%lld"` 中的哪一个来输出吗？你的选择是否同时兼容 32-bit 和 64-bit 平台？

- `clock_t`。这是 `clock(3)` 的返回类型。
- `dev_t`。这是 `mknod(3)` 的参数类型。
- `in_addr_t`、`in_port_t`。这是 `struct sockaddr_in` 的成员类型。
- `nfds_t`。这是 `poll(2)` 的参数类型。
- `off_t`。这是 `lseek(2)` 的参数类型，麻烦的是，这个类型与宏定义 `_FILE_OFFSET_BITS` 有关。
- `pid_t`、`uid_t`、`gid_t`。这是 `getpid(2)`/`getuid(2)`/`getgid(2)` 的返回类型
- `ptrdiff_t`。`printf()` 专门定义了 `"t"` 前缀来支持这一类型（即使用 `"%td"`）。
- `size_t`、`ssize_t`。这两个类型到处都在用。`printf()` 为此专门定义了 `"z"` 前缀来支持这两个类型（即使用 `"%zu"` 或 `"%zd"` 来打印）。
- `socklen_t`。这是 `bind(2)` 和 `connect(2)` 的参数类型。
- `time_t`。这是 `time(2)` 的返回类型，也是 `gettimeofday(2)` 和 `clock_gettime(2)` 的结构体参数的成员类型。

如果在 C 程序里要正确打印以上类型的整数，恐怕要费一番脑筋。《The Linux Programming Interface》的作者建议（3.6.2 节）先统一转换为 `long` 类型，再用 `"%ld"` 来打印；对于某些类型仍然需要特殊处理，比如 `off_t` 的类型可能是 `long long`。

另外，`int64_t` 在 32-bit 和 64-bit 平台上是不同的类型，为此，如果程序要打印 `int64_t` 变量，需要包含 `<inttypes.h>` 头文件，并且使用 `PRId64` 宏：

```
#include <stdio.h>
#define __STDC_FORMAT_MACROS
#include <inttypes.h>

int main()
{
    int64_t x = 100;
    printf("%" PRId64 "\n", x);
    printf("%06" PRId64 "\n", x);
}
```

muduo 的 `Timestamp` class 使用了 `PRId64`。Google C++ 编码规范也提到了 64-bit 兼容性。²¹

这些问题在 C++ 里都不存在，在这方面 `iostream` 是个进步。

C `stdio` 在类型安全方面原本还有一个缺点，即格式化字符串与参数类型不匹配会造成难以发现的 bug，不过现在的编译器已经能够检测很多这种错误（使用 `-Wall` 编译选项）：

```
int main()
{
    double d = 100.0;
    // warning: format '%d' expects type 'int', but argument 2 has type 'double'
    printf("%d\n", d);

    short s;
    // warning: format '%d' expects type 'int*', but argument 2 has type 'short int*'
    scanf("%d", &s);

    size_t sz = 1;
    // no warning
    printf("%zd\n", sz);
}
```

不可扩展

C `stdio` 的另外一个缺点是无法支持自定义的类型，比如我写了一个 `Date` class，我无法像打印 `int` 那样用 `printf()` 来直接打印 `Date` 对象。

```
struct Date
{
    int year, month, day;
};

Date date;
printf("%D", &date); // WRONG
```

`glibc` 放宽了这个限制，允许用户调用 `register_printf_function(3)` 注册自己的类型。当然，前提是与现有的格式字符不冲突（这其实大大限制了这个功能的用处，现实中也几乎没有人真的去用它）。^{22 23}

性能

C `stdio` 的性能方面有两个弱点。

²¹ http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#64-bit_Portability

²² <http://www.gnu.org/s/hello/manual/libc/Printf-Extension-Example.html>

²³ http://en.wikipedia.org/wiki/Printf#Custom_format_placeholders

1. 使用一种 little language（现在流行叫 DSL）来配置格式。这固然有利于紧凑性和灵活性，但损失了一点点效率。每次打印一个整数都要先解析 "%d" 字符串，大多数情况下这不是问题，某些场合则需要自己写整数到字符串的转换。
2. C locale 的负担。locale 指的是不同语种对“什么是空白”、“什么是字母”，“什么是小数点”有不同的定义（德语中小数点是逗号，不是句点）。C 语言的 printf()、scanf()、isspace()、isalpha()、ispunct()、strtod() 等等函数都和 locale 有关，而且可以在运行时动态更改 locale。就算是程序只使用默认的“C” locale，仍然要为此付出代价。

11.6.2 iostream 的设计初衷

iostream 的设计初衷包括克服 C stdio 的缺点，提供一个高效的可扩展的类型安全的 IO 机制。“可扩展”有两层意思：一是可以扩展到用户自定义类型，二是通过继承 iostream 来定义自己的 stream。本文把前一种称为“类型可扩展”，把后一种称为“功能可扩展”。

类型可扩展和类型安全

“类型可扩展”和“类型安全”都是通过函数重载来实现的。

iostream 对初学者很友好，用 iostream 重写与前面同样功能的代码：

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int i;
    short s;
    float f;
    double d;
    string name;

    cin >> i >> s >> f >> d >> name;
    cout << i << " " << s << " " << f << " " << d << " " << name << endl;
}
```

这段代码恐怕比 scanf/printf 版本容易解释得多，而且没有安全性（security）方面的问题。

我们自己的类型也可以融入 iostream，使用起来与 built-in 类型没有区别。这主要得力于 C++ 可以定义 non-member functions/operators。

Linux 多线程服务端编程：使用 muduo C++ 网络库

```
#include <ostream> // 是不是太重量级了？

class Date
{
public:
    Date(int year, int month, int day)
        : year_(year), month_(month), day_(day)
    { }

    void writeTo(std::ostream& os) const
    {
        os << year_ << '-' << month_ << '-' << day_;
    }

private:
    int year_, month_, day_;
};

std::ostream& operator<<(std::ostream& os, const Date& date)
{
    date.writeTo(os);
    return os;
}

int main()
{
    Date date(2011, 4, 3);
    std::cout << date << std::endl;
}
```

`iostream` 凭借这两点（类型安全和类型可扩展），基本克服了 `stdio` 在使用上的不便与不安全。如果 `iostream` 止步于此，那它将是一个非常便利的库，可惜它前进了另外一步。

`iostream` 的演变大致可分为三个阶段。第一阶段是 Bjarne Stroustrup 在 CFront 1.0 里实现的 `streams` 库²⁴。这个库符合前述“类型安全、可扩展、高效”等特征，Bjarne 发明了用移位操作符（<< 和 >>）做 I/O 的办法，`istream` 和 `ostream` 都是具体类，也没有 `manipulator`。第二阶段，Jerry Schwarz 设计了“经典”`iostream`，在 CFront 2.0 中他的设计大部分得以体现。他发明了 `manipulator`，实现手法是以函数指针参数来重载输入输出操作符；他还采用多重继承和虚拟继承手法，设计了现在我们看到的 `ios` 菱形继承体系；此外，`istream` 有了基类 `ios`，也有了派生类 `ifstream` 和 `istrstream`，`ostream` 也是如此。第三阶段，在 C++ 标准化的过程中，`iostream` 有大幅更新，Nathan Myers 设计了 `Locale/Facet` 体系，`iostream` 被模板化以适应宽窄两种字符，以及以 `stringstream` 替换 `strstream` 等。

²⁴ http://www.softwarepreservation.org/projects/c_plus_plus/cfront/release_1.0/src/cfront/incl/stream.h/view

11.6.3 iostream 与标准库其他组件的交互

“值语义”与“对象语义”

不同于标准库其他 class 的“值语义 (value semantics)”，iostream 是“对象语义 (object semantics)”²⁵，即 iostream 是 non-copyable。这是正确的，因为如果 fstream 代表一个打开的文件的话，拷贝一个 fstream 对象意味着什么呢？表示打开了两个文件吗？如果销毁一个 fstream 对象，它会关闭文件句柄，那么另一个 fstream 对象副本会因此受影响吗？

iostream 禁止拷贝，利用对象的生命期来明确管理资源（如文件），很自然地就避免了这些问题。这就是 RAII，一种重要且独特的 C++ 编程手法。

C++ 同时支持“数据抽象 (data abstraction)”和“面向对象编程 (object-oriented)”，其实主要就是“值语义”与“对象语义”的区别，这是一个比较大的主题，见 §11.7。

std::string

iostream 可以与 std::string 配合得很好。但是有一个问题：谁依赖谁？

std::string 的 operator<< 和 operator>> 是如何声明的？注意 operator<< 是个二元操作符，它的参数是 std::ostream 和 std::string。<string> 头文件在声明这两个 operator 的时候要不要 #include <iostream>？

iostream 和 std::string 都可以单独 include 来使用，显然 iostream 头文件里不会定义 std::string 的 << 和 >> 操作。但是，如果 <string> 要 #include <iostream>，岂不是让 string 的用户被迫也用了 iostream？编译 iostream 头文件可是相当的慢啊（因为 iostream 是 template，其实现代码都放到了头文件中）。

标准库的解决办法是定义 <iosfwd> 头文件，其中包含 istream 和 ostream 等的前向声明（forward declarations），这样 <string> 头文件在定义输入输出操作符时就可以不必包含 <iostream>，只需要包含简短得多的 <iosfwd>，避免引入不必要的依赖。我们自己写程序也可借此学习如何支持可选的功能。

另外值得注意的是，istream::getline() 成员函数的参数类型是 char*，因为 <istream> 没有包含 <string>，而我们常用的 std::getline() 函数是个 non-member function，定义在 <string> 里边。

²⁵ 对象语义在其他面向对象的语言里通常叫做“引用语义 (reference semantics)”。为了避免与 C++ 的“引用”类型冲突，我这里用“对象语义”这个术语。

std::complex

标准库的复数类 `std::complex` 的情况比较复杂。`<complex>` 头文件会自动包含 `<sstream>`，后者会包含 `<istream>` 和 `<ostream>`，这是个不小的负担。问题是，为什么这么实现？

它的 `operator>>` 操作比 `string` 复杂得多，如何应对格式不正确的情况？输入字符串不会遇到格式不正确，但是输入一个复数则可能遇到各种问题，比如数字的格式不对等。有谁会真的在产品项目里用 `operator>>` 来读入字符方式表示的复数，这样的代码的健壮性如何保证？基于同样的理由，我认为产品代码中应该避免用 `istream` 来读取带格式的内容，后面也不再谈 `istream` 格式化输入的缺点，它已经落选。

它的 `operator<<` 也很奇怪，它不是直接使用参数 `ostream& os` 对象来输出，而是先构造 `ostringstream`，输出到该 `string stream`，再把结果字符串输出到 `ostream`。简化后的代码如下：

```
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::complex<T>& x)
{
    std::ostringstream s;
    s << '(' << x.real() << ',' << x.imag() << ')';
    return os << s.str();
}
```

注意到 `ostringstream` 会用到动态分配内存。也就是说，每输出一个 `complex` 对象就会分配释放一次内存，效率堪忧。

根据以上分析，我认为 `istream` 和 `complex` 配合得不好，但是它们耦合得更紧密（与 `string/istream` 相比），这可能是个不得已的技术限制吧（`complex` 是 `class template`，其 `operator<<` 必须在头文件中定义，而这个定义又用到了 `ostringstream`，不得已包含了 `sstream` 的实现）。

如果程序要对 `complex` 做 IO，从效率和健壮性方面考虑，建议不要使用 `istream`。

11.6.4 istream 在使用方面的缺点

在简单使用 `istream` 的时候，它确实比 `stdio` 方便，但是深入一点就会发现，二者可说各擅胜场。下面谈一谈 `istream` 在使用方面的缺点。

格式化输出很烦琐

ostream 采用 manipulator 来格式化，如果我想按照 2010-04-03 的格式输出前面定义的 Date class，那么代码要改成：

```
class Date
{
    // ...

    void writeTo(std::ostream& os) const
    {
-   os << year_ << '-' << month_ << '-' << day_;
+   os << year_ << '-'
+       << std::setw(2) << std::setfill('0') << month_ << '-'
+       << std::setw(2) << std::setfill('0') << day_;
    }

    // ...
}
```

假如用 stdio，会简短得多，因为 printf 采用了一种表达能力较强的小语言来描述输出格式。

```
class Date
{
    // ...

    void writeTo(std::ostream& os) const
    {
-   os << year_ << '-' << month_ << '-' << day_;
+   char buf[32];
+   snprintf(buf, sizeof buf, "%d-%02d-%02d", year_, month_, day_);
+   os << buf;
    }

    // ...
}
```

使用小语言来描述格式还带来了另外一个好处：外部可配置。

外部可配置性

能不能用外部的配置文件来定义程序中日期的格式？在 C stdio 中很好办，把格式字符串 "%d-%02d-%02d" 保存到配置里就行。但是 ostream 呢？它的格式是写在代码里的，灵活性大打折扣。

再举一个例子，程序的 message 的多语言化。

```
const char* name = "Shuo Chen";
int age = 29;
printf("My name is %1$s, I am %2$d years old.\n", name, age);
cout << "My name is " << name << ", I am " << age << " years old." << endl;
```

对于 `stdio`，要让这段程序支持中文的话，把代码中的“`My name is ...`”，替换为“`我叫%1$s，今年%2$d岁。\\n`”即可。也可以把这段提示语做成资源文件，在运行时读入。而对于 `iostream`，恐怕没有这么方便，因为代码是支离破碎的。

C `stdio` 的格式化字符串体现了重要的“数据就是代码”的思想，这种“数据”与“代码”之间的相互转换是程序灵活性的根源，远比 OO 更为灵活。

stream 的状态

如果我想用十六进制方式输出一个整数 `x`，那么可以用 `hex` 操控符，但是这会改变 `ostream` 的状态。比如说

```
int x = 8888;
cout << hex << showbase << x << endl; // print 0x22b8
cout << 123 << endl;                  // print 0x7b
```

这段代码会把 123 也按照十六进制方式输出，这恐怕不是我们想要的。

再举一个例子，`setprecision()` 也会造成持续影响：

```
double d = 123.45;
printf("%.3f\\n", d);
cout << d << endl;
cout << setw(8) << fixed << setprecision(3) << d << endl;
cout << d << endl;
```

输出是：

```
$ ./a.out
123.450    %8.3f 的输出
123.45     默认 cout 格式
123.450    我们设置的精度
123.450    精度持续影响后续输出
```

可见代码中的 `setprecision()` 影响了后续输出的精度。注意 `setw()` 不会造成影响，它只对下一个输出有效。

这说明，如果使用 `manipulator` 来控制格式，需要时刻小心以防影响了后续代码；而使用 C `stdio` 就没有这个问题，它是“上下文无关的”。

知识的通用性

在 C 语言之外，有其他很多语言也支持 `printf()` 风格的格式化，例如 Java、Perl、Ruby 等等²⁶。学会 `printf()` 的格式化方法，这个知识还可以用到其他语言中。但是 C++ `iostream` “只此一家，别无分店”。反正都是格式化输出，学习 `stdio` 投资回报率更高。

²⁶ http://en.wikipedia.org/wiki/Printf#Programming_languages_with_printf

基于这点考虑,我认为不必深究 istream 的格式化方法,只需要用好它最基本的类型安全输出即可。在真的需要格式化的场合,可以考虑 snprintf() 打印到栈上缓冲,再用 ostream 输出。

线程安全与原子性

istream 的另外一个问题是线程安全性。POSIX.1-2001 明确要求 stdio 函数是线程安全的²⁷,而且还提供了 flockfile(3)/funlockfile(3) 之类的函数来明确控制 FILE* 的加锁与解锁。

istream 在线程安全方面没有保证,就算单个 operator<< 是线程安全的,也不能保证原子性。因为 cout << a << b; 是两次函数调用,相当于 cout.operator<<(a).operator<<(b)。两次调用中间可能会被打断进行上下文切换,造成输出内容不连续,插入了其他线程打印的字符。而 fprintf(stdout, "%s %d", a, b); 是一次函数调用,而且是线程安全的,打印的内容不会受其他线程影响。因此,istream 并不适合在多线程程序中做 logging。

istream 的局限

根据以上分析,我们可以归纳 istream 的局限:

- 输入方面,istream 不适合输入带格式的数据,因为“纠错”能力不强,进一步的分析请见孟岩写的《契约思想的一个反面案例》,孟岩说“复杂的设计必然带来复杂的使用规则,而面对复杂的使用规则,用户是可以投票的,那就是:你做你的,我不用!”可谓鞭辟入里。如果要用 istream,我推荐的做法是用 std::getline() 读入一行数据到 std::string,然后用正则表达式来判断内容正误,并做分组,最后用 strtod()/strtol() 之类的函数做类型转换。这样似乎更容易写出健壮的程序。
- 输出方面,ostream 的格式化输出非常烦琐,而且写死在代码里,不如 stdio 的小语言那么灵活通用。建议只用作简单的无格式输出。
- log 方面,由于 ostream 没有办法在多线程程序中保证一行输出的完整性,建议不要直接用它来写 log。如果是简单的单线程程序,输出数据量较少的情况下可以酌情使用。产品代码应该用成熟的 logging 库,见第 5 章。
- in-memory 格式化方面,由于 ostringstream 会动态分配内存,它不适合性能要求较高的场合。

²⁷ <http://www.kernel.org/doc/man-pages/online/pages/man7/threads.7.html>

- 文件 IO 方面，如果用作文本文件的输入或输出，`fstream` 有上述的缺点；如果用作二进制数据的输入输出，那么自己简单封装一个 `File class` 似乎更好用，也不必为用不到的功能付出代价（后文还有具体例子）。`ifstream` 的一个用处是在程序启动时读入简单的文本配置文件。如果配置文件是其他文本格式的（XML 或 JSON），那么用相应的库来读，也用不到 `ifstream`。
- 性能方面，`iostream` 没有兑现“高效性”诺言²⁸。`iostream` 在某些场合比 `stdio` 快，在某些场合比 `stdio` 慢，对于性能要求较高的场合，我们应该自己实现字符串转换（见后文的代码与测试）。

既然有这么多局限，`iostream` 在实际项目中的应用就大为受限了，在这上面投入太多的精力实在不值得。说实话，我没有见过哪个 C++ 产品代码使用 `iostream` 来作为输入输出设施。Google 的 C++ 编程规范也对 `stream` 的使用做了明确的限制。²⁹

11.6.5 `iostream` 在设计方面的缺点

`iostream` 的设计有相当多的 WTFs³⁰，stackoverflow 有人抱怨说：“If you had to judge by today’s software engineering standards, would C++’s IOStreams still be considered well-designed?”³¹

面向对象的设计

`iostream` 是个面向对象的 IO 类库，本节简单介绍它的继承体系。对 `iostream` 略有了解的人 would 知道它用了多重继承和虚拟继承，简单地画个类图如下（见图 11-1），这是典型的菱形继承。

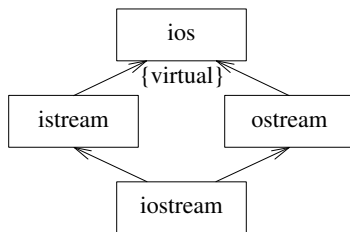


图 11-1

²⁸ 在线 ACM/ICPC 判题网站上，如果一个简单的偏重 IO 的题目发生超时错误，那么把其中 `iostream` 的输入输出换成 `stdio`，有时就能过关。另外可以先试试调用 `cin.sync_with_stdio(false)`；（<http://stackoverflow.com/questions/9371238>）。

²⁹ <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Streams>

³⁰ http://www.osnews.com/story/19266/WTFs_m

³¹ <http://stackoverflow.com/questions/2753060/who-architected-designed-cs-iostreams-and-would-it-still-be-considered-well>

如果加深一点了解，会发现 `iostream` 现在是模板化的，同时支持窄字符和宽字符。图 11-2 是现在的继承体系，同时画出了 `fstream(s)` 和 `stringstream(s)`。图 11-2 中方框的第二三行是模板的具现化类型，即我们代码里常用的具体类型（通过 `typedef` 定义）。这个继承体系糅合了面向对象与泛型编程，但可惜它两方面都不讨好。

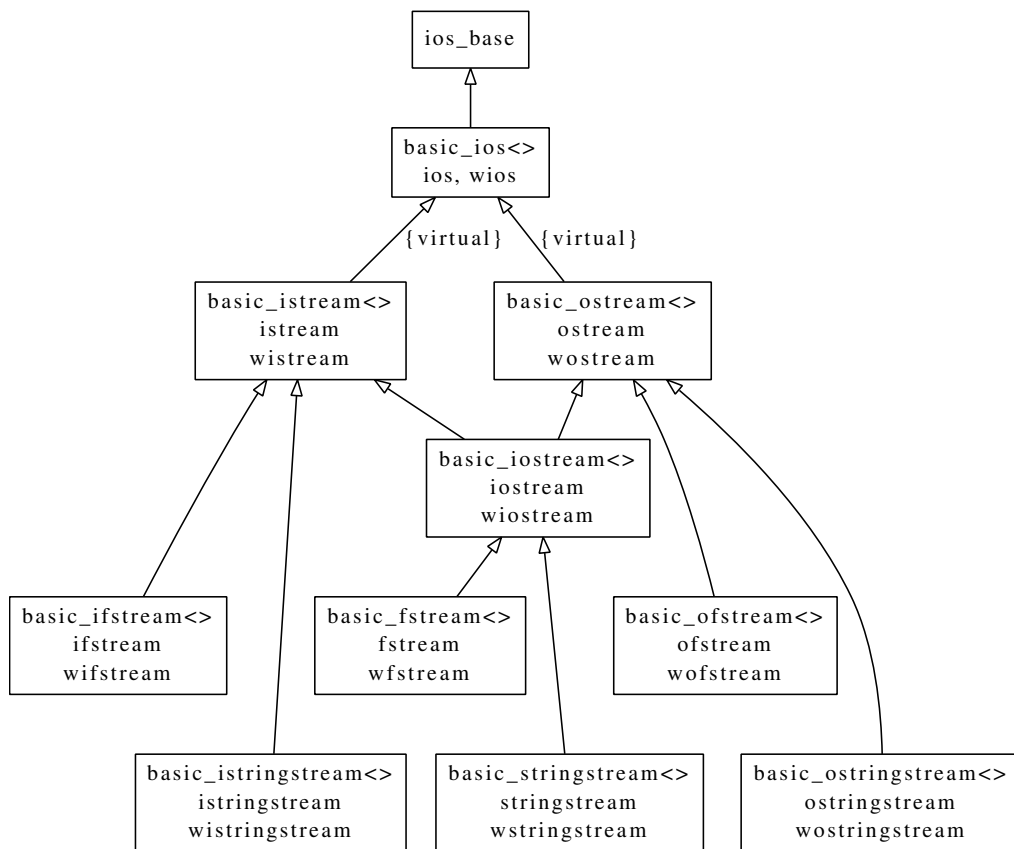


图 11-2

再进一步加深了解，发现还有一个平行的 `streambuf` 继承体系（见图 11-3），`fstream` 和 `stringstream` 的主要区别在于使用了不同的 `streambuf` 派生类型。

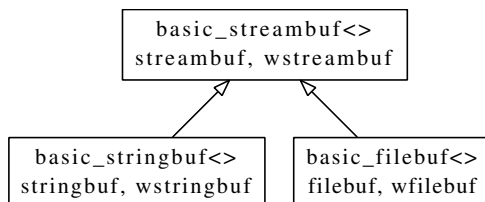


图 11-3

再把这两个继承体系画到一幅图里，如图 11-4 所示。

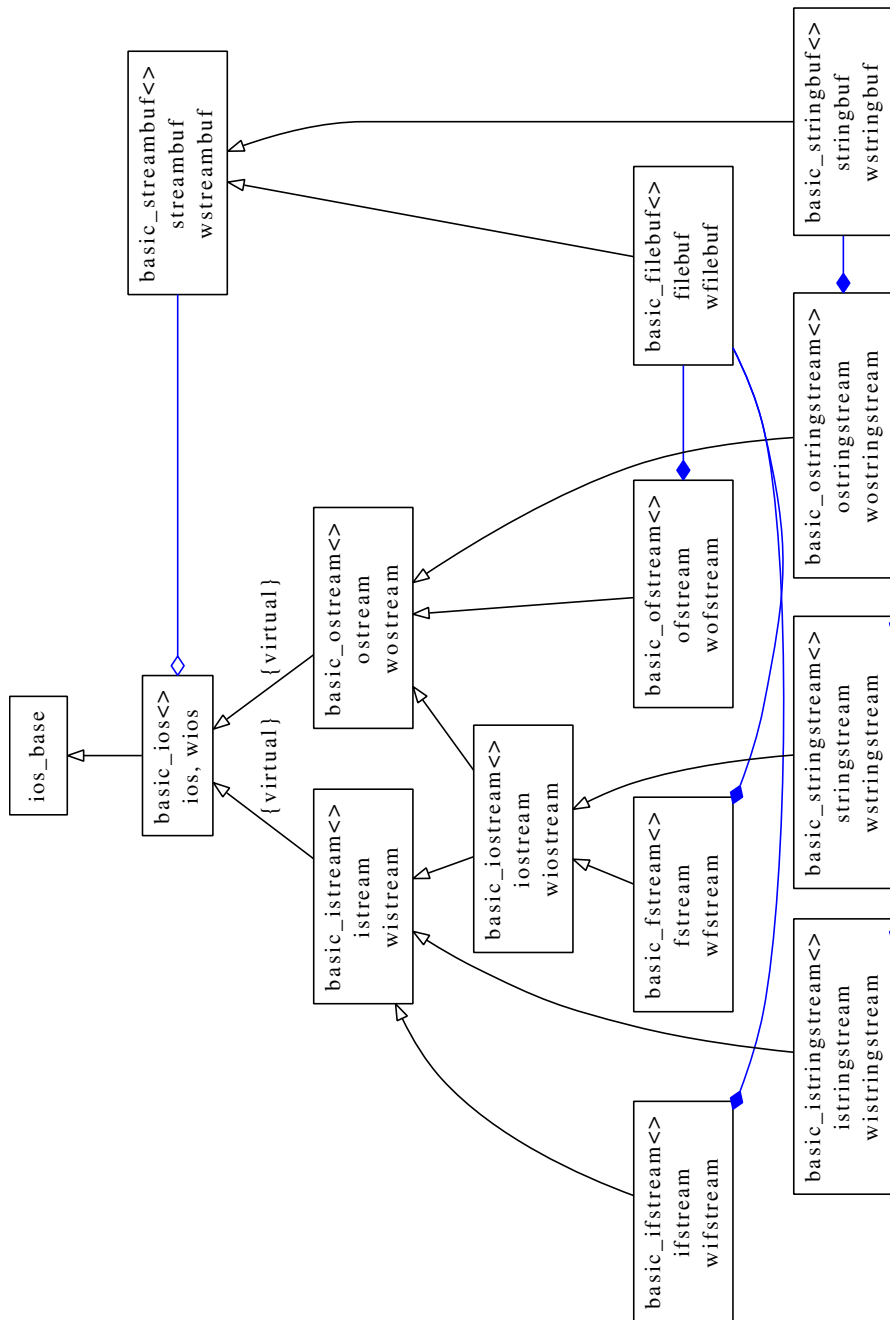


图 11-4

注意到 `basic_ios` 持有了 `streambuf` 的指针；而 `fstream(s)` 和 `stringstream(s)` 则分别包含 `filebuf` 和 `stringbuf` 的对象。看上去有点像 Bridge 模式。

看了这样“巴洛克”的设计，有没有人还打算在自己的项目中通过继承 `istream` 来实现自己的 `stream`，以实现功能扩展呢？

面向对象方面的设计缺陷

本节我们分析一下 `istream` 的设计违反了哪些 OO 准则。

我们知道，面向对象中的 `public` 继承需要满足 Liskov 替换原则，继承非为复用，乃为被复用³²。在程序里需要用到 `ostream` 的地方（例如 `operator<<`），我传入 `ofstream` 或 `ostringstream` 都应该能按预期工作，这就是 OO 继承强调的“可替换性”，派生类的对象可以替换基类对象，从而被客户端代码 `operator<<` 复用。

`istream` 的继承体系多次违反了 Liskov 原则，这些地方继承的目的是为了复用基类的代码，图 11-5 中我把违规的继承关系用虚线标出。

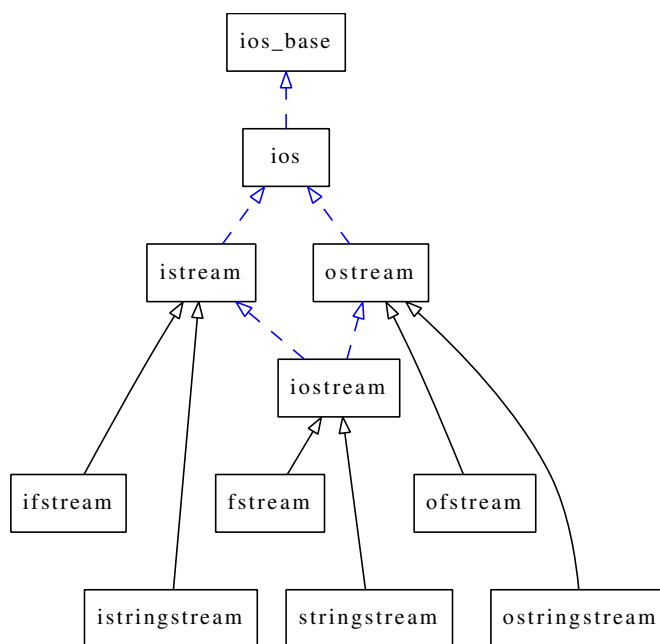


图 11-5

³² 《Effective C++ 中文版（第3版）》[EC3, 条款32]：确保你的 `public` 继承模型塑出 is-a 关系。《C++ 编程规范》[CCS, 条款37]：`public` 继承意味着可替换性。

在现有的继承体系中（见图 11-5），合理的有：³³

- ifstream **is-a** istream
- ofstream **is-a** ostream
- fstream **is-a** iostream
- istreamstringstream **is-a** istream
- ostreamstringstream **is-a** ostream
- stringstream **is-a** iostream

我认为不怎么合理的有：

- ios 继承 ios_base。有没有哪种情况下函数期待 ios_base 对象，但是客户可以传入一个 ios 对象替代之？如果没有，这里用 public 继承是不是违反 OO 原则？
- istream 继承 ios。有没有哪种情况下函数期待 ios 对象，但是客户可以传入一个 istream 对象替代之？如果没有，这里用 public 继承是不是违反 OO 原则？
- ostream 继承 ios。有没有哪种情况下函数期待 ios 对象，但是客户可以传入一个 ostream 对象替代之？如果没有，这里用 public 继承是不是违反 OO 原则？
- iostream 多重继承 istream 和 ostream。为什么 iostream 要同时继承两个 non-interface class？这是接口继承还是实现继承？是不是可以用组合（composition）来替代？³⁴

用组合替换继承之后的体系如图 11-6 所示。

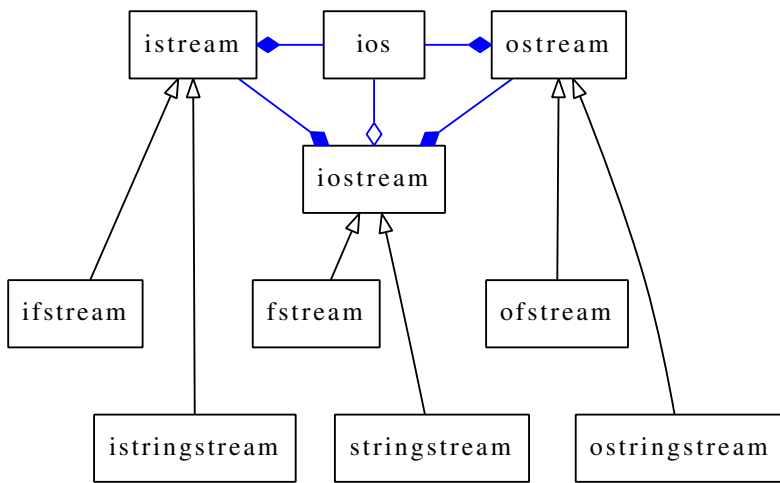


图 11-6

注意到在新的设计中，只有真正的 **is-a** 关系采用了 public 继承，其他均以组合来代替，组合关系以菱形箭头表示。新的设计没有使用虚拟继承或多重继承。

³³ 按英语语法，这里的 **is-a** 应该写作 **is-an**，此处从简。

³⁴ 见《Effective C++ 中文版（第 3 版）》[EC3，条款 38]：通过组合模塑出 has-a 或“以某物实现”。《C++ 编程规范》[CCS，条款 34]：尽可能以组合代替继承。

其中 istream 的新实现值得一提，代码结构如下：

```
class istream;
class ostream;

class istream
{
public:
    istream& get_istream();
    ostream& get_ostream();
    virtual ~istream();

    // ...
};
```

这样一来，在需要 istream 对象表现得像 istream 的地方，调用 get_istream() 函数返回一个 istream 的引用；在需要 istream 对象表现得像 ostream 的地方，调用 get_ostream() 函数返回一个 ostream 的引用。功能不受影响，而且代码更清晰，istream 和 ostream 也不必使用虚拟继承了。（我非常怀疑 istream class 的真正价值，一个东西既可读又可写，说明它是一个 sophisticated IO 对象，为什么还用这么厚的 OO 封装？）

阳春的 locale

istream 的故事还不止这些，它还包含一套阳春的 locale/facet 实现，这套实践中没人用的东西进一步增加了 istream 的复杂度，而且不可避免地影响其性能。Nathan Myers 正是其始作俑者³⁵。

ostream 自身定义的针对整数和浮点数的 operator<< 成员函数的函数体是：

```
ostream& ostream::operator<<(int val) // 或 double val
{
    bool failed =
        use_facet<num_put>(getloc()).put(
            ostreambuf_iterator(*this), *this, fill(), val).failed();
    // ...
}
```

它会调用 num_put::put()，后者会去调用 num_put::do_put()，而 do_put() 是个虚函数，没办法 inline。istream 在性能方面的不足恐怕部分来自于此。这个虚函数白白浪费了把 template 的实现放到头文件应得的好处，编译和运行速度都快不起来。这就是我说 istream 在泛型方面不讨好的原因。

我没有深入挖掘其中的细节，感兴趣的读者可以移步观看 facet 的继承体系：<http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00431.html>。

³⁵ <http://www.cantrip.org/locale.html>

据此分析，我不认为以 `iostream` 为基础的上层程序库（比方说那些克服 `iostream` 格式化方面的缺点的库）有多大的实用价值。

臆造抽象

孟岩评价“`iostream` 最大的缺点是臆造抽象”，我非常赞同他的观点。

这个评价同样适用于 Java 那一套“叠床架屋”的 `InputStream`、`OutputStream`、`Reader`、`Writer` 继承体系，.NET 也搞了这么一套繁文缛节。

乍看之下，用 `input stream` 表示一个可以“读”的数据流，用 `output stream` 表示一个可以“写”的数据流，屏蔽底层细节，面向接口编程，“符合面向对象原则”，似乎是一件美妙的事情。但是，真实的世界要残酷得多。

IO 是个极度复杂的东西，就拿最常见的 `memory stream`、`file stream`、`socket stream` 来说，它们之间的差异极大：

- 是单向 IO 还是双向 IO。只读或者只写？还是既可读又可写？
- 顺序访问还是随机访问。可不可以 `seek`？可不可以退回 `n` 字节？
- 文本数据还是二进制数据。输入数据格式有误怎么办？如何编写健壮的处理输入的代码？
- 有无缓冲。`write 500` 字节是否能保证完全写入？有没有可能只写入了 300 字节？余下 200 字节怎么办？
- 是否阻塞。会不会返回 `EWouldBlock` 错误？
- 有哪些出错的情况。这是最难的，`memory stream` 几乎不可能出错，`file stream` 和 `socket stream` 的出错情况完全不同。`socket stream` 可能遇到对方断开连接，`file stream` 可能遇到超出磁盘配额。

根据以上列举的初步分析，我不认为有办法设计一个公共的基类把各方面的情况都考虑周全。各种 IO 设施之间共性太小，差异太大，例外太多。如果硬要用面向对象来建模，基类要么太瘦（只放共性，这个基类包含的 `interface functions` 没多大用），要么太肥（把各种 IO 设施的特性都包含进来，这个基类包含的 `interface functions` 很多，但是不是每一个都能调用）。

一个基类设计得好，大家才愿意去继承它。比如 `Runnable` 是个很好的抽象，有不计其数的实现。`InputStream/OutputStream` 好歹也有若干个实现（见图 11-7）。反观 `istream/ostream`，只有标准库提供的两套默认实现，在项目中极少有人会去继承并扩展它，是不是说明 `istream/ostream` 这一套抽象不怎么好使呢？

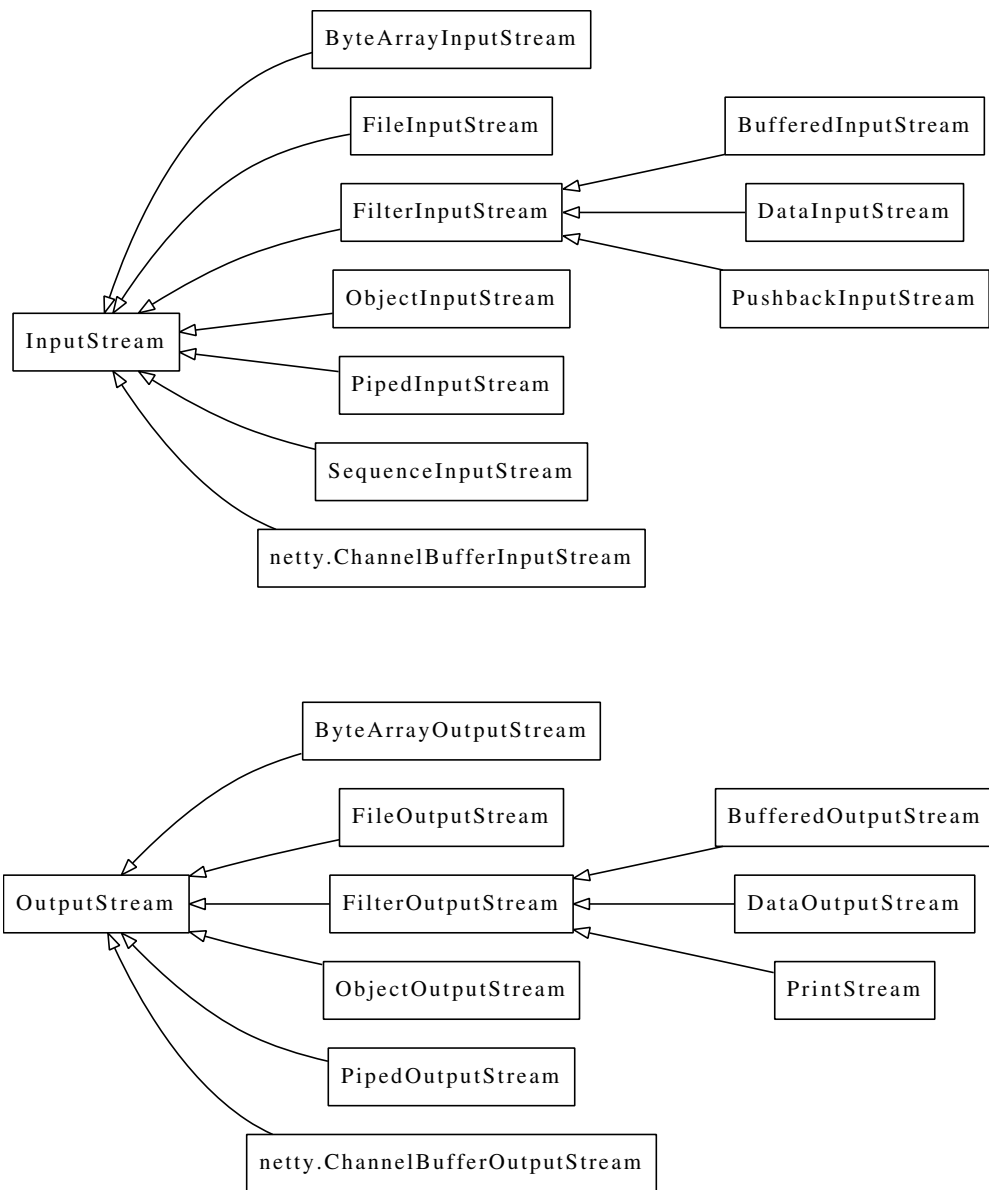


图 11-7

当然，假如 Java 有 C++ 那样强大的 `template` 机制，图 11-7 中的继承体系能简化不少。

若要在 C 语言里解决这个问题，通常的办法是用一个 `int` 表示 IO 对象（`file` 或 `PIPE` 或 `socket`），然后配以 `read()`/`write()`/`lseek()`/`fcntl()` 等一系列全局函数，程序员自己搭配组合。这个做法我认为比面向对象的方案要简洁高效。

`iostream` 在性能方面没有比 `stdio` 高多少，在健壮性方面多半不如 `stdio`，在灵活性方面受制于本身的复杂设计而难以让使用者自行扩展。目前看起来只适合一些简单的、要求不高的应用，但是又不得不为它的复杂设计付出运行时代价，总之，其定位有点不上不下。

在实际的项目中，我们可以提炼出一些简单高效的 `strip-down` 版本，在获得便利性的同时避免付出不必要的代价。

11.6.6 一个 300 行的 `memory buffer output stream`

我认为以 `operator<<` 来输出数据非常适合 `logging`（见第 5 章），因此写了一个简单的 `muduo::LogStream` class。代码不到 300 行，完全独立于 `iostream`，位于 `muduo/base/LogStream.{h,cc}`。

这个 `LogStream` 做到了类型安全和类型可扩展，效率也较高。它不支持定制格式化、不支持 `locale/facet`、没有继承、`buffer` 也没有继承与虚函数、没有动态分配内存、`buffer` 大小固定。简单地说，适合 `logging` 以及简单的字符串转换。这基本上是 Bjarne 在 1984 年写的 `ostream` 的翻版。

`LogStream` 的接口定义如下：

```
class Buffer;

class LogStream : boost::noncopyable
{
    typedef LogStream self;
public:

    self& operator<<(bool);

    self& operator<<(short);
    self& operator<<(unsigned short);
    self& operator<<(int);
    self& operator<<(unsigned int);
    self& operator<<(long);
    self& operator<<(unsigned long);
    self& operator<<(long long);
    self& operator<<(unsigned long long);

    self& operator<<(const void*);
```

```

self& operator<<(float);
self& operator<<(double);
// self& operator<<(long double);

self& operator<<(char);
// self& operator<<(signed char);
// self& operator<<(unsigned char);

self& operator<<(const char*);
self& operator<<(const string&);

void append(const char* data, int len);
const Buffer& buffer() const { return buffer_; }
void resetBuffer() { buffer_.reset(); }

private:
    Buffer buffer_;
};

```

LogStream 本身不是线程安全的，它不适合做线程间的共享对象。正确的方式是每条 log 消息构造一个 LogStream，用完就扔。LogStream 的成本极低，这么做不会有什么性能损失。

整数到字符串的高效转换

muduo::LogStream 的整数转换是自己写的，用的是 Matthew Wilson 的算法，见 §12.3 “带符号整数的除法与余数”。这个算法比 stdio 和 iostream 都要快。

浮点数到字符串的高效转换

目前 muduo::LogStream 的浮点数格式化采用的是 snprintf()。所以从性能上与 stdio 持平，比 ostream 快一些。

浮点数到字符串的转换是个复杂的话题，这个领域 20 年以来没有什么进展（目前的实现大都基于 David M. Gay 在 1990 年的工作：《Correctly Rounded Binary-Decimal and Decimal-Binary Conversions》，代码：<http://netlib.org/fp/>），直到 2010 年才有突破。

Florian Loitsch 发明了新的更快的算法 Grisu3，他的论文《Printing floating-point numbers quickly and accurately with integers》发表在 PLDI 2010，代码见 Google V8 引擎以及 <http://code.google.com/p/double-conversion/>。有兴趣的读者可以阅读这篇博客³⁶。

将来 muduo::LogStream 可能会改用 Grisu3 算法实现浮点数转换。

³⁶ <http://www.serpentine.com/blog/2011/06/29/here-be-dragons-advances-in-problems-you-didnt-even-know-you-had/>

性能对比

由于 `muduo::LogStream` 抛掉了很多负担，因此可以预见它的性能好于 `ostringstream` 和 `stdio`。我做了一个简单的性能测试，结果如表 11-1 和表 11-2 所示。表 11-1 和表 11-2 中的数字是打印 1 000 000 次的用时，以毫秒为单位，越小越好。

表 11-1 64-bit 测试结果

	snprintf	ostringstream	LogStream
int	499	363	113
double	2315	3835	2338
int64_t	486	347	145
void*	419	330	47

表 11-2 32-bit 测试结果

	snprintf	ostringstream	LogStream
int	544	453	116
double	2241	4030	2267
int64_t	725	958	654
void*	690	425	65

从表 11-1 和表 11-2 看出，`ostreamstream` 有时候比 `snprintf()` 快，有时候比它慢，`muduo::LogStream` 比它们两个都快得多（`double` 类型除外）。

泛型编程

其他程序库如何使用 `LogStream` 作为输出呢？办法很简单，用模板。

前面我们定义了 `Date` class 针对 `std::ostream` 的 `operator<<`，只要稍作修改就能同时适用于 `std::ostream` 和 `LogStream`。而且 `Date` 的头文件不再需要 `#include <ostream>`，降低了耦合。

```
// 不必包含 LogStream 或 ostream 头文件
class Date
{
public:
    Date(int year, int month, int day);

-   void writeTo(std::ostream& os) const
+   template<typename OStream>
+   void writeTo(OStream& os) const
    {
        char buf[32];
        snprintf(buf, sizeof buf, "%d-%02d-%02d", year_, month_, day_);
```

```

        os << buf;
    }

private:
    int year_, month_, day_;
};

-std::ostream& operator<<(std::ostream& os, const Date& date)
+template<typename OStream>
+OStream& operator<<(OStream& os, const Date& date)
{
    date.writeTo(os);
    return os;
}

```

格式化

`muduo::LogStream` 本身不支持格式化，不过我们很容易为它做扩展，定义一个简单的 `Fmt` class 就行，而且不影响 `stream` 的状态。

```

class Fmt : boost::noncopyable
{
public:
    template<typename T>
    Fmt(const char* fmt, T val)
    {
        BOOST_STATIC_ASSERT(boost::is_arithmetic<T>::value == true);
        length_ = snprintf(buf_, sizeof buf_, fmt, val);
    }

    const char* data() const { return buf_; }
    int length() const { return length_; }

private:
    char buf_[32];
    int length_;
};

inline LogStream& operator<<(LogStream& os, const Fmt& fmt)
{
    os.append(fmt.data(), fmt.length());
    return s;
}

```

使用方法：

```

LogStream os;
double x = 19.82;
int y = 43;
os << Fmt("%.3f", x) << Fmt("%4d", y);

```


11.6.7 现实的 C++ 程序如何做文件 IO

下面举三个例子，Google Protobuf Compiler、Google leveldb、Kyoto Cabinet。

Google Protobuf Compiler

Google Protobuf 是一种高效的网络传输格式，它用一种协议描述语言来定义消息格式，并且自动生成序列化代码。Protobuf Compiler 是这种“协议描述语言”的编译器，它读入协议文件 .proto，编译生成 C++、Java、Python 代码。proto 文件是个文本文件，然而 Protobuf Compiler 并没有使用 ifstream 来读取它，而是使用了自己的 FileInputStream 来读取文件。

大致代码流程如下：

1. ZeroCopyInputStream³⁷ 是一个抽象基类。
2. FileInputStream³⁸ 继承并实现了 ZeroCopyInputStream。
3. Tokenizer³⁹ 是词法分析器，它把 proto 文件分解为一个个字元（token）。Tokenizer 的构造函数以 ZeroCopyInputStream 为参数，从该 stream 读入文本。
4. Parser⁴⁰ 是语法分析器，它把 proto 文件解析为语法树，以 FileDescriptorProto 表示。Parser 的构造函数以 Tokenizer 为参数，从它读入字元。

由此可见，即便是读取文本文件，C++ 程序也不一定要用 ifstream。

Google leveldb

Google leveldb 是一个高效的持久化 key-value db。⁴¹ 它定义了三个精简的 interface 用于文件输入输出：

- SequentialFile
- RandomAccessFile
- WritableFile

接口函数如下：

³⁷ http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/io/zero_copy_stream.h#122

³⁸ http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/io/zero_copy_stream_impl.h#55

³⁹ <http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/io/tokenizer.h#75>

⁴⁰ <http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/compiler/parser.h#59>

⁴¹ <http://code.google.com/p/leveldb>

```

struct Slice {
    const char* data_;
    size_t size_;
};

// A file abstraction for reading sequentially through a file
class SequentialFile {
public:
    SequentialFile() { }
    virtual ~SequentialFile();

    virtual Status Read(size_t n, Slice* result, char* scratch) = 0;
    virtual Status Skip(uint64_t n) = 0;
};

// A file abstraction for randomly reading the contents of a file.
class RandomAccessFile {
public:
    RandomAccessFile() { }
    virtual ~RandomAccessFile();

    virtual Status Read(uint64_t offset, size_t n, Slice* result,
                        char* scratch) const = 0;
};

// A file abstraction for sequential writing. The implementation
// must provide buffering since callers may append small fragments
// at a time to the file.
class WritableFile {
public:
    WritableFile() { }
    virtual ~WritableFile();

    virtual Status Append(const Slice& data) = 0;
    virtual Status Close() = 0;
    virtual Status Flush() = 0;
    virtual Status Sync() = 0;
};

```

leveldb 明确区分 input 和 output，并进一步把 input 分为 sequential 和 random access，然后提炼出了三个简单的接口，每个接口只有屈指可数的几个函数。这几个接口在各个平台下的实现也非常简单明了^{42 43}，一看就懂。

注意这三个接口使用了虚函数，我认为这是正当的，因为一次 IO 往往伴随着系统调用和 context switch，虚函数的开销比起 context switch 来可以忽略不计。相反，iostream 每次 operator<<() 就调用虚函数，似乎不太明智。

⁴² http://code.google.com/p/leveldb/source/browse/trunk/util/env_posix.cc#35

⁴³ http://code.google.com/p/leveldb/source/browse/trunk/util/env_chromium.cc#176

Kyoto Cabinet

Kyoto Cabinet 也是一个 key-value db，是前几年流行的 Tokyo Cabinet 的升级版。它采用了与 leveldb 不同的文件抽象。KC 定义了一个 File class，同时包含了读写操作，这是一个 fat interface。⁴⁴ 在具体实现方面，它没有使用虚函数，而是采用 `#ifdef` 来区分不同的平台⁴⁵，等于把两份独立的代码写到了同一个文件中。

相比之下，Google leveldb 的做法更高明一些。

小结

在 C++ 项目中，自己写个 File class，把项目用到的文件 IO 功能简单封装一下（以 RAII 手法封装 FILE* 或者 file descriptor 都可以，视情况而定），通常就能满足需要。记得把拷贝构造和赋值操作符禁用，在析构函数里释放资源，避免泄露内部的 handle，这样就能自动避免很多 C 语言文件操作的常见错误。

如果要用 stream 方式做 logging，可以抛开繁重的 iostream，自己写一个简单的 LogStream，重载几个 operator<< 操作符，用起来一样方便；而且可以用 stack buffer，轻松做到线程安全与高效。见第 5 章。

11.7 值语义与数据抽象

本文是 §11.6 “iostream 的用途与局限”的后续，在 §11.6.3 “iostream 与标准库其他组件的交互”中，我简单地提到了 iostream 对象和 C++ 标准库中的其他对象（主要是容器和 string）具有不同的语义，主要体现在 iostream 不能拷贝或赋值。下面具体谈一谈我对这个问题的理解。

本文的“对象”定义较为宽泛：a region of memory that has a type，在这个定义下，int、double、bool 变量都是对象。

11.7.1 什么是值语义

值语义（value semantics）指的是对象的拷贝与原对象无关⁴⁶，就像拷贝 int 一样。C++ 的内置类型（bool/int/double/char）都是值语义，标准库里的 `complex<>`、

⁴⁴ http://fallabs.com/kyotocabinet/api/classkyotocabinet_1_1File.html

⁴⁵ <http://code.google.com/p/read-taobao-code/source/browse/trunk/tair/src/storage/kdb/kyotocabinet/kcfile.cc>

⁴⁶ http://www.boost.org/doc/libs/1_51_0/doc/html/any/reference.html

`pair<>`、`vector<>`、`map<>`、`string` 等等类型也都是值语义，拷贝之后就与原对象脱离关系。Java 语言的 `primitive types` 也是值语义。

与值语义对应的是“对象语义 (object semantics)”，或者叫做引用语义 (reference semantics)，由于“引用”一词在 C++ 里有特殊含义，所以我在本文中使用“对象语义”这个术语。对象语义指的是面向对象意义下的对象，对象拷贝是禁止的。例如 `muduo` 里的 `Thread` 是对象语义，拷贝 `Thread` 是无意义的，也是被禁止的：因为 `Thread` 代表线程，拷贝一个 `Thread` 对象并不能让系统增加一个一模一样的线程。

同样的道理，拷贝一个 `Employee` 对象是没有意义的，一个雇员不会变成两个雇员，他也不会领两份薪水。拷贝 `TcpConnection` 对象也没有意义，系统中只有一个 TCP 连接，拷贝 `TcpConnection` 对象不会让我们拥有两个连接。`Printer` 也是不能拷贝的，系统只连接了一个打印机，拷贝 `Printer` 并不能凭空增加打印机。凡此总总，面向对象意义下的“对象”是 `non-copyable`。

Java 中的 `class` 对象都是对象语义/引用语义。

```
ArrayList<Integer> a = new ArrayList<Integer>();  
ArrayList<Integer> b = a;
```

那么 `a` 和 `b` 指向的是同一个 `ArrayList` 对象，修改 `a` 同时也会影响 `b`。

值语义与 `immutable` 无关。Java 有 `value object` 一说，按 (PoEAA 486) 的定义，它实际上是 `immutable object`，例如 `String`、`Integer`、`BigInteger`、`joda.time.Date-Time` 等等（因为 Java 没有办法实现真正的值语义 `class`，只好用 `immutable object` 来模拟）。尽管 `immutable object` 有其自身的用处，但不是本文的主题。`muduo` 中的 `Date`、`Timestamp` 也都是 `immutable` 的。

C++ 中的值语义对象也可以是 `mutable`，比如 `complex<>`、`pair<>`、`vector<>`、`map<>`、`string` 都是可以修改的。`muduo` 的 `InetAddress` 和 `Buffer` 都具有值语义，它们都是可以修改的。

值语义的对象不一定是 POD，例如 `string` 就不是 POD，但它是值语义的。

值语义的对象不一定小，例如 `vector<int>` 的元素可多可少，但它始终是值语义的。当然，很多值语义的对象都是小的，例如 `complex<>`、`muduo::Date`、`muduo::Timestamp`。

11.7.2 值语义与生命期

值语义的一个巨大好处是生命期管理很简单，就跟 `int` 一样——你不需要操心 `int` 的生命期。值语义的对象要么是 `stack object`，要么直接作为其他 `object` 的成员，因

此我们不用担心它的生命期（一个函数使用自己 stack 上的对象，一个成员函数使用自己的数据成员对象）。相反，对象语义的 object 由于不能拷贝，因此我们只能通过指针或引用来使用它。

一旦使用指针和引用来操作对象，那么就要担心所指的对象是否已被释放，这一度是 C++ 程序 bug 的一大来源。此外，由于 C++ 只能通过指针或引用来获得多态性，那么在 C++ 里从事基于继承和多态的面向对象编程有其本质的困难——对象生命期管理（资源管理）。

考虑一个简单的对象建模——家长与子女：a Parent has a Child, a Child knows its Parent。在 Java 中很好写，不用担心内存泄漏，也不用担心空悬指针：

```
public class Parent
{
    private Child myChild;
}
```

Java code

```
public class Child
{
    private Parent myParent;
}
```

Java code

只要正确初始化 myChild 和 myParent，那么 Java 程序员就不用担心出现访问错误。一个 handle 是否有效，只需要判断其是否 non null。

在 C++ 中就要为资源管理费一番脑筋：Parent 和 Child 都代表的是真人，肯定是不能拷贝的，因此具有对象语义。Parent 是直接持有 Child 吗？抑或 Parent 和 Child 通过指针互指？Child 的生命期由 Parent 控制吗？如果还有 ParentClub 和 School 两个 class，分别代表家长俱乐部和学校：ParentClub has many Parent(s)，School has many Child(ren)，那么如何保证它们始终持有有效的 Parent 对象和 Child 对象？何时才能安全地释放 Parent 和 Child？

直接但是易错的写法：

```
class Child;

class Parent : boost::noncopyable
{
    Child* myChild;
};
```

C++ code

```
class Child : boost::noncopyable
{
    Parent* myParent;
};
```

C++ code

如果直接使用指针作为成员，那么如何确保指针的有效性？如何防止出现空悬指针？Child 和 Parent 由谁负责释放？在释放某个 Parent 对象的时候，如何确保程序中没有指向它的指针？那么释放某个 Child 对象的时候呢？

这一系列问题一度是 C++ 面向对象编程头疼的问题，不过现在有了 smart pointer，我们可以借助 smart pointer 把对象语义转换为值语义⁴⁷，从而轻松解决对象生命期问题：让 Parent 持有 Child 的 smart pointer，同时让 Child 持有 Parent 的 smart pointer，这样始终引用对方的时候就不用担心出现空悬指针。当然，其中一个 smart pointer 应该是 weak reference，否则会出现循环引用，导致内存泄漏。到底哪一个 is weak reference，则取决于具体应用场景。

如果 Parent 拥有 Child，Child 的生命期由其 Parent 控制，Child 的生命期小于 Parent，那么代码就比较简单：

```
class Parent;

class Child : boost::noncopyable
{
public:
    explicit Child(Parent* myParent_)
        : myParent(myParent_)
    { }

private:
    Parent* myParent;
};

class Parent : boost::noncopyable
{
public:
    Parent()
        : myChild(new Child(this))
    { }

private:
    boost::scoped_ptr<Child> myChild;
};
```

在上面这个设计中，Child 的指针不能泄露给外界，否则仍然有可能出现空悬指针。

⁴⁷ 即像持有 int 一样持有对象（的智能指针），其实智能指针本身既不是值语义也不是对象语义。

如果 Parent 与 Child 的生命期相互独立，就要麻烦一些：

```
class Parent;
typedef boost::shared_ptr<Parent> ParentPtr;

class Child : boost::noncopyable
{
public:
    explicit Child(const ParentPtr& myParent_)
        : myParent(myParent_)
    { }

private:
    boost::weak_ptr<Parent> myParent;
};
typedef boost::shared_ptr<Child> ChildPtr;

class Parent : public boost::enable_shared_from_this<Parent>,
               private boost::noncopyable
{
public:
    Parent()
    { }

    void addChild()
    {
        myChild.reset(new Child(shared_from_this()));
    }

private:
    ChildPtr myChild;
};

int main()
{
    ParentPtr p(new Parent);
    p->addChild();
}
```

上面这个 shared_ptr + weak_ptr 的做法似乎有点小题大做。

考虑一个稍微复杂一点的对象模型：“a Child has parents: mom and dad; a Parent has one or more Child(ren); a Parent knows his/her spouse.” 这个对象模型用 Java 表述一点都不复杂，垃圾收集会帮我们搞定对象生命周期。

```
public class Parent
{
    private Parent mySpouse;
    private ArrayList<Child> myChildren;
}
```

Java code

```
public class Child
{
    private Parent myMom;
    private Parent myDad;
}
```

Java code

如果用 C++ 来实现，如何才能避免出现空悬指针，同时避免出现内存泄漏呢？借助 `shared_ptr` 把裸指针转换为值语义，我们就不用担心这两个问题了：

C++ code

```
class Parent;
typedef boost::shared_ptr<Parent> ParentPtr;

class Child : boost::noncopyable
{
public:
    explicit Child(const ParentPtr& myMom_,
                  const ParentPtr& myDad_)
        : myMom(myMom_),
          myDad(myDad_)
    {
    }

private:
    boost::weak_ptr<Parent> myMom;
    boost::weak_ptr<Parent> myDad;
};
typedef boost::shared_ptr<Child> ChildPtr;

class Parent : boost::noncopyable
{
public:
    Parent()
    {
    }

    void setSpouse(const ParentPtr& spouse)
    {
        mySpouse = spouse;
    }

    void addChild(const ChildPtr& child)
    {
        myChildren.push_back(child);
    }

private:
    boost::weak_ptr<Parent> mySpouse;
    std::vector<ChildPtr> myChildren;
};
```



```
int main()
{
    ParentPtr mom(new Parent);
    ParentPtr dad(new Parent);
    mom->setSpouse(dad);
    dad->setSpouse(mom);
    {
        ChildPtr child(new Child(mom, dad));
        mom->addChild(child);
        dad->addChild(child);
    }
    {
        ChildPtr child(new Child(mom, dad));
        mom->addChild(child);
        dad->addChild(child);
    }
}
```

C++ code

如果不使用 smart pointer，用 C++ 做面向对象编程将会困难重重。

11.7.3 值语义与标准库

C++ 要求凡是能放入标准容器的类型必须具有值语义。准确地说：type 必须是 *SGIAssignable* concept 的 model。但是，由于 C++ 编译器会为 class 默认提供 copy constructor 和 assignment operator，因此除非明确禁止，否则 class 总是可以作为标准库的元素类型——尽管程序可以编译通过，但是隐藏了资源管理方面的 bug。

因此，在写一个 C++ class 的时候，让它默认继承 `boost::noncopyable`，几乎总是正确的。

在现代 C++ 中，一般不需要自己编写 copy constructor 或 assignment operator，因为只要每个数据成员都具有值语义的话，编译器自动生成的 member-wise copying & assigning 就能正常工作；如果以 smart ptr 为成员来持有其他对象，那么就能自动启用或禁用 copying & assigning。例外：编写 HashMap 这类底层库时还是需要自己实现 copy control。

11.7.4 值语义与 C++ 语言

C++ 的 class 本质上是值语义的，这才会出现 object slicing 这种语言独有的问题，也才会需要程序员注意 pass-by-value 和 pass-by-const-reference 的取舍。在其他面向对象编程语言中，这都不需要费脑筋。

值语义是 C++ 语言三大约束之一，C++ 的设计初衷是让用户定义的类型（class）能像内置类型（int）一样工作，具有同等的地位。为此 C++ 做了以下设计（妥协）：

- class 的 layout 与 C struct 一样，没有额外的开销。定义一个“只包含一个 int 成员的 class”的对象开销和定义一个 int 一样。
 - 甚至 class data member 都默认是 uninitialized，因为函数局部的 int 也是如此。
 - class 可以在 stack 上创建，也可以在 heap 上创建。因为 int 可以是 stack variable。
 - class 的数组就是一个个 class 对象挨着，没有额外的 indirection。因为 int 数组就是这样的。因此派生类数组的指针不能安全转换为基类指针。
 - 编译器会为 class 默认生成 copy constructor 和 assignment operator。其他语言没有 copy constructor 一说，也不允许重载 assignment operator。C++ 的对象默认是可以拷贝的，这是一个尴尬的特性。
 - 当 class type 传入函数时，默认是 make a copy（除非参数声明为 reference）。因为把 int 传入函数时是 make a copy。
- C++ 的“函数调用”比其他语言复杂之处在于参数传递和返回值传递。C、Java 等语言都是传值，简单地复制几个字节的内存就行了。但是 C++ 对象是值语义，如果以 pass-by-value 方式把对象传入函数，会涉及拷贝构造。代码里看到一句简单的函数调用，实际背后发生的可能是一长串对象构造操作，因此减少无谓的临时对象是 C++ 代码优化的关键之一。
- 当函数返回一个 class type 时，只能通过 make a copy（C++ 不得不定义 RVO 来解决性能问题）。因为函数返回 int 时是 make a copy。
 - 以 class type 为成员时，数据成员是嵌入的。例如 `pair<complex<double>, size_t>` 的 layout 就是 `complex<double>` 挨着 `size_t`。

这些设计带来了性能上的好处，原因是 memory locality。比方说我们在 C++ 里定义 `complex<double>` class，`array of complex<double>`，`vector<complex<double> >`，它们的 layout 如图 11-8 所示。（re 和 im 分别是复数的实部和虚部。）

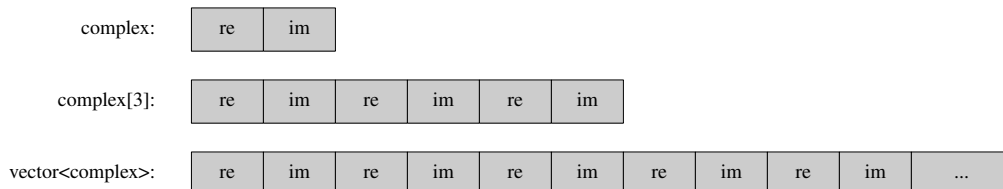


图 11-8

而如果我们在 Java 里干同样的事情, layout 大不一样, memory locality 也差很多 (见图 11-9)。⁴⁸

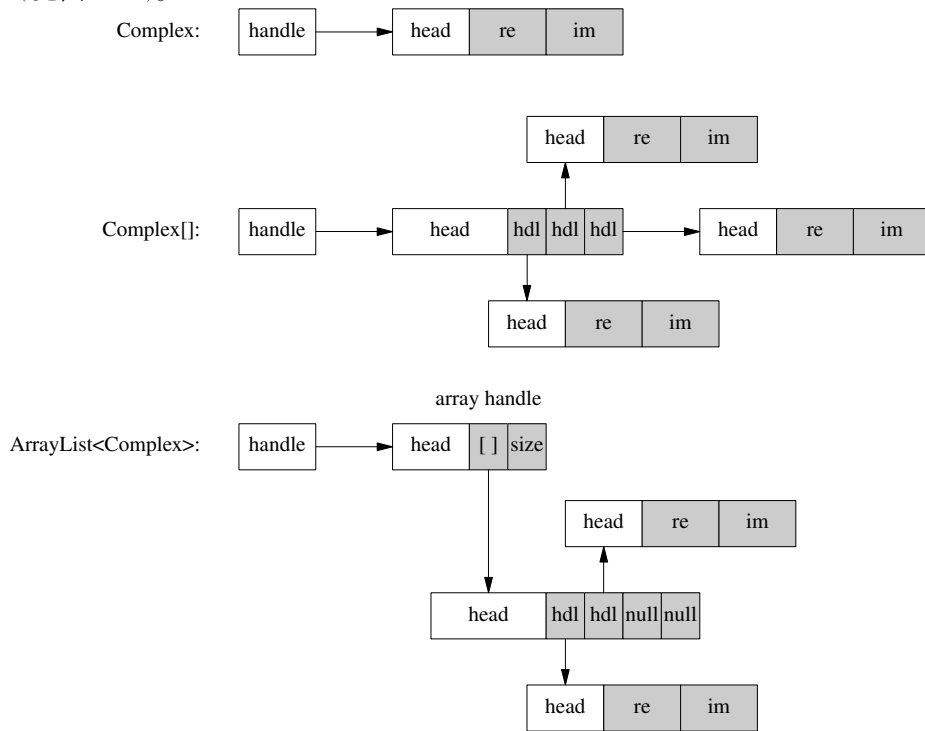


图 11-9

在 Java 中每个 object 都有 head, 在常见的 JVM 中至少有两个 word 的开销。对比 Java 和 C++, 可见 C++ 的对象模型要紧凑得多。

11.7.5 什么是数据抽象

本节谈一谈与值语义紧密相关的数据抽象 (data abstraction), 解释为什么它是与面向对象并列的一种编程范式, 为什么支持面向对象的编程语言不一定支持数据抽象。C++ 在最初的时候以 data abstraction 为卖点, 不过随着时间的流逝, 现在似乎很多人只知 Object-Oriented, 不知 data abstraction 了。C++ 的强大之处在于“抽象”不以性能损失为代价, 本节我们将看到具体例子。

数据抽象 (data abstraction) 是与面向对象 (object-oriented) 并列的一种编程范式 (programming paradigm)。说“数据抽象”或许显得陌生, 它的另外一个名字

⁴⁸ 图 11-9 中的 handle 是 Java 的 reference, 为了避免与 C++ 引用混淆, 这里换个写法。

“抽象数据类型 (abstract data type, ADT)”想必如雷贯耳。

“支持数据抽象”一直是 C++ 语言的设计目标, Bjarne Stroustrup 在他的《The C++ Programming Language (第 2 版)》(1991 年出版)中写道:

The C++ programming language is designed to

- be a better C
- support data abstraction
- support object-oriented programming

这本书的第 3 版(1997 年出版)增加了一条:

C++ is a general-purpose programming language with a bias towards systems programming that

- is a better C,
- supports data abstraction,
- supports object-oriented programming, and
- supports generic programming.

在 C++ 的早期文献⁴⁹中有一篇 Bjarne Stroustrup 于 1984 年写的《Data Abstraction in C++》⁵⁰。在这个页面还能找到 Bjarne 写的关于 C++ 操作符重载和复数运算的文章, 作为数据抽象的详解与范例。可见 C++ 早期是以数据抽象为卖点的, 支持数据抽象是 C++ 相对于 C 的一大优势。

作为语言的设计者, Bjarne 把数据抽象作为 C++ 的四个子语言之一。这个观点不是被普遍接受的, 比如作为语言的使用者, Scott Meyers 在 [EC3] 中把 C++ 分为四个子语言: C、Object-Oriented C++、Template C++、STL。在 Scott Meyers 的分类法中, 就没有出现数据抽象, 而是归入了 Object-Oriented C++。

那么到底什么是数据抽象? 简单地说, 数据抽象是用来描述(抽象)数据结构的。数据抽象就是 ADT。一个 ADT 主要表现为它支持的一些操作, 比方说 `stack::push()`、`stack::pop()`, 这些操作应该具有明确的时间和空间复杂度。另外, 一个 ADT 可以隐藏其实现细节, 例如 `stack` 既可以用动态数组实现, 又可以用链表实现。

按照这个定义, 数据抽象和基于对象(object-based)很像, 那么它们的区别在哪里? 语义不同。ADT 通常是值语义, 而 object-based 是对象语义。(这两种语义的定义见 §11.7.1 “什么是值语义”)。ADT class 是可以拷贝的, 拷贝之后的 instance 与原 instance 脱离关系。

⁴⁹ http://www.softwarepreservation.org/projects/c_plus_plus/index.html#cfrent

⁵⁰ http://www.softwarepreservation.org/projects/c_plus_plus/cfrent/release_e/doc/DataAbstraction.pdf

比方说

```
stack<int> a;  
a.push(10);  
stack<int> b = a;  
b.pop();
```

这时候 a 里仍然有元素 10。

C++ 标准库中的数据抽象

C++ 标准库里 `complex<>`、`pair<>`、`vector<>`、`list<>`、`map<>`、`set<>`、`string`、`stack`、`queue` 都是数据抽象的例子。`vector` 是动态数组，它的主要操作有 `size()`、`begin()`、`end()`、`push_back()` 等等，这些操作不仅含义清晰，而且计算复杂度都是常数。类似地，`list` 是链表，`map` 是有序关联数组，`set` 是有序集合、`stack` 是 FILO 栈、`queue` 是 FIFO 队列。“动态数组”、“链表”、“有序集合”、“关联数组”、“栈”、“队列”都是定义明确（操作、复杂度）的抽象数据类型。

数据抽象与面向对象的区别

本文把 `data abstraction`、`object-based`、`object-oriented` 视为三个编程范式。这种细致的分类或许有助于理解区分它们之间的差别。

庸俗地讲，面向对象（`object-oriented`）有三大特征：封装、继承、多态。而基于对象（`object-based`）则只有封装，没有继承和多态，即只有具体类，没有抽象接口。它们两个都是对象语义。

面向对象真正核心的思想是消息传递（`messaging`），“封装继承多态”只是表象。关于这一点，孟岩⁵¹和王益⁵²都有精彩的论述，笔者不再赘言。

数据抽象与它们两个的界限在于“语义”，数据抽象不是对象语义，而是值语义。比方说 `muduo` 里的 `TcpConnection` 和 `Buffer` 都是具体类，但前者是基于对象的（`object-based`），而后者是数据抽象。

类似地，`muduo::Date`、`muduo::Timestamp` 都是数据抽象。尽管这两个 `class` 简单到只有一个 `int/long` 数据成员，但是它们各自定义了一套操作（`operation`），并隐藏了内部数据，从而让它从 `data aggregation` 变成了 `data abstraction`。

数据抽象是针对“数据”的，这意味着 `ADT class` 应该可以拷贝，只要把数据复制一份就行了。如果一个 `class` 代表了其他资源（文件、员工、打印机、账号），那么

⁵¹ <http://blog.csdn.net/myan/article/details/5928531>

⁵² <http://cxwangyi.wordpress.com/2011/06/19/杂谈现代高级编程语言/>

它通常就是 object-based 或 object-oriented，而不是数据抽象。

ADT class 可以作为 Object-based/object-oriented class 的成员，但反过来不成立，因为这样一来 ADS class 的拷贝就失去意义了。

11.7.6 数据抽象所需的语言设施

不是每个语言都支持数据抽象，下面简要列出“数据抽象”所需的语言设施。

支持数据聚合 数据聚合即 data aggregation，或者叫 value aggregates。即定义 C-style struct，把有关数据放到同一个 struct 里。FORTRAN 77 没有这个能力，FORTRAN 77 无法实现 ADT。这种数据聚合 struct 是 ADT 的基础，struct List、struct HashTable 等能把链表和哈希表结构的数据放到一起，而不是用几个零散的变量来表示它。

全局函数与重载 例如我定义了 complex，那么我可以同时定义 complex sin(const complex& x) 和 complex exp(const complex& x) 等等全局函数来实现复数的三角函数和指数运算。sin() 和 exp() 不是 complex 的成员，而是全局函数 double sin(double) 和 double exp(double) 的重载。这样能让 double a = sin(b); 和 complex a = sin(b); 具有相同的代码形式，而不必写成 complex a = b.sin();。

C 语言可以定义全局函数，但是不能与已有的函数重名，也就没有重载。Java 没有全局函数，而且 Math class 是封闭的，并不能往其中添加 sin(Complex)。

成员函数与 private 数据 数据也可以声明为 private，防止外界意外修改。不是每个 ADT 都适合把数据声明为 private，例如 complex、Point、pair<> 这样的 ADT 使用 public data 更加合理。

要能够在 struct 里定义操作，而不是只能用全局函数来操作 struct。比方说 vector 有 push_back() 操作，push_back 是 vector 的一部分，它必须直接修改 vector 的 private data members，因此无法定义为全局函数。

这两点其实就是定义 class，现在的语言都能直接支持，C 语言除外。

拷贝控制 (copy control) copy control 是拷贝 stack a; stack b = a; 和赋值 stack b; b = a; 的合称。

当拷贝一个 ADT 时会发生什么？比方说拷贝一个 stack，是不是应该把它的每个元素按值拷贝到新 stack？

如果语言支持显示控制对象的生命期（比方说 C++ 的确定性析构），而 ADT 用到了动态分配的内存，那么 copy control 更为重要，可防止访问已经失效的对象。

由于 C++ class 是值语义，copy control 是实现深拷贝的必要手段，而且 ADT 用到的资源只涉及动态分配的内存，所以深拷贝是可行的。相反，object-based 编程风格中的 class 往往代表某样真实的事物（Employee、Account、File 等等），深拷贝无意义。

C 语言没有 copy control，也没有办法防止拷贝，一切要靠程序员自己小心在意。FILE* 可以随意拷贝，但是只要关闭其中一个 copy，其他 copy 也都失效了，跟空悬指针一般。整个 C 语言对待资源（malloc() 得到的内存，open() 打开的文件，socket() 打开的连接）都是这样的，用整数或指针来代表（即“句柄”）。而整数和指针类型的“句柄”是可以随意拷贝的，很容易就造成重复释放、遗漏释放、使用已经释放的资源等等常见错误。这方面 C++ 是一个显著的进步，我认为 boost::noncopyable 是 Boost 里最值得推广的库。

操作符重载 如果要写动态数组，我们希望能像使用内置数组一样使用它，比如支持下标操作。C++ 可以重载 operator[] 来做到这一点。

如果要写复数，我们希望能像使用内置的 double 一样使用它，比如支持加减乘除。C++ 可以重载 operator+ 等操作符来做到这一点。

如果要写日期与时间，我们希望它能直接用大于或小于号来比较先后，用 == 来判断是否相等。C++ 可以重载 operator< 等操作符来做到这一点。

这要求语言能重载成员与全局操作符。操作符重载是 C++ 与生俱来的特性，1984 年的 CFront E 就支持操作符重载，并且提供了一个 complex class，这个 class 与目前标准库的 complex<> 在使用上无区别。

如果没有操作符重载，那么用户定义的 ADT 与内置类型用起来就不一样了（想想有的语言要区分 == 和 equals，代码写起来实在很累赘）。Java 里有 BigInteger，但是 BigInteger 用起来和普通 int/long 大不相同：

```
public static BigInteger mean(BigInteger x, BigInteger y) {
    BigInteger two = BigInteger.valueOf(2);
    return x.add(y).divide(two);
}

public static long mean(long x, long y) {
    return (x + y) / 2;
}
```

Java code

Java code

当然，操作符重载容易被滥用，因为这样显得很“酷”。我认为只在 ADT 表示一个“数值”的时候才适合重载加减乘除，其他情况下用具名函数为好，因此 `muduo::Timestamp` 只重载了关系操作符，没有重载加减操作符。另外一个理由见 §12.6 “采用有利于版本管理的代码格式”。

效率无损 “抽象”不代表低效。在 C++ 中，提高抽象的层次并不会降低效率。不然的话，人们宁可在低层次上编程，而不愿使用更便利的抽象，数据抽象也就失去了市场。后面我们将看到一个具体的例子。

模板与泛型 如果我写了一个 `IntVector`，那么我不想为 `double` 和 `string` 再实现一遍同样的代码。我应该把 `vector` 写成 `template`，然后用不同的类型来具现化它，从而得到 `vector<int>`、`vector<double>`、`vector<complex>`、`vector<string>` 等具体类型。

不是每个 ADT 都需要这种泛型能力，一个 `Date` class 就没必要让用户指定该用哪种类型的整数，`int32_t` 足够了。

根据上面的要求，不是每个面向对象语言都能原生支持数据抽象，也说明数据抽象不是面向对象的子集。

11.7.7 数据抽象的例子

下面我们看看数值模拟 *N*-body 问题的两个程序，前一个是用 C 语言，后一个是用 C++ 语言。这个例子来自编程语言的性能对比网站⁵³。两个程序使用的算法相同。

C 语言版，完整代码见 `recipes/puzzle/file_nbody.c`，下面是核心代码。`struct planet` 保存行星位置、速度、质量，位置和速度各有三个分量。程序模拟几大行星在三维空间中受引力支配的运动。

其中最核心的算法是 `advance()` 函数实现的数值积分，它根据各个星球之间的距离和引力，算出加速度，再修正速度，然后更新星球的位置。这个 naïve 算法的复杂度是 $O(N^2)$ 。

```
struct planet
{
    double x, y, z;
    double vx, vy, vz;
    double mass;
};
```

C code

⁵³ <http://shootout.alioth.debian.org/gp4/benchmark.php?test=nbody&lang=all>


```

void advance(int nbodies, struct planet *bodies, double dt)
{
    for (int i = 0; i < nbodies; i++)
    {
        struct planet *p1 = &(bodies[i]);
        for (int j = i + 1; j < nbodies; j++)
        {
            struct planet *p2 = &(bodies[j]);
            double dx = p1->x - p2->x;
            double dy = p1->y - p2->y;
            double dz = p1->z - p2->z;
            double distance_squared = dx * dx + dy * dy + dz * dz;
            double distance = sqrt(distance_squared);
            double mag = dt / (distance * distance_squared);
            p1->vx -= dx * p2->mass * mag;
            p1->vy -= dy * p2->mass * mag;
            p1->vz -= dz * p2->mass * mag;
            p2->vx += dx * p1->mass * mag;
            p2->vy += dy * p1->mass * mag;
            p2->vz += dz * p1->mass * mag;
        }
    }
    for (int i = 0; i < nbodies; i++)
    {
        struct planet * p = &(bodies[i]);
        p->x += dt * p->vx;
        p->y += dt * p->vy;
        p->z += dt * p->vz;
    }
}

```

C code

C++ 数据抽象版，完整代码见 `recipes/puzzle/file_nbody.cc`，下面是其代码骨架。首先定义 `Vector3` 这个抽象，代表三维向量，它既可以是位置，又可以是速度。本处略去了 `Vector3` 的操作符重载（`Vector3` 支持常见的向量加减乘除运算）。然后定义 `Planet` 这个抽象，代表一个行星，它有两个 `Vector3` 成员：位置和速度。需要说明的是，按照语义，`Vector3` 是数据抽象，而 `Planet` 是 object-based。

```

struct Vector3
{
    Vector3(double x, double y, double z)
        : x(x), y(y), z(z)
    { }

    double x;
    double y;
    double z;
};

```

C++ code

```

struct Planet
{
    Planet(const Vector3& position, const Vector3& velocity, double mass)
        : position(position), velocity(velocity), mass(mass)
    { }

    Vector3 position;
    Vector3 velocity;
    const double mass;
};

```

C++ code

相同功能的 `advance()` 代码则简短得多，而且更容易验证其正确性。（设想假如把 C 语言版的 `advance()` 中的 `vx`、`vy`、`vz`、`dx`、`dy`、`dz` 写错位了，这种错误较难发现。）

```

void advance(int nbodies, Planet* bodies, double delta_time)
{
    for (Planet* p1 = bodies; p1 != bodies + nbodies; ++p1)
    {
        for (Planet* p2 = p1 + 1; p2 != bodies + nbodies; ++p2)
        {
            Vector3 difference = p1->position - p2->position;
            double distance_squared = magnitude_squared(difference);
            double distance = std::sqrt(distance_squared);
            double magnitude = delta_time / (distance * distance_squared);
            p1->velocity -= difference * p2->mass * magnitude;
            p2->velocity += difference * p1->mass * magnitude;
        }
    }
    for (Planet* p = bodies; p != bodies + nbodies; ++p)
    {
        p->position += delta_time * p->velocity;
    }
}

```

C++ code

C++ code

尽管 C++ 使用了更高层的抽象 `Vector3`，但它的性能和 C 语言一样快。看看 `memory layout` 就会明白。

C `struct` 的成员是连续存储的，`struct` 数组也是连续的，如图 11-10 所示。

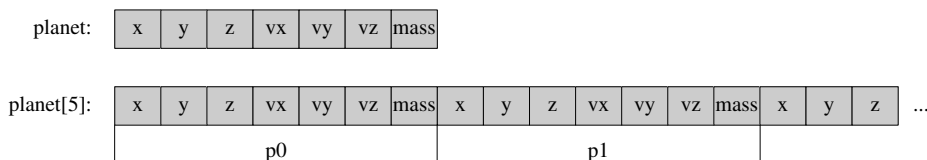


图 11-10

尽管 C++ 定义了 `Vector3` 这个抽象，但它的内存布局并没有改变（见图 11-11），C++ `Planet` 的布局和 C `planet` 一模一样，`Planet[]` 的布局也和 C 数组一样。

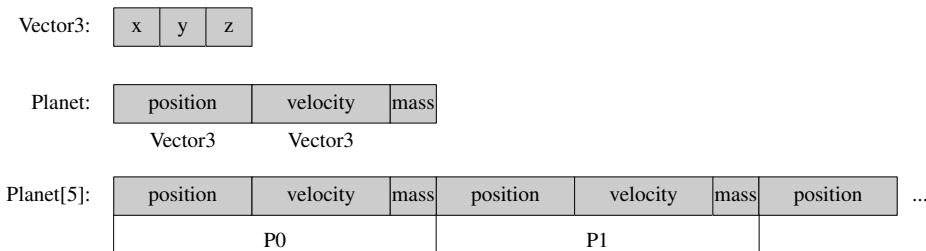


图 11-11

另一方面，C++ 的 `inline` 函数在这里也起了巨大作用，我们可以放心地调用 `Vector3::operator+=( )` 等操作符，编译器会生成和 C 一样高效的代码。

不是每个编程语言都能做到在提升抽象的时候不影响性能，来看看 Java 的内存布局。如果我们用 `class Vector3`、`class Planet`、`Planet[]` 的方式写一个 Java 版的 *N-body* 程序，内存布局将会是如图 11-12 所示的样子。这样大大降低了 *memory locality*，有兴趣的读者可以对比 Java 和 C++ 的实现效率。

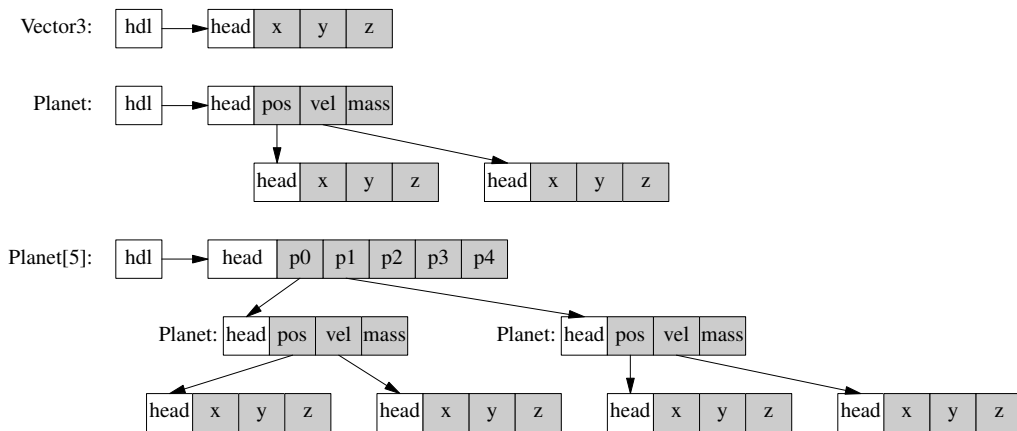


图 11-12

注：这里的 *N-body* 算法只为比较语言之间的性能与编程的便利性；真正科研中用到的 *N-body* 算法会使用更高级和底层的优化，复杂度是 $O(N \log N)$ ，在大规模模拟时其运行速度也比本 *naïve* 算法快得多。

更多的例子

- Date 与 Timestamp, 这两个 class 的“数据”都是整数, 各定义了一套操作, 用于表达日期与时间这两个概念。
- BigInteger, 它本身就是一个“数”。如果用 C++ 实现 BigInteger, 那么阶乘函数写出来十分自然。下面第二个函数是 Java 语言的版本。

```
// C++ code
BigInteger factorial(int n)
{
    BigInteger result(1);
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}

// Java code
public static BigInteger factorial(int n) {
    BigInteger result = BigInteger.ONE;
    for (int i = 1; i <= n; ++i) {
        result = result.multiply(BigInteger.valueOf(i));
    }
    return result;
}
```

高精度运算库 gmp 有一套高质量的 C++ 封装⁵⁴

- 图形学中的三维齐次坐标 Vector4 和对应的 4×4 变换矩阵 Matrix4⁵⁵。
- 金融领域中经常成对出现的“买入价/卖出价”, 可以封装为 BidOffer struct, 这个 struct 的成员可以有 mid() (中间价)、spread() (买卖差价)、加减操作符等等。

小结

数据抽象是 C++ 的重要抽象手段, 适合封装“数据”, 它的语义简单, 容易使用。数据抽象能简化代码书写, 减少偶然错误。

在新写一个 class 的时候, 先想清楚它是值语义还是对象语义。一般来说, 一个项目里只有少量的 class 是值语义, 比如一些 snapshot 的数据, 而大多数 class 都是对象语义。

⁵⁴ http://gmplib.org/manual/C_002b_002b-Interface-General.html#C_002b_002b-Interface-General

⁵⁵ http://www.ogre3d.org/docs/api/html/classOgre_1_1Matrix4.html

如果是对象语义的 `class`，那么应该立刻继承 `boost::noncopyable`，防止编译器自动生成的拷贝构造函数和赋值操作符在无意中破坏程序行为⁵⁶。（比如防止有人误将对象语义的 `class` 放入标准库容器。）

⁵⁶ 我认为 C++ 最好修改语言规则，一旦 `class` 定义了析构函数，那么编译器就不应该自动生成拷贝构造函数和赋值操作符。似乎 C++11 已经做了类似的规定？

第 12 章

C++ 经验谈

我对 C++ 的基本态度是“练从难处练，用从易处用”，因此本章有几节“负面”的内容。我坚信软件开发一定要时刻注意减少不必要的复杂度，一些花团锦簇的招式玩不好反倒会伤到自己。作为应用程序的开发者，对技术的运用要明智，不要为了解决难度系数为 10 的问题而去强攻难度系数为 100 的问题，这就本末倒置了。

12.1 用异或来交换变量是错误的

翻转一个字符串，例如把 "12345" 变成 "54321"，这是一个最简单不过的编码任务，即便是 C 语言初学者也能毫不费力地写出类似如下的代码：

```
// 版本一，用中间变量交换两个数，好代码
void reverse_by_swap(char* str, int n)
{
    char* begin = str;
    char* end = str + n - 1;

    while (begin < end)
    {
        char tmp = *begin;
        *begin = *end;
        *end = tmp;
        ++begin;
        --end;
    }
}
```

Version 1

Version 1

上面这段代码清晰，直白，没有任何高深的技巧。不知从什么时候开始，有人“发明”了不使用临时变量交换两个数的办法，用关键词“不用临时变量 交换 两个数”在 Google 上能搜到很多文章。下面是一个典型的实现：

Version 2

```
// 版本二，用异或运算交换两个数，烂代码
void reverse_by_xor(char* str, int n)
{
    // WARNING: BAD code
    char* begin = str;
    char* end = str + n - 1;

    while (begin < end)
    {
        *begin ^= *end;
        *end ^= *begin;
        *begin ^= *end;
        ++begin;
        --end;
    }
}
```

Version 2

受一些过时的教科书的误导，有人认为程序里少用一个变量，节省一个字节的空
间，会让程序运行得更快。这是不对的，至少在这里不成立：

1. 这个所谓的“技巧”在现代的机器上只会更慢（我甚至怀疑它从来就不可能比
原始办法快）。原始办法是两次内存读和写，这个“技巧”是六读三写加三次异
或（或许编译器可以优化成两读三写加三次异或）。
2. 同样也不能节省内存，因为中间变量 `tmp` 通常会 是寄存器（稍后有汇编代码供
分析）。就算它在函数的局部堆栈（`stack`）上，反正栈已经开在那儿了，也没有
进一步的函数调用，根本节约不了一丁点内存。
3. 相反，由于计算步骤较多，会使用更多的指令，编译后的机器码长度会增加。
（这不是什么大问题，短的代码不一定快，后面有另外一个例子。）

这个技巧的意义完全在于应付无聊的面试，所以知道就行，但绝对不能放在产品
代码中。我也想不出问这样的面试题意义何在。

更有甚者，把其中三句：

```
*begin ^= *end;
*end ^= *begin;
*begin ^= *end;
```

写成一句：

```
*begin ^= *end ^= *begin ^= *end; // WRONG
```

这更是大有问题，会导致未定义的行为（**undefined behavior**）¹。在 C/C++ 语言的一条语句中，一个变量的值只允许改变一次。（像 `x = x++` 这种代码都是未定义行为，因为 `x` 有两次写入。²）在 C/C++ 语言里没有哪条规则保证这两种写法是等价的。（致语言律师：我知道，黑话叫序列点³，一个语句可能不止一个序列点，请允许我在这里使用不精确的表述。）

这不是一个值得炫耀的技巧，只会丑化、劣化代码。

C++ 对翻转字符串这个问题有更简单的解法——调用 STL 里的 `std::reverse()` 函数。有人担心调用函数会有开销，这种担心是多余的，现在的编译器会把 `std::reverse()` 这种简单函数自动内联展开，生成出来的优化汇编代码和“版本一”一样快。

```
// 版本三，用 std::reverse 颠倒一个区间，优质代码
void reverse_by_std(char* str, int n)
{
    std::reverse(str, str + n);
}
```

Version 3

Version 3

12.1.1 编译器会分别生成什么代码

注意：查看编译器生成的汇编代码固然是了解程序行为的一个重要手段，但是千万不要认为看到的永远是永恒真理，它只是一时一地的真相。将来换了硬件平台或编译器，情况可能会变化。重要的不是为什么版本一比版本二快，而是如何发现这个事实。不要“猜（*guess*）”，要“测（*benchmark*）”。

以 g++ 版本 4.4.1，编译参数 `-O2 -march=core2`，x86 Linux 系统为例。

版本一 版本一编译得到的汇编代码是：

```
.L3:
    movzbl    (%edx), %ecx
    movzbl    (%eax), %ebx
    movb      %bl, (%edx)
    movb      %cl, (%eax)
    incl      %edx
    decl      %eax
    cmpl      %eax, %edx
    jb         .L3
```

¹ gcc 的作者明说这种写法是 **undefined** 的，见 http://gcc.gnu.org/bugzilla/show_bug.cgi?id=39121。

² http://www.stroustrup.com/bs_faq2.html#evaluation-order

³ GCC 4.x 有一个编译警告选项 `-Wsequence-point` 可以报告这种错误。

我用 C 语言翻译一下：

```
register char bl, cl;
register char* eax;
register char* edx;

L3:
cl = *edx; // 读
bl = *eax; // 读
*edx = bl; // 写
*eax = cl; // 写
++edx;
--eax;
if (edx < eax) goto L3;
```

一共两读两写，临时变量没有使用内存，都在寄存器里完成。考虑指令级并行和 cache 的话，中间六条语句估计能在三四个周期执行完。

版本二

```
.L9:
    movzbl    (%edx), %ecx
    xorb      (%eax), %cl
    movb      %cl, (%eax)
    xorb      (%edx), %cl
    movb      %cl, (%edx)
    decl      %edx
    xorb      %cl, (%eax)
    incl      %eax
    cmpl      %edx, %eax
    jb        .L9
```

C 语言翻译：

```
// 声明与前面一样
cl = *edx;    // 读
cl ^= *eax;   // 读，异或
*eax = cl;    // 写
cl ^= *edx;   // 读，异或
*edx = cl;    // 写
--edx;
*eax ^= cl;   // 读、写，异或
++eax;
if (eax < edx) goto L9;
```

一共六读三写三次异或，多了两条指令。指令多不一定就慢，但是这里异或版实测比临时变量版要慢许多，因为它的每条指令都用到了前面一条指令的计算结果，没法并行执行。

版本三 生成的代码与“版本一”一样快。

```
.L21:
    movzbl    (%eax), %ecx
    movzbl    (%edx), %ebx
    movb      %bl, (%eax)
    movb      %cl, (%edx)
    incl      %eax

.L23:
    decl      %edx
    cmpl      %edx, %eax
    jb        .L21
```

这告诉我们，不要想当然地优化，也不要低估编译器的能力。关于现在的编译器有多聪明，Felix von Leitner 有一个不错的介绍⁴。

Bjarne Stroustrup 说过：“我喜欢优雅和高效的代码。代码逻辑应当直截了当，叫缺陷难以隐藏；尽量减少依赖关系，使之便于维护；以某种全局策略一以贯之地处理全部出错情况；性能调校至接近最优，省得引诱别人实施无原则的优化 (unprincipled optimizations)，搞出一团乱麻。整洁的代码只做好一件事。”⁵

这恐怕就是 Bjarne 提及的没有原则的优化，甚至根本连优化都不是。代码的清晰性是首要的。

12.1.2 为什么短的代码不一定快

§12.3 将会谈到负整数的除法运算，其中引用了一段把整数转为字符串的代码。函数反复计算一个整数除以 10 的商和余数。我原以为编译器会用一条 DIV 除法指令来算，实际生成的代码让我大吃一惊：

```
.L2:
    movl      $1717986919, %eax
    imull     %ebx
    movl      %ebx, %eax
    sarl      $31, %eax
    sarl      $2, %edx
    subl      %eax, %edx
    movl      %edx, %eax
    leal      (%edx,%edx,4), %edx
    addl      %edx, %edx
    subl      %edx, %ebx
    movl      %ebx, %edx
    movl      %eax, %ebx
```

⁴ http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf

⁵ 译文引自韩磊翻译的《代码整洁之道》，笔者对文字略有修改。

```
movzbl  (%edi,%edx), %eax
movb    %al, (%esi)
addl    $1, %esi
testl   %ebx, %ebx
jne     .L2
```

一条 DIV 指令被替换成了十来条指令，编译器不是傻子，必然有原因。这里我不详细解释到底是怎么算的，基本思路是把除法转换为乘法，用倒数来算。其中出现了一个魔数 1717986919，转换成十六进制是 0x66666667，等于 $(2^{33} + 3)/5$ 。

现代处理器的乘法运算和加减法一样快，比除法快一个数量级左右，编译器生成这样的代码是有理由的。十多年前出版的巨著《程序设计实践》[TPoP] 中介绍过如何做 micro benchmarking，方法和结果都值得一读，当然里边的数据恐怕有点过时了。

有本奇书《Hacker's Delight》（中译本《高效程序的奥秘》），展示了大量这种速算技巧。其中第 10 章专门讲整数常量的除法。我不会把其中如天书般的技巧应用到产品代码中，但是我相信现代编译器的作者是知道这些技巧的，他们会合理地使用这些技巧来提高生成代码的质量。现在已经不是那个懂点汇编就能打败 C/C++ 编译器的时代了。

Mark C. Chu-Carroll 有一篇博客《The “C is Efficient” Language Fallacy》⁶ 的观点我非常赞同，即用清晰的代码表达程序员的意图，让编译器容易实施优化。

Making real applications run really fast is something that's done with the help of a compiler. Modern architectures have reached the point where people can't code effectively in assembler anymore — switching the order of two independent instructions can have a dramatic impact on performance in a modern machine, and the constraints that you need to optimize for are just more complicated than people can generally deal with.

So for modern systems, **writing an efficient program is sort of a partnership**. The human needs to carefully choose algorithms — the machine can't possibly do that. And the machine needs to carefully compute instruction ordering, pipeline constraints, memory fetch delays, etc. The two together can build really fast systems. But the two parts aren't independent: **the human needs to express the algorithm in a way that allows the compiler to understand it well enough to be able to really optimize it**.

⁶ http://scienceblogs.com/goodmath/2006/11/the_c_is_efficient_language_fa.php

最后，说说 C++ 模板。假如要编写一个任意进制的转换程序。C 语言的函数声明是：

```
bool convert(char* buf, size_t bufsize, int value, int radix);
```

既然进制是编译期常量，C++ 可以用带非类型模板参数的函数模板来实现，函数的代码与 C 相同。

```
template<int radix>
bool convert(char* buf, size_t bufsize, int value);
```

模板确实会使代码膨胀，但是这样的膨胀有时候是好事情，编译器能针对不同的常数生成快速算法。滥用 C++ 模板当然是错的，适当使用不会有问题。

12.2 不要重载全局 `::operator new()`

本文只考虑 Linux x86 平台，服务端开发（不考虑 Windows 的跨 DLL 内存分配释放问题）。本文假定读者知道 `::operator new()` 和 `::operator delete()` 是干什么的，与通常用的 `new/delete` 表达式有何区别和联系，这方面的知识可参考侯捷先生的文章《池内春秋》[jjhou02]，或者这篇文章：<http://www.relisoft.com/book/tech/9new.html>。

C++ 的内存管理是个老生常谈的话题，我在 §1.7 “插曲：系统地避免各种指针错误”中简单回顾了一些常见的问题以及在现代 C++ 中的解决办法。基本上，按现代 C++ 的手法（RAII）来管理内存，你很难遇到什么内存方面的错误。“没有错误”是基本要求，不代表“足够好”。我们常常会设法优化性能，如果 profiling 表明 hot spot 在内存分配和释放上，重载全局的 `::operator new()` 和 `::operator delete()` 似乎是一个一劳永逸的好办法（以下简称为“重载 `::operator new()`”）。本节试图说明这个办法往往行不通。

12.2.1 内存管理的基本要求

如果只考虑分配和释放，内存管理基本要求是“不重不漏”：既不重复 `delete`，也不漏掉 `delete`。也就是说我们常说的 `new/delete` 要配对，“配对”不仅是个数相等，还隐含了 `new` 和 `delete` 的调用本身要匹配，不要“东家借的东西西家还”。例如：

- 用系统默认的 `malloc()` 分配的内存要交给系统默认的 `free()` 去释放。
- 用系统默认的 `new` 表达式创建的对象要交给系统默认的 `delete` 表达式去析构并释放。

- 用系统默认的 `new[]` 表达式创建的对象要交给系统默认的 `delete[]` 表达式去析构并释放。
- 用系统默认的 `::operator new()` 分配的内存要交给系统默认的 `::operator delete()` 去释放。
- 用 `placement new` 创建的对象要用 `placement delete`（为了表述方便，姑且这么说吧）去析构（其实就是直接调用析构函数）。
- 从某个内存池 A 分配的内存要还给这个内存池。
- 如果定制 `new/delete`，那么要按规矩来。见《Effective C++ 中文版（第 3 版）》[EC3] 第 8 章“定制 `new` 和 `delete`”。

做到以上这些不难，是每个 C++ 开发人员的基本功。不过，如果你想重载全局的 `::operator new()`，事情就麻烦了。

12.2.2 重载 `::operator new()` 的理由

[EC3, 条款 50] 列举了定制 `new/delete` 的几点理由：

- 检测代码中的内存错误；
- 优化性能；
- 获得内存使用的统计数据。

这些都是正当的需求，后面我们将会看到，不重载 `::operator new()` 也能达到同样的目的。

12.2.3 `::operator new()` 的两种重载方式

1. 不改变其签名，无缝直接替换系统原有的版本，例如：

```
#include <new>

void* operator new(size_t size);
void operator delete(void* p);
```

用这种方式的重载，使用方不需要包含任何特殊的头文件，也就是说不需要看见这两个函数声明。“性能优化”通常用这种方式。

2. 增加新的参数，调用时也提供这些额外的参数，例如：

```
// 此函数返回的指针必须能被普通的 ::operator delete(void*) 释放
void* operator new(size_t size, const char* file, int line);
```

```
// 此函数只在构造函数抛异常的情况下才会被调用
void operator delete(void* p, const char* file, int line);
```

然后用的时候是

```
Foo* p = new (__FILE__, __LINE__) Foo; // 这样能跟踪是哪个文件哪一行代码分配的内存
```

我们也可以用宏替换 `new` 来节省打字。用这里的第二种方式重载，使用方需要看到这两个函数声明，也就是说要主动包含你提供的头文件。“检测内存错误”和“统计内存使用情况”通常会用这种方式重载。当然，这不是绝对的。

在学习 C++ 的阶段，每个人都可以写个一两百行的程序来验证教科书上的说法，重载 `::operator new()` 在这样的玩具程序里边不会造成什么麻烦。

不过，我认为在现实的产品开发中，重载 `::operator new()` 乃是下策，我们有更简单、安全的办法来达到以上目标。

12.2.4 现实的开发环境

作为 C++ 应用程序的开发人员，在编写稍具规模的程序时，我们通常会用到一些 `library`。我们可以根据 `library` 的提供方把它们大致分为这么几大类：

1. C 语言的标准库，也包括 Linux 编程环境提供的 `glibc` 系列函数。
2. 第三方的 C 语言库，例如 `OpenSSL`。
3. C++ 语言的标准库，主要是 `STL`。（我想没有人在产品中使用 `iostream` 吧？）
4. 第三方的通用 C++ 库，例如 `Boost.Regex`，或者某款 XML 库。
5. 公司其他团队的人开发的内部基础 C++ 库，比如网络通信和日志等基础设施。
6. 本项目组的同事自己开发的针对本应用的基础库，比如某三维模型的仿射变换模块。

在使用这些 `library` 的时候，不可避免地要在各个 `library` 之间交换数据。比方说 `library A` 的输出作为 `library B` 的输入，而 `library A` 的输出本身常常会用到动态分配的内存（比如 `std::vector<double>`）。

如果所有的 C++ `library` 都用同一套内存分配器（就是系统默认的 `new/delete`），那么内存的释放就很方便，直接交给 `delete` 去释放就行。如果不是这样，那就得时时

刻刻记住“这一块内存是属于哪个分配器的，是系统默认的还是我们定制的，释放的时候不要还错了地方”。

由于 C 语言不像 C++ 一样提供了那么多的定制性，C library 通常都会默认直接用 malloc/free 来分配和释放内存，不存在上面提到的“内存还错地方”问题。或者有的考虑更全面的 C library 会让你注册两个函数，用于其内部分配和释放内存，这就能完全掌控该 library 的内存使用。这种依赖注入的方式在 C++ 里变得花哨而无用，见笔者写的《C++ 标准库中的 allocator 是多余的》⁷。

但是，如果重载了 ::operator new()，事情恐怕就没有这么简单了。

12.2.5 重载 ::operator new() 的困境

首先，重载 ::operator new() 不会给 C 语言的库带来任何麻烦。当然，重载它得到的三点好处也无法让 C 语言的库享受到。以下仅考虑 C++ library 和主程序。

规则 1：绝对不能在 library 里重载 ::operator new()

如果你是某个 library 的作者，你的 library 要提供给别人使用，那么你无权重载全局 ::operator new(size_t)（注意这是前面提到的第一种重载方式），因为这非常具有侵略性：任何用到你的 library 的程序都被迫使用了你重载的 ::operator new()，而别人很可能不愿意这么做。另外，如果有两个 library 都试图重载 ::operator new(size_t)，那么它们会打架，我估计会发生 duplicated symbol link error。（这还算是好的，如果某个实现偷偷盖住了另一个实现，会在运行时发生诡异的现象。）干脆，作为 library 的编写者，大家都不要重载 ::operator new(size_t) 好了。

那么第二种重载方式呢？

首先，::operator new(size_t size, const char* file, int line) 这种方式得到的 void* 指针必须同时能被 ::operator delete(void*) 和 ::operator delete(void* p, const char* file, int line) 这两个函数释放。这时候你需要决定，你的 ::operator new(size_t size, const char* file, int line) 返回的指针是不是兼容系统默认的 ::operator delete(void*)。

如果不兼容（也就是说不能用系统默认的 ::operator delete(void*) 来释放内存），那么你得重载 ::operator delete(void*)，让它的行为与你的 ::operator new(size_t size, const char* file, int line) 匹配。一旦你决定重载 ::operator delete(void*)，

⁷ <http://blog.csdn.net/Solstice/archive/2009/08/02/4401382.aspx>

那么你必须重载 `::operator new(size_t)`，这就回到了规则 1：你无权重载全局 `::operator new(size_t)`。

如果选择兼容系统默认的 `::operator delete(void*)`，那么你在 `::operator new(size_t size, const char* file, int line)` 里能做的事情非常有限，比方说你不能额外动态分配内存来做 **house keeping** 或保存统计数据（无论显式还是隐式），因为系统默认的 `::operator delete(void*)` 不会释放你额外分配的内存。（这里隐式分配内存指的是往 `std::map<>` 这样的容器里添加元素。）看到这里，估计很多人已经晕了，但这还没完。

其次，在 **library** 里重载 `::operator new(size_t size, const char* file, int line)` 还涉及你的重载要不要暴露给 **library** 的使用者（其他 **library** 或主程序）。这里“暴露”有两层意思：

1. 包含你的头文件的代码会不会用你重载的 `::operator new()`，
2. 重载之后的 `::operator new()` 分配的内存能不能在你的 **library** 之外被安全地释放。如果不行，那么你是不是要暴露某个接口函数来让使用者安全地释放内存？或者返回 `shared_ptr`，利用其“捕获”析构动作（`deleter`）的特性？（§1.10）

听上去好像挺复杂？这里就不一一展开讨论了。总之，作为 **library** 的作者，我建议你绝对不要动“重载 `::operator new()`”的念头。

事实 2：在主程序里重载 `::operator new()` 的作用不大

这不是一条规则，而是我试图说明这么做没有多大意义。

如果用第一种方式重载全局 `::operator new(size_t)`，会影响本程序用到的所有 C++ **library**，这么做或许不会有什么问题，不过我建议你使用 §12.2.6 介绍的更简单的“替代办法”。

如果用第二种方式重载 `::operator new(size_t size, const char* file, int line)`，那么你的行为是否惠及本程序用到的其他 C++ **library** 呢？比方说你要不要统计 C++ **library** 中的内存使用情况？如果某个 **library** 会返回它自己用 `new` 分配的内存和对象，让你用完之后自己释放，那么是否打算对错误释放内存做检查？

C++ **library** 在代码组织上有两种形式：

1. 以头文件方式提供（如以 STL 和 Boost 为代表的模板库）；
2. 以头文件 + 二进制库文件方式提供（大多数非模板库以此方式发布）。

对于纯以头文件方式实现的 library，可以在你的程序的每个.cpp 文件的第一行包含重载 `::operator new()` 的头文件，这样程序里用到的其他 C++ library 也会转而使用你的 `::operator new()` 来分配内存。当然这是一种相当有侵略性的做法，如果运气好，编译和运行都没问题；如果运气差一点，可能会遇到编译错误，这其实还不算坏事；如果运气更差一点，编译没有错误，运行的时候时不时地出现非法访问，导致 `segment fault`；或者在某些情况下你定制的分配策略与 library 有冲突，内存数据损坏，出现莫名其妙的行为。

对于以库文件方式实现的 library，这么做并不能让其受惠，因为 library 的源文件已经编译成了二进制代码，它不会调用你新重载的 `::operator new`。（想想看，已经编译的二进制代码怎么可能提供额外的 `new (__FILE__, __LINE__)` 参数呢？）更麻烦的是，如果某些头文件有 `inline` 函数，还会引起诡异的“串扰”。即 library 有的部分用了你的分配器，有的部分用了系统默认的分配器，然后在释放内存的时候没有给对地方，造成分配器的数据结构被破坏。

总之，第二种重载方式看似功能更丰富，但其实与程序里使用的其他 C++ library 很难无缝配合。

综上，对于现实生活中的 C++ 项目，重载 `::operator new()` 几乎没有用武之地，因为很难处理好与程序所用的 C++ library 的关系，毕竟大多数 library 在设计的时候没有考虑到你会重载 `::operator new()` 并强塞给它。

如果确实需要定制内存分配，该如何办？

12.2.6 解决办法：替换 `malloc()`

很简单，替换 `malloc()`。如果需要，直接从 `malloc` 层面入手，通过 `LD_PRELOAD` 来加载一个 .so，其中有 `malloc/free` 的替代实现（`drop-in replacement`），这样能同时为 C 和 C++ 代码服务，而且避免 C++ 重载 `::operator new()` 的阴暗角落。

对于“检测内存错误”这一用法，我们可以用 `valgrind`、`dmalloc`、`efence` 来达到相同的目的，专业的除错工具比自己“山寨”一个内存检查器要靠谱。

对于“统计内存使用数据”，替换 `malloc` 同样能得到足够的信息，因为我们可以用 `backtrace()` 函数来获得调用栈，这比 `new (__FILE__, __LINE__)` 的信息更丰富。比方说你通过分析 `(__FILE__, __LINE__)` 发现 `std::string` 大量分配释放内存，有超出预期的开销，但是你却不知道代码里哪一部分在反复创建和销毁 `std::string` 对象，因为 `(__FILE__, __LINE__)` 只能告诉你最内层的调用函数。用 `backtrace()` 能找到真正的发起调用者。

对于“性能优化”这一用法，我认为在目前的多线程开发中，自己实现一个能打败系统默认的 `malloc` 的内存分配器是不现实的。一个通用的内存分配器本来就有相当的难度，为多线程程序实现一个安全和高效的通用（全局）内存分配器超出了一般开发人员的能力。不如使用现有的针对多核多线程优化的 `malloc`，例如 Google `tcmalloc` 和 Intel TBB 里的内存分配器⁸。好在这些 `allocator` 都不是侵入式的，也无须重载 `::operator new()`。

12.2.7 为单独的 `class` 重载 `::operator new()` 有问题吗

与全局 `::operator new()` 不同，`per-class operator new()` 和 `operator delete ()` 的影响面要小得多，它只影响本 `class` 及其派生类。似乎重载 `member ::operator new()` 是可行的。我对此持反对态度。

如果一个 `class Node` 需要重载 `member ::operator new()`，说明它用到了特殊的内存分配策略，常见的情况是使用了内存池或对象池。我宁愿把这一事实明显地摆出来，而不是改变 `new Node` 语句的默认行为。具体地说，是用 `factory` 来创建对象，比如 `static Node* Node::createNode()` 或者 `static shared_ptr<Node> Node::createNode()`。

这可以归结为最小惊讶原则：如果我在代码里读到 `Node* p = new Node`，我会认为它在 `heap` 上分配了内存。如果 `Node class` 重载了 `member ::operator new()`，那么我要事先仔细阅读 `node.h` 才能发现其实这行代码使用了私有的内存池。为什么不得写得明确一点呢？写成 `Node* p = NodeFactory::createNode()`，那么我能猜到 `NodeFactory::createNode()` 肯定做了什么与 `new Node` 不一样的事情，免得将来大吃一惊。

The Zen of Python⁹ 说 “explicit is better than implicit”，我深信不疑。

12.2.8 有必要自行定制内存分配器吗

如果写一个简单的只能分配固定大小的 `allocator`，确实很容易做到比系统的 `malloc` 更快，因为每次分配操作就是移动一下指针。但是我认为普通程序员很难写出可以与 `libc` 的 `malloc` 相媲美的通用内存分配器，在多核多线程时代更是如此。因为 `libc` 有专人维护，会不断把适合新硬件体系结构的分配算法与策略整合进去。在打

⁸ http://locklessinc.com/benchmarks_allocator.shtml

⁹ <http://www.python.org/dev/peps/pep-0020/>

算写自己的内存池之前, 建议先看一看 Andrei Alexandrescu 在 ACCU 2008 会议的演讲《Memory Allocation: Either Love it or Hate It (Or Think It's Just OK)》¹⁰ 和论文《Reconsidering Custom Memory Allocation》¹¹。

总结

重载 `::operator new()` 或许在某些临时的场合能应个急, 但是不应该作为一种策略来使用。如果需要, 我们可以从 `malloc` 层面入手, 彻底替换内存分配器。

12.3 带符号整数的除法与余数

最近研究整数到字符串的转换, 读到了 Matthew Wilson 的《Efficient Integer to String Conversions》系列文章¹²。他的巧妙之处在于, 用一个对称的 `digits` 数组搞定了负数转换的边界条件(二进制补码的正负整数表示范围不对称)。代码大致如下, 经过改写:

```
const char* convert(char buf[], int value)
{
    static char digits[19] =
        { '9', '8', '7', '6', '5', '4', '3', '2', '1',
          '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
    static const char* zero = digits + 9; // zero 指向 '0'

    // works for -2147483648 .. 2147483647
    int i = value;
    char* p = buf;
    do {
        // lsd - least significant digit
        int lsd = i % 10; // lsd 可能小于 0
        i /= 10;          // 是向下取整还是向零取整?
        *p++ = zero[lsd]; // 下标可能为负
    } while (i != 0);

    if (value < 0) {
        *p++ = '-';
    }
    *p = '\0';
    std::reverse(buf, p);
    return p; // p - buf 为整数长度
}
```

¹⁰ <http://accu.org/content/conf2008/Alexandrescu-memory-allocation.screen.pdf>

¹¹ <http://www.cs.umass.edu/~emery/pubs/berger-oopsla2002.pdf> <http://www.cs.umass.edu/~emery/talks/OOPSLA-2002.ppt>

¹² <http://synesis.com.au/publications.html> 搜 conversions

这段简短的代码对 32-bit int 的全部取值都是正确的（从 -2 147 483 648 到 2 147 483 647）。可以视为 itoa() 的参考实现，算是面试的标准答案。

读到这份代码，我的心中顿时升起一个疑虑：《C Traps and Pitfalls》第 7.7 节讲到，C 语言中的整数除法（/）和取模（%）运算在操作数为负的时候，结果是 implementation-defined¹³。

也就是说，如果 m、d 都是整数，

```
int q = m / d;  
int r = m % d;
```

那么 C 语言只保证 $m = q \times d + r$ 。如果 m、d 当中有负数，那么 q 和 r 的正负号是由实现决定的。比如 $(-13)/4 = (-3)$ 或 $(-13)/4 = (-4)$ 都是合法的。如果采用后一种实现，那么这段转换代码就错了（因为将有 $(-1)\%10 = 9$ ）。只有商向 0 取整，代码才能正常工作。

为了弄清这个问题，我研究了一番。

12.3.1 语言标准怎么说

C89 我手头没有 ANSI C89 的文稿，只好求助于 [K&R]，此书第 41 页第 2.5 节讲到“The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, ...”确实是实现相关的。为此，C89 专门提供了 div() 函数，这个函数算出的商是向 0 取整的，便于编写可移植的程序。我得再去查 C++ 标准。

C++98 第 5.6.4 节写道：“If the second operand of / or % is zero the behavior is undefined; otherwise $(a/b)*b + a\%b$ is equal to a. If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined.”C++ 也没有规定余数的正负号（C++03 的叙述与此一模一样）。

不过这里有一个注脚，提到“According to work underway toward the revision of ISO C, the preferred algorithm for integer division follows the rules defined in the ISO Fortran standard, ISO/IEC 1539:1991, in which the quotient is always rounded toward zero.”即 C 语言的修订标准会采用和 Fortran 一样的取整算法。我又去查了 C99 标准。

¹³ 网上能下载到的一份简略版也有相同的内容，见 <http://www.literateprogramming.com/ctraps.pdf> 第 7.5 节。

C99 第 6.5.5.6 节说 “When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded.”（脚注：This is often called “truncation toward zero”.)

C99 明确规定了商是向 0 取整的，也就是意味着余数的符号与被除数相同，前面的转换算法能正常工作。C99 Rationale¹⁴ 提到了这个规定的原因：“In Fortran, however, the result will always truncate toward zero, and the overhead seems to be acceptable to the numeric programming community. Therefore, C99 now requires similar behavior, which should facilitate porting of code from Fortran to C.”既然 Fortran 在数值计算领域都做了如此规定，说明开销（如果有的话）是可以接受的。

C++11 标准第 5.6.4 节采用了与 C99 类似的表述：“For integral operands the / operator yields the algebraic quotient with any fractional part discarded; (This is often called truncation towards zero.)”可见 C++ 还是尽力保持与 C 的兼容性。

小结：C89 和 C++98 都留给实现去决定，而 C99 和 C++11 都规定商向 0 取整，这算是语言的进步吧。

12.3.2 C/C++ 编译器的表现

我主要关心 G++ 和 VC++ 这两个编译器。需要说明的是，用代码案例来探查编译器的行为是靠不住的，尽管前面的代码在两个编译器下都能正常工作。除非在文档里有明确表述，否则编译器可能会随时更改实现——毕竟我们关心的就是 *implementation-defined* 行为。

G++ 4.4¹⁵：GCC always follows the C99 requirement that the result of division is truncated towards zero. G++ 一直遵循 C99 规范，商向 0 取整，算法能正常工作。

Visual C++ 2008¹⁶：The sign of the remainder is the same as the sign of the dividend. 这个说法与商向 0 取整是等价的，算法也能正常工作。

12.3.3 其他语言的规定

既然 C89/C++98/C99/C++0x 已经很有多样性了，索性弄清楚其他语言是怎么定义整数除法的。这里只列出笔者接触过的几种常用语言。

¹⁴ <http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf>

¹⁵ <http://gcc.gnu.org/onlinedocs/gcc/Integers-implementation.html>

¹⁶ <http://msdn.microsoft.com/en-us/library/eayc4fzk.aspx>

Java Java 语言规范¹⁷ 明确说 “Integer division rounds toward 0”。另外对于 int 整数除法溢出，特别规定不抛异常，且 $-2\,147\,483\,648 / -1 = -2\,147\,483\,648$ （以及相应的 long 版本）。

C# C# 3.0 语言规定¹⁸ “The division rounds the result towards zero”。对于溢出的情况，规定在 checked 上下文中抛 ArithmeticException 异常；在 unchecked 上下文里没有明确规定，可抛可不抛。（据了解，C# 1.0/2.0 可能有所不同。）

Python Python 在语言参考手册¹⁹ 的显著位置标明，商是向负无穷取整。（Plain or long integer division yields an integer of the same type; the result is that of mathematical division with the ‘floor’ function applied to the result.）

Ruby Ruby 的语言手册没有明说，不过库的手册²⁰ 说明了也是向负无穷取整。（The quotient is rounded toward-infinity.）

Perl Perl 语言默认按浮点数来计算除法²¹，所以没有这个问题。Perl 的整数取模运算规则与 Python/Ruby 一致。

不过要注意，use integer；有可能会改变运算结果，例如：

```
print -10 % 3; // => 2

use integers;
print -10 % 3; // => -1
```

Lua Lua 缺省没有整数类型，除法一律按浮点数来算，因此不涉及商的取整。

综上所述，在整数除法的取整问题上，语言分为两个阵营，脚本语言彼此是相似的，C99/C++11/Java/C# 则属于另一个阵营，在移植代码时要小心。既然 Python 和 Ruby 的官方解释器都是用 C 实现的，但是运算规则又自成一体，那么必定能从代码中找到证据。

12.3.4 脚本语言解释器代码

Python 的代码很好读，我很快就找到了 2.6.6 版实现整数除法和取模运算的函数 `i_divmod()`²²。

¹⁷ http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.17.2

¹⁸ <http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>

¹⁹ <http://docs.python.org/reference/expressions.html#binary-arithmetic-operations>

²⁰ http://www.ruby-doc.org/docs/ProgrammingRuby/html/ref_c_numeric.html#Numeric.divmod

²¹ <http://perldoc.perl.org/perlop.html#Multiplicative-Operators>

²² <http://svn.python.org/view/python/tags/r266/Objects/intobject.c?view=markup>

```

python/tags/r266/Objects/intobject.c
565 /* Return type of i_divmod */
566 enum divmod_result {
567     DIVMOD_OK,                /* Correct result */
568     DIVMOD_OVERFLOW,          /* Overflow, try again using longs */
569     DIVMOD_ERROR              /* Exception raised */
570 };
571
572 static enum divmod_result
573 i_divmod(register long x, register long y,
574          long *p_xdivy, long *p_xmody)
575 {
576     long xdivy, xmody;
577
578     if (y == 0) {
579         PyErr_SetString(PyExc_ZeroDivisionError,
580                         "integer division or modulo by zero");
581         return DIVMOD_ERROR;
582     }
583     /* (-sys.maxint-1)/-1 is the only overflow case. */
584     if (y == -1 && UNARY_NEG_WOULD_OVERFLOW(x))
585         return DIVMOD_OVERFLOW;
586     xdivy = x / y;
587     /* xdivy*y can overflow on platforms where x/y gives floor(x/y)
588      * for x and y with differing signs. (This is unusual
589      * behaviour, and C99 prohibits it, but it's allowed by C89;
590      * for an example of overflow, take x = LONG_MIN, y = 5 or x =
591      * LONG_MAX, y = -5.) However, x - xdivy*y is always
592      * representable as a long, since it lies strictly between
593      * -abs(y) and abs(y). We add casts to avoid intermediate
594      * overflow.
595      */
596     xmody = (long)(x - (unsigned long)xdivy * y);
597     /* If the signs of x and y differ, and the remainder is non-0,
598      * C89 doesn't define whether xdivy is now the floor or the
599      * ceiling of the infinitely precise quotient. We want the floor,
600      * and we have it iff the remainder's sign matches y's.
601      */
602     if (xmody && ((y ^ xmody) < 0) /* i.e. and signs differ */) {
603         xmody += y;
604         --xdivy;
605         assert(xmody && ((y ^ xmody) >= 0));
606     }
607     *p_xdivy = xdivy;
608     *p_xmody = xmody;
609     return DIVMOD_OK;
610 }

```

python/tags/r266/Objects/intobject.c

注意到这段代码甚至考虑了 $-2\,147\,483\,648 / -1$ 在 32-bit 下会溢出这个特殊情况，让我大吃一惊。宏定义 `UNARY_NEG_WOULD_OVERFLOW` 和函数 `int_mul()` 前面的注释也值得一读。

```

python/tags/r266/Objects/intobject.c
554 /* Integer overflow checking for unary negation: on a 2's-complement
555 * box, -x overflows iff x is the most negative long. In this case we
556 * get -x == x. However, -x is undefined (by C) if x /is/ the most
557 * negative long (it's a signed overflow case), and some compilers care.
558 * So we cast x to unsigned long first. However, then other compilers
559 * warn about applying unary minus to an unsigned operand. Hence the
560 * weird "0-".
561 */
562 #define UNARY_NEG_WOULD_OVERFLOW(x) \
563     ((x) < 0 && (unsigned long)(x) == 0-(unsigned long)(x))
python/tags/r266/Objects/intobject.c

python/tags/r266/Objects/intobject.c
489 /*
490 Integer overflow checking for * is painful: Python tried a couple ways, but
491 they didn't work on all platforms, or failed in endcases (a product of
492 -sys.maxint-1 has been a particular pain).
493
494 Here's another way:
495
496 The native long product x*y is either exactly right or *way* off, being
497 just the last n bits of the true product, where n is the number of bits
498 in a long (the delivered product is the true product plus i*2**n for
499 some integer i).
500
501 The native double product (double)x * (double)y is subject to three
502 rounding errors: on a sizeof(long)==8 box, each cast to double can lose
503 info, and even on a sizeof(long)==4 box, the multiplication can lose info.
504 But, unlike the native long product, it's not in *range* trouble: even
505 if sizeof(long)==32 (256-bit longs), the product easily fits in the
506 dynamic range of a double. So the leading 50 (or so) bits of the double
507 product are correct.
508
509 We check these two ways against each other, and declare victory if they're
510 approximately the same. Else, because the native long product is the only
511 one that can lose catastrophic amounts of information, it's the native long
512 product that must have overflowed.
513 */
514
515 static PyObject *
516 int_mul(PyObject *v, PyObject *w)
517 {
518     long a, b;
519     long longprod;           /* a*b in native long arithmetic */
520     double doubled_longprod; /* (double)longprod */
521     double doubleprod;       /* (double)a * (double)b */
522
523     CONVERT_TO_LONG(v, a);
524     CONVERT_TO_LONG(w, b);
525     /* casts in the next line avoid undefined behaviour on overflow */
526     longprod = (long)((unsigned long)a * b);
527     doubleprod = (double)a * (double)b;

```



```

528     doubled_longprod = (double)longprod;
529
530     /* Fast path for normal case: small multiplicands, and no info
531        is lost in either method. */
532     if (doubled_longprod == doubleprod)
533         return PyInt_FromLong(longprod);
534
535     /* Somebody somewhere lost info. Close enough, or way off? Note
536        that a != 0 and b != 0 (else doubled_longprod == doubleprod == 0).
537        The difference either is or isn't significant compared to the
538        true value (of which doubleprod is a good approximation).
539     */
540     {
541         const double diff = doubled_longprod - doubleprod;
542         const double absdiff = diff >= 0.0 ? diff : -diff;
543         const double absprod = doubleprod >= 0.0 ? doubleprod :
544             -doubleprod;
545         /* absdiff/absprod <= 1/32 iff
546            32 * absdiff <= absprod -- 5 good bits is "close enough" */
547         if (32.0 * absdiff <= absprod)
548             return PyInt_FromLong(longprod);
549         else
550             return PyLong_Type.tp_as_number->nb_multiply(v, w);
551     }
552 }

```

python/tags/r266/Objects/intobject.c

Ruby 的代码要混乱一些，花点时间还是能找到的。以下是 Ruby 1.8.7-p334 的实现，位于 `fixdivmod()` 函数。²³

```

2185 static void
2186 fixdivmod(x, y, divp, modp)
2187     long x, y;
2188     long *divp, *modp;
2189 {
2190     long div, mod;
2191
2192     if (y == 0) rb_num_zerodiv();
2193     if (y < 0) {
2194         if (x < 0)
2195             div = -x / -y;
2196         else
2197             div = - (x / -y);
2198     }
2199     else {
2200         if (x < 0)
2201             div = - (-x / y);
2202         else
2203             div = x / y;
2204     }

```

ruby/tags/v1_8_7_334/numeric.c

²³ http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/tags/v1_8_7_334/numeric.c?view=markup

```
2205     mod = x - div*y;
2206     if ((mod < 0 && y > 0) || (mod > 0 && y < 0)) {
2207         mod += y;
2208         div -= 1;
2209     }
2210     if (divp) *divp = div;
2211     if (modp) *modp = mod;
2212 }
```

— ruby/tags/v1_8_7_334/numeric.c

注意到 Ruby 的 Fixnum 整数的表示范围比机器字长小 1 bit，直接避免了溢出的可能。

12.3.5 硬件实现

既然 C/C++ 以效率著称，那么应该是贴近硬件实现的。我考察了几种常见的硬件平台，它们基本都支持 C99/C++11 的语意，也就是说新规定没有额外开销。列举如下。（其实我们只关心带符号除法，不过为了完整性，这里一并列出 unsigned/signed 整数除法指令。）

Intel x86/x64 Intel x86 系列的 DIV/IDIV 指令明确提到是向 0 取整，与 C99、C++11、Java、C# 一致。

MIPS 很奇怪，我在 MIPS 的参考手册里没有查到 DIV/DIVU 指令的取整方向，不过根据 Patterson & Hennessy²⁴ 的讲解，似乎向 0 取整硬件上实现起来比较容易。

ARM/Cortex-M3 ARM 没有硬件除法指令，所以不存在这个问题。Cortex-M3 有硬件除法，SDIV/UDIV 指令都是向 0 取整的。Cortex-M3 的除法指令不能同时算出余数，这很特殊。

MMIX MMIX 是 Donald Knuth 设计的 64-bit CPU，替换原来的 MIX 机器。DIV 和 DIVU 指令都是向负无穷取整，这是我知道的唯一支持 Python/Ruby 语义的“硬件”平台。

总结

想不到小小的整数除法都有这么多名堂。一段只涉及整数运算的代码，即便能在各种语法相似的语言里运行，结果也可能完全不同。把 C 语言里运行得好好的整数运算代码原样复制到 Python 里，也可能因为负数除法而出错。反之亦然，用 Python 编写的原型代码移植到 C/C++ 里也可能出现行为异常，不可不察。

在实际项目中，可以使用特定的指令加速，参见 <http://wm.ite.pl/articles/sse-itoa.html>。

²⁴ *Computer Organization and Design: The Hardware/Software Interface*, 4th ed.

12.4 在单元测试中 mock 系统调用

本书 §9.7 曾经谈到单元测试在分布式程序开发中的优缺点（主要是缺点）。但是，在某些情况下，单元测试是很有必要的，在测试 failure 场景的时候尤其重要，比如：

- 在开发存储系统时，模拟 `read(2)/write(2)` 返回 `EIO` 错误（有可能是磁盘写满了，也有可能是磁盘出现了坏道读不出数据）。
- 在开发网络库的时候，模拟 `write(2)` 返回 `EPIPE` 错误（对方意外断开连接）。
- 在开发网络库的时候，模拟自连接（self-connection），网络库应该用 `getsockname(2)` 和 `getpeername(2)` 判断是否是自连接，然后断开之。
- 在开发网络库的时候，模拟本地 ephemeral port 耗尽，`connect(2)` 返回 `EAGAIN` 临时错误。
- 让 `gethostbyname(2)` 返回我们预设的值，防止单元测试给公司的 DNS Server 带来太大压力。

这些 test case 恐怕很难用前文提到的 test harness 来测试，该单元测试上场了。现在的问题是，如何 mock 这些系统函数？或者换句话说，如何把对系统函数的依赖注入被测程序中？

12.4.1 系统函数的依赖注入

在《修改代码的艺术》[WELC] 一书第 4.3.2 节中，作者介绍了链接期接缝（link seam），正好可以解决我们的问题。另外，在 Stack Overflow 的一个帖子²⁵里也总结了几种做法。

如果程序（库）在编写的时候就考虑了可测试性，那么用不到上面的 hack 手段，我们可以从设计上解决依赖注入的问题。这里提供两个思路。

其一 采用传统的面向对象的手法，借助运行期的迟绑定实现注入与替换。自己写一个 System interface，把程序里用到的 `open`、`close`、`read`、`write`、`connect`、`bind`、`listen`、`accept`、`gethostname`、`getpeername`、`getsockname` 等等函数统统用虚函数封装一层。然后在代码里不要直接调用 `open()`，而是调用 `System::instance().open()`。这样代码主动把控制权交给了 System interface，我们可以在这里动动手脚。在写单元测试的时候，把这个 singleton instance 替换为我们的 mock object，这样就能模拟各种 error code。

²⁵ <http://stackoverflow.com/questions/2924440/advice-on-mocking-system-calls>

其二 采用编译期或链接期的迟绑定。注意到在第一种做法中，运行期多态是不必要的，因为程序从生到死只会用到一个 `implementation object`。为此付出虚函数调用的代价似乎有些不值。（其实，跟系统调用比起来，虚函数这点开销可忽略不计。）

我们可以写一个 `system namespace` 头文件，在其中声明 `read()` 和 `write()` 等普通函数，然后在 `.cc` 文件里转发给对应系统的系统函数 `::read()` 和 `::write()` 等。

```

namespace sockets                                     muduo/net/SocketsOps.h
{
    int connect(int sockfd, const struct sockaddr_in& addr);
}

int sockets::connect(int sockfd, const struct sockaddr_in& addr)
{
    return ::connect(sockfd, sockaddr_cast(&addr), sizeof addr);
}

```

有了这么一层间接性，就可以在编写单元测试的时候动动手脚，链接我们的 `stub` 实现，以达到替换实现的目的：

```

int sockets::connect(int sockfd, const struct sockaddr_in& addr)
{
    errno = EAGAIN;
    return -1;
}

```

一个 C++ 程序只能有一个 `main()` 入口，所以要把程序做成 `library`，再用单元测试代码链接这个 `library`。假设有一个 `mynetcat` 程序，为了编写 C++ 单元测试，我们把它拆成两部分，即 `library` 和 `main()`，源文件分别是 `mynetcat.cc` 和 `main.cc`。

在编译普通程序的时候：

```
g++ main.cc mynetcat.cc SocketsOps.cc -o mynetcat
```

在编译单元测试时这么写：

```
g++ test.cc mynetcat.cc MockSocketsOps.cc -o test
```

以上是最简单的例子。在实际开发中可以让 `stub` 功能更强大一些，比如根据不同的 `test case` 返回不同的错误。

第二种做法无须用到虚函数，代码写起来也比较简洁，只用前缀 `sockets::` 即可。例如在应用程序的代码里写 `sockets::connect(fd, addr)`。

muduo 目前还没有涉及系统调用的单元测试，只是预留了这些 stub。

namespace 的好处在于它不是封闭的，我们可以随时打开往里添加新的函数，而不用改动原来的头文件。这也是以 non-member non-friend 函数为接口的优点。

以上两种做法还有一个好处，即只 mock 我们关心的部分代码。如果程序用到了 SQLite 或 Berkeley DB 这些会访问本地文件系统的第三方库，那么我们的 System interface 或 system namespace 不会拦截这些第三方库的 open(2)、close(2)、read(2)、write(2) 等系统调用。

12.4.2 链接期垫片 (link seam)

如果程序在一开始编码的时候没有考虑单元测试，那么又该如何注入 mock 系统调用呢？

上面第二种做法已经给出了答案，那就是使用 link seam（链接期垫片）。

比方说要仿冒 connect(2) 函数，那么我们在单元测试程序里实现一个自己的 connect() 函数，它遮盖了同名的系统函数。在链接的时候，linker 会优先采用我们自己定义的函数。（这对动态链接是成立的；如果是静态链接，会报 multiple definition 错误。好在绝大多数情况下 libc 是动态链接的。）

```

typedef int (*connect_func_t)(int sockfd,
                               const struct sockaddr *addr,
                               socklen_t addrlen);

connect_func_t connect_func = dlsym(RTDL_NEXT, "connect");

bool mock_connect;
int mock_connect_errno;

// mock connect
extern "C" int connect(int sockfd,
                      const struct sockaddr *addr,
                      socklen_t addrlen)
{
    if (mock_connect) {
        errno = mock_connect_errno;
        return errno == 0 ? 0 : -1;
    } else {
        return connect_func(sockfd, addr, addrlen);
    }
}

```

如果程序真的要调用 `connect(2)` 怎么办? 在我们自己的 `mock connect(2)` 里不能再调用 `connect()` 了, 否则会出现无限递归。为了防止这种情况, 我们用 `dlsym(RTDL_NEXT, "connect")` 获得 `connect(2)` 系统函数的真实地址, 然后通过函数指针 `connect_func` 来调用它。

例子: ZooKeeper 的 C client library

ZooKeeper 的 C client library 正是采用了 link seams 来编写单元测试, 代码见:

<http://svn.apache.org/repos/asf/zookeeper/tags/release-3.3.3/src/c/tests/LibCMocks.h>

<http://svn.apache.org/repos/asf/zookeeper/tags/release-3.3.3/src/c/tests/LibCMocks.cc>

其他做法

Stack Overflow 的帖子里还提到了一个做法, 可以方便地替换动态库里的函数, 即使用 `ld(1)` 的 `--wrap` 参数, 文档里说得很清楚, 这里不再赘述。

第三方 C++ 库

Link seam 同样适用于第三方 C++ 库

比方说公司的某个基础库团队提供了 `File class`, 但是这个 `class` 没有使用虚函数, 我们无法通过 `sub-classing` 的办法来实现 `mock object`。

```
class File : boost::noncopyable
{
public:
    File(const char* filename);
    ~File();

    int readn(void* data, int len);
    int writen(const void* data, int len);
    size_t getSize() const;
private:
};
```

File.h

File.h

如果需要为用到 `File class` 的程序编写单元测试, 那么我们可以自己定义其成员函数的实现, 这样可以注入任何我们想要的结果。

```
int File::readn(void* data, int len)
{
    return -1;
}
```

MockFile.cc

MockFile.cc

这个做法对动态库是可行的，但对于静态库则会报错。我们要么让对方提供专供单元测试的动态库，要么拿过源码来自己编译一个。

Java 也有类似的做法，在 `class path` 里替换我们自己的 `stub jar` 文件，以实现 `link seam`。不过 Java 有很强的反射机制，很少用得着 `link seam` 来实现依赖注入。

12.5 慎用匿名 namespace

匿名 namespace（anonymous namespace 或称 unnamed namespace）是 C++ 语言的一项非常有用的功能，其主要目的是让该 namespace 中的成员（变量或函数）具有独一无二的全局名称，避免名字碰撞（name collisions）。一般在编写 `.cpp` 文件时，如果需要写一些小的 helper 函数，我们常常会放到匿名 namespace 里。muduo 0.1.7 中的 `muduo/base/Date.cc` 和 `muduo/base/Thread.cc` 等处就用到了匿名 namespace。

我最近在工作中遇到并重新思考了这一问题，发现匿名 namespace 并不是多多益善。

12.5.1 C 语言的 static 关键字的两种用法

C 语言的 static 关键字有两种用途：

第 1 种 用于函数内部修饰变量，即函数内的静态变量。这种变量的生存期长于该函数，使得函数具有一定的“状态”。使用静态变量的函数一般是不可重入的，也不是线程安全的，比如 `strtok(3)`。

第 2 种 用在文件级别（函数体之外），修饰变量或函数，表示该变量或函数只在本文件可见，其他文件看不到、也访问不到该变量或函数。专业的说法叫“具有 internal linkage”（简言之：不暴露给别的 translation unit）。

C 语言的这两种用法很明确，一般也不容易混淆。

12.5.2 C++ 语言的 static 关键字的四种用法

由于 C++ 引入了 class，在保持与 C 语言兼容的同时，static 关键字又有了两种新用法：

第 3 种 用于修饰 class 的数据成员，即所谓“静态成员”。这种数据成员的生存期大于 class 的对象（实体/instance）。静态数据成员是每个 class 有一份，普通数据成员是每个 instance 有一份，因此也分别叫做 class variable 和 instance variable。

第 4 种 用于修饰 class 的成员函数，即所谓“静态成员函数”。这种成员函数只能访问 class variable 和其他静态程序函数，不能访问 instance variable 或 instance method。

当然，这几种用法可以相互组合，比如 C++ 的成员函数（无论 static 还是 instance）都可以有其局部的静态变量（上面的用法 1）。对于 class template 和 function template，其中的 static 对象的真正个数跟 template instantiation（模板具现化）有关，相信学过 C++ 模板的人不会陌生。

可见在 C++ 里 static 被 overload 了多次。匿名 namespace 的引入是为了减轻 static 的负担，它替换了 static 的第 2 种用途。也就是说，在 C++ 里不必使用文件级的 static 关键字，我们可以用匿名 namespace 达到相同的效果。（其实严格地说，linkage 或许稍有不同，这里不展开讨论了。）

12.5.3 匿名 namespace 的不利之处

在工程实践中，匿名 namespace 有两大不利之处：

1. 匿名 namespace 中的函数是“匿名”的，那么在确实需要引用它的时候就比较麻烦。

比如在调试的时候不便给其中的函数设断点，如果你像我一样使用的是 gdb 这样的文本模式 debugger；又比如 profiler 的输出结果也不容易判别到底是哪个文件中的 calculate() 函数需要优化。

2. 使用某些版本的 g++ 时，同一个文件每次编译出来的二进制文件会变化。

比如说拿到一个会发生 core dump 的二进制可执行文件，无法确定它是由哪个 revision 的代码编译出来的。毕竟编译结果不可复现，具有一定的随机性。（当然，在正式场合，这应该由软件配置管理（SCM）流程来解决。）

另外这也可能让某些 build tool 失灵，如果该工具用到了编译出来的二进制文件的 MD5 的话。

考虑下面这段简短的代码：

```
namespace
{
    void foo()
    {
    }
}
```

anon.cc


```
int main()
{
    foo();
}
```

anon.cc

对于问题 1: gdb 的 <tab> 键自动补全功能能帮我们设定断点, 不是什么大问题。前提是你知道那个 “(anonymous namespace)::foo()” 正是你想要的函数。

```
$ gdb ./a.out
GNU gdb (GDB) 7.0.1-debian

(gdb) b '<tab>
(anonymous namespace)      __data_start      _end
(anonymous namespace)::foo()  __do_global_ctors_aux    _fini
_DYNAMIC                    __do_global_dtors_aux    _init
_GLOBAL_OFFSET_TABLE_      __dso_handle            _start
_IO_stdin_used              __gxx_personality_v0     anon.cc
__CTOR_END__                __gxx_personality_v0@plt call_gmon_start
__CTOR_LIST__               __init_array_end        completed.6341
__DTOR_END__                __init_array_start      data_start
__DTOR_LIST__               __libc_csu_fini          dtor_idx.6343
__FRAME_END__               __libc_csu_init          foo
__JCR_END__                 __libc_start_main        frame_dummy
__JCR_LIST__                __libc_start_main@plt    int
__bss_start                 _edata                   main

(gdb) b '(<tab>
anonymous namespace)      anonymous namespace)::foo()

(gdb) b '(anonymous namespace)::foo()'
Breakpoint 1 at 0x400588: file anon.cc, line 4.
```

麻烦的是, 如果两个文件 anon.cc 和 anonlib.cc 都定义了匿名空间中的 foo() 函数 (这不会冲突), 那么 gdb 无法区分这两个函数, 你只能给其中一个设断点。或者你使用 文件名: 行号 的方式来分别设断点。(从技术上说, 匿名 namespace 中的函数是 weak text, 链接的时候如果发生符号重名, linker 不会报错。)

从根本上解决的办法是使用普通具名 namespace, 如果怕重名, 可以把源文件名 (必要时加上路径) 作为 namespace 名字的一部分。

对于问题 2: 把 anon.cc 编译两次, 分别生成 a.out 和 b.out:

```
$ g++ --version
g++ (GCC) 4.2.4 (Ubuntu 4.2.4-1ubuntu4)

$ g++ -g -o a.out anon.cc
$ g++ -g -o b.out anon.cc
```

```
$ md5sum a.out b.out
0f7a9cc15af7ab1e57af17ba16afcd70 a.out
8f22fc2bbfc27beb922aefa97d174e3b b.out

$ diff -u <(nm a.out) <(nm b.out)
--- /dev/fd/63 2011-02-15 22:27:58.960754999 +0800
+++ /dev/fd/62 2011-02-15 22:27:58.960754999 +0800
@@ -2,7 +2,7 @@
0000000000600940 d _GLOBAL_OFFSET_TABLE_
0000000000400634 R _IO_stdin_used
w _Jv_RegisterClasses
-0000000000400538 t _ZN36_GLOBAL__N_anon.cc_00000000_E2CEE513fooEv
+0000000000400538 t _ZN36_GLOBAL__N_anon.cc_00000000_CB51498D3fooEv
0000000000600748 d __CTOR_END__
0000000000600740 d __CTOR_LIST__
0000000000600758 d __DTOR_END__
```

由上可见，g++ 4.2.4 会随机地给匿名 namespace 生成一个唯一的名字（foo() 函数的 mangled name 中的 E2CEE51 和 CB51498D 是随机的），以保证名字不冲突。也就是说，同样的源文件，两次编译得到的二进制文件内容不相同，这有时候会造成问题或困惑。

这可以用 gcc 的 `-frandom-seed` 参数解决，具体见 gcc 文档。

这个现象在 gcc 4.2.4 中存在（之前的版本估计类似），在 gcc 4.4.5 中不存在。

12.5.4 替代办法

如果前面的“不利之处”给你带来了困扰，解决办法也很简单，就是使用普通具名 namespace。当然，要起一个好的名字，比如 Boost 里就常常用 `boost::detail` 来放那些“不应该暴露给客户，但又不得不放到头文件里”的函数或 class。

总而言之，匿名 namespace 没什么大问题，使用它也不是什么过错。万一它碍事了，可以用普通具名 namespace 替代之。

12.6 采用有利于版本管理的代码格式

版本管理（version controlling）是每个程序员的基本技能，C++ 程序员也不例外。版本管理的基本功能之一是追踪代码变化，让你能清楚地知道代码是如何一步步变成现在的这个样子的，以及每次 check-in 都具体改动了哪些内部。无论是传统的集中式版本管理工具，如 Subversion，还是新型的分布式管理工具，如 Git/Hg，比较两个版本（revision）的差异都是其基本功能，即俗称“做一下 diff”。

diff 的输出是个窥孔 (peephole)，它的上下文有限 (diff -u 默认显示前后 3 行)。在做 code review 的时候，如果仅凭这“一孔之见”就能发现代码改动有问题，那就再好不过了。

C 和 C++ 都是自由格式的语言，代码中的换行符被当做 white space 来对待。(当然，我们说的是预处理 (preprocess) 之后的情况)。对编译器来说一模一样的代码可以有多种写法，比如

```
foo(1, 2, 3, 4);
```

和

```
foo(1,
    2,
    3,
    4);
```

词法分析的结果是一样的，语意也完全一样。

对人来说，这两种写法读起来不一样，对于版本管理工具来说，同样功能的修改造成的差异 (diff) 也往往不一样。所谓“有利于版本管理”，就是指在代码中合理使用换行符，对 diff 工具友好，让 diff 的结果清晰明了地表达代码的改动。diff 一般以行为单位，也可以以单词为单位，本文只考虑最常见的逐行比较 (diff by lines)。

12.6.1 对 diff 友好的代码格式

多行注释也用 //，不用 /* */

Scott Meyers 写的《Effective C++ (第 2 版)》第 4 条建议使用 C++ 风格，我这里为他补充一条理由：对 diff 友好。比如，我要注释一大段代码 (其实这不是个好的做法，但是在实践中有时会遇到)，如果用 /* */，那么得到的 diff 是：

```
--- a/examples/asio/tutorial/timer5/timer.cc
+++ b/examples/asio/tutorial/timer5/timer.cc
@@ -18,6 +18,7 @@ class Printer : boost::noncopyable
 }

+ /*
+ ~Printer()
+ {
@@ -38,6 +39,7 @@ class Printer : boost::noncopyable
 }
+ */

void print2()
{
```

Linux 多线程服务端编程：使用 muduo C++ 网络库

从这样的 diff output 能看出注释了哪些代码吗？

如果用 //，结果会清晰很多：

```

--- a/examples/asio/tutorial/timer5/timer.cc
+++ b/examples/asio/tutorial/timer5/timer.cc
@@ -18,26 +18,26 @@ class Printer : boost::noncopyable
    loop2_>runAfter(1, boost::bind(&Printer::print2, this));
    }

- ~Printer()
- {
-     std::cout << "Final count is " << count_ << "\n";
- }
+ // ~Printer()
+ // {
+ //     std::cout << "Final count is " << count_ << "\n";
+ // }

- void print1()
- {
-     muduo::MutexLockGuard lock(mutex_);
-     if (count_ < 10)
-     {
-         std::cout << "Timer 1: " << count_ << "\n";
-         ++count_;
-
-         loop1_>runAfter(1, boost::bind(&Printer::print1, this));
-     }
-     else
-     {
-         loop1_>quit();
-     }
- }
+ // void print1()
+ // {
+ //     muduo::MutexLockGuard lock(mutex_);
+ //     if (count_ < 10)
+ //     {
+ //         std::cout << "Timer 1: " << count_ << "\n";
+ //         ++count_;
+ //
+ //         loop1_>runAfter(1, boost::bind(&Printer::print1, this));
+ //     }
+ //     else
+ //     {
+ //         loop1_>quit();
+ //     }
+ // }

    void print2()
    {

```

同样的道理，取消注释的时候 // 也比 /* */ 更清晰。

另外，如果用 `/* */` 来做多行注释，从 diff 不一定能看出来你是在修改代码还是修改注释。比如以下 diff 似乎修改了 `muduo::EventLoop::runAfter()` 的调用参数：

```
--- a/examples/asio/tutorial/timer5/timer.cc
+++ b/examples/asio/tutorial/timer5/timer.cc
@@ -32,7 +32,7 @@ class Printer : boost::noncopyable
     std::cout << "Timer 1: " << count_ << std::endl;
     ++count_;

-    loop1->runAfter(1, boost::bind(&Printer::print1, this));
+    loop1->runAfter(2, boost::bind(&Printer::print1, this));
 }
 else
 {
```

其实这个修改发生在注释中（要增加上下文才能看到，diff -U 20，多一道手续，降低了工作效率），对代码行为没有影响：

```
--- a/examples/asio/tutorial/timer5/timer.cc
+++ b/examples/asio/tutorial/timer5/timer.cc
@@ -20,31 +20,31 @@ class Printer : boost::noncopyable

/*
~Printer()
{
    std::cout << "Final count is " << count_ << std::endl;
}

void print1()
{
    muduo::MutexLockGuard lock(mutex_);
    if (count_ < 10)
    {
        std::cout << "Timer 1: " << count_ << std::endl;
        ++count_;

-        loop1->runAfter(1, boost::bind(&Printer::print1, this));
+        loop1->runAfter(2, boost::bind(&Printer::print1, this));
    }
    else
    {
        loop1->quit();
    }
}
*/

void print2()
{
    muduo::MutexLockGuard lock(mutex_);
    if (count_ < 10)
    {
        std::cout << "Timer 2: " << count_ << std::endl;
        ++count_;
```

总之，不要用 `/* */` 来注释多行代码。

或许是时过境迁了，大家都在用 `//` 注释了，《Effective C++（第3版）》去掉了这一条建议。

局部变量与成员变量的定义

基本原则是，一行代码只定义一个变量，比如

```
double x;
double y;
```

将来代码增加一个 `double z` 的时候，`diff` 输出一眼就能看出改了什么：

```
@@ -63,6 +63,7 @@ private:
    int count_;
    double x;
    double y;
+   double z;
};

int main()
```

如果把 `x` 和 `y` 写在一行，`diff` 的输出就得多看几眼才知道：

```
@@ -61,7 +61,7 @@ private:
    muduo::net::EventLoop* loop1_;
    muduo::net::EventLoop* loop2_;
    int count_;
-   double x, y;
+   double x, y, z;
};

int main()
```

所以，一行只定义一个变量更利于版本管理。同样的道理适用于 `enum` 成员的定义、数组的初始化列表等。

函数声明中的参数

如果函数的参数大于 3 个，那么在逗号后面换行，这样每个参数占一行，便于 `diff`。以 `muduo::net::TcpClient` 为例：

```
class TcpClient : boost::noncopyable
{
public:
    TcpClient(EventLoop* loop,
              const InetAddress& serverAddr,
              const string& name);
```

muduo/net/TcpClient.h

muduo/net/TcpClient.h

如果将来 `TcpClient` 的构造函数增加或修改一个参数，那么很容易从 diff 看出来。这恐怕比在一行长代码里数逗号要高效一些。

函数调用时的参数

在函数调用的时候，如果参数大于 3 个，那么把实参分行写。

以 `muduo::net::EPollPoller` 为例：

```
Timestamp EPollPoller::poll(int timeoutMs, ChannelList* activeChannels)
{
    int numEvents = ::epoll_wait(epollfd_,
                                &*events_.begin(),
                                static_cast<int>(events_.size()),
                                timeoutMs);
    Timestamp now(Timestamp::now());
```

这样一来，如果将来重构引入了一个新参数（当然，`epoll_wait` 不会有这个问题），那么函数定义和函数调用的地方的 diff 具有相同的形式（比方说都是在倒数第二行加了一行内容），很容易肉眼验证有没有错位。如果参数写在一行里边，就得睁大眼睛数逗号了。

class 初始化列表的写法

同样的道理，class 初始化列表（initializer list）也遵循一行一个的原则，这样将来如果加入新的成员变量，那么两处（class 定义和 ctor 定义）的 diff 具有相同的形式，让错误无所遁形。以 `muduo::net::Buffer` 为例：

```
class Buffer : public muduo::copyable
{
public:
    static const size_t kCheapPrepend = 8;
    static const size_t kInitialSize = 1024;

    Buffer()
        : buffer_(kCheapPrepend + kInitialSize),
          readerIndex_(kCheapPrepend),
          writerIndex_(kCheapPrepend)
    { }
    // 省略
private:
    std::vector<char> buffer_;
    size_t readerIndex_;
    size_t writerIndex_;
};
```

注意，初始化列表的顺序必须和数据成员声明的顺序相同。

与 namespace 有关的缩进

Google 的 C++ 编程规范明确指出，namespace 不增加缩进²⁶。这么做非常有道理，方便 `diff -p` 把函数名显示在每个 diff chunk 的头上。

如果对函数实现做 diff，chunk name 是函数名，让人一眼就能看出改的是哪个函数，如下面所示的灰底部分。

```
diff --git a/muduo/net/SocketsOps.cc b/muduo/net/SocketsOps.cc
--- a/muduo/net/SocketsOps.cc
+++ b/muduo/net/SocketsOps.cc
@@ -125,7 +125,7 @@ int sockets::accept(int sockfd, struct sockaddr_in* addr)
     case ENOTSOCK:
     case EOPNOTSUPP:
         // unexpected errors
-        LOG_FATAL << "unexpected error of ::accept";
+        LOG_FATAL << "unexpected error of ::accept " << savedErrno;
         break;
     default:
         LOG_FATAL << "unknown error of ::accept " << savedErrno;
```

如果对 class 做 diff，那么 chunk name 就是 class name。

```
diff --git a/muduo/net/Buffer.h b/muduo/net/Buffer.h
--- a/muduo/net/Buffer.h
+++ b/muduo/net/Buffer.h
@@ -60,13 +60,13 @@ class Buffer : public muduo::copyable
     std::swap(writerIndex_, rhs.writerIndex_);
 }

- size_t readableBytes();
+ size_t readableBytes() const;

- size_t writableBytes();
+ size_t writableBytes() const;

- size_t prependableBytes();
+ size_t prependableBytes() const;

const char* peek() const;
```

diff 原本是为 C 语言设计的，C 语言没有 namespace 缩进一说，所以它默认会找到“顶格写”的函数作为一个 diff chunk 的名字。如果函数名前面有空格，它就不认得了。muduo 的代码都遵循这一规则，例如：

²⁶ http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Namespace_Formatting

```

namespace muduo
{
    // class 从第一列开始写，不缩进
    class Timestamp : public muduo::copyable
    {
        // ...
    };
}

```

```

// 函数的实现也从第一列开始写，不缩进。
Timestamp Timestamp::now()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    int64_t seconds = tv.tv_sec;
    return Timestamp(seconds * kMicroSecondsPerSecond + tv.tv_usec);
}

```

相反，Boost 中的某些库的代码是按 namespace 来缩进的，这样的话看 diff 往往不知道改动的是哪个 class 的哪个成员函数。

这个或许可以通过设置 diff 取函数名的正则表达式来解决，但是如果我们写代码的时候就注意把函数“顶格写”，那么就不用去动 diff 的默认设置了。另外，正则表达式不能完全匹配函数名，因为函数名属于上下文无关语法（context-free syntax），你没办法写一个正则语法去匹配上下文无关语法。我总能写出某种函数声明，让你的正则表达式失效（想想函数的返回类型，它可能是一个非常复杂的东西，更别说参数了）。更何况 C++ 的语法是上下文相关的，比如，你猜 `Foo<Bar> qux;` 是个表达式还是变量定义？

public 与 private

我认为这是 C++ 语法的一个缺陷，如果我把一个成员函数从 public 区移到 private 区，那么从 diff 上看不出来我干了什么，例如：

```

@@ -37,7 +37,6 @@ class TcpClient : boost::noncopyable
    void connect();
    void disconnect();

-   bool retry() const;
-   void enableRetry() { retry_ = true; }

    /// Set connection callback.
@@ -60,6 +59,7 @@ class TcpClient : boost::noncopyable

```

```

void newConnection(int sockfd);
/// Not thread safe, but in loop
void removeConnection(const TcpConnectionPtr& conn);
+ bool retry() const;

EventLoop* loop_;
boost::scoped_ptr<Connector> connector_; // avoid revealing Connector

```

从上面的 diff 能看出我把 `retry()` 变成 `private` 了吗？对此我也没有好的解决办法，总不能在每个函数前面都写上 `public:` 或 `private:` 吧？

对此 Java 和 C# 都做得比较好，它们把 `public/private` 等修饰符放到每个成员函数的定义中。这么做增加了信息的冗余度，让 diff 的结果更直观。

避免使用版本控制软件的 **keyword substitution** 功能

这么做是为了避免 diff 噪声。

比方说，如果我想比较 0.1.1 和 0.1.2 两个代码分支有哪些改动，我通常会在 `branches` 目录执行 `diff 0.1.1 0.1.2 -ru`。两个 branch 中的 `muduo/net/EventLoop.h` 其实是一样的（先后从同一个 revision 分支出来）。但是如果这个文件使用了 SVN 的 `keyword substitution` 功能（比如 `$Id$`），diff 会报告这两个 branches 中的文件不一样，如下所示。

```

diff -rup 0.1.1/muduo/net/EventLoop.h 0.1.2/muduo/net/EventLoop.h
--- 0.1.1/muduo/net/EventLoop.h 2011-05-02 23:11:02.000000000 +0800
+++ 0.1.2/muduo/net/EventLoop.h 2011-05-02 23:12:22.000000000 +0800
@@ -8,7 +8,7 @@
//
// This is a public header file, it must only include public header files.

-// $Id: EventLoop.h 4 2011-05-01 10:11:02Z schen $
+// $Id: EventLoop.h 5 2011-05-02 15:12:22Z schen $

#ifndef MUDUO_NET_EVENTLOOP_H
#define MUDUO_NET_EVENTLOOP_H

```

这样纯粹增加了噪声，这是 RCS/CVS 时代的过时做法。文件的 Id 不应该在文件内容中出现，这些 metadata 跟源文件的内容无关，应该由版本管理软件额外提供。

12.6.2 对 **grep** 友好的代码风格

操作符重载

C++ 工具匮乏，在一个项目里，要找到一个函数的定义或许不算太难（最多就是分析一下重载和模板特化），但是要找到一个函数的使用就难多了。不比 Java，在 Eclipse 里按 `Ctrl+Shift+G` 组合键就能找到所有的引用点。

Linux 多线程服务端编程：使用 muduo C++ 网络库

假如我要做一个重构，想先找到代码里所有用到 `muduo::timeDifference()` 的地方，判断一下工作是否可行，基本上唯一的办法是 `grep`。用 `grep` 还不能排除同名的函数和注释里的内容。这也说明了为什么要用 `//` 来引导注释，因为在 `grep` 的时候，一眼就能看出这行代码是在注释里的。

在我看来，`operator overloading` 应仅限于和 STL `algorithm/container` 配合时使用，比如 `std::transform()` 和 `map<Key, Value>`，其他情况都用具名函数为宜。原因之一是，我根本用 `grep` 找不到在哪儿用到了减号 `operator-`。这也是 `muduo::Timestamp` class 只提供 `operator<()` 而不提供 `operator+()` `operator-` 的原因。我提供了两个函数 `timeDifference()` 和 `addTime()` 来实现所需的功能。

又比如，Google Protocol Buffers 的回调是 `Closure` class，它的接口用的是 `virtual function Run()` 而不是 `virtual operator()()`。

static_cast 与 C-style cast

为什么 C++ 要引入 `static_cast` 之类的转型操作符，原因之一就是像 `(int*) pBuffer` 这样的表达式基本上没办法用 `grep` 判断出它是个强制类型转换，写不出一个刚好只匹配类型转换的正则表达式。（其语法是上下文无关的，无法用正则搞定。）

如果类型转换都用 `*_cast`，那只要 `grep` 一下，我就能知道代码里哪儿用了 `reinterpret_cast` 转换，便于迅速地检查有没有用错。为了强调这一点，`muduo` 开启了编译选项 `-Wold-style-cast` 来帮助查找 C-style casting，这样在编译时就能帮我们找到问题。

12.6.3 一切为了效率

如果用图形化的文件比较工具，似乎能避免上面列举的问题。但无论是 Web 还是客户端，无论是 `diff by words` 还是 `diff by lines` 都不能解决全部问题，效率也不一定更高。

对于 p. 533 举的例子，如果想知道是谁在什么时候增加的 `double z`，在分行写的情况下，用 `git blame` 或 `svn blame` 立刻就能找到始作俑者。如果写成一行，那就得把文件的 `revisions` 拿来一个个人工比较，因为这一行 `double x = 0.0, y = 1.0, z = -1.0;` 可能修改过多次，你得一个个看才知道什么时候加入了变量 `z`。另外几种情况也使得 `blame` 的输出更易读。

比如 p. 535 改动了一行代码，你还是要向上翻页去找改的是哪个函数。人眼看的话还有“看走眼”的可能，又得再定睛观瞧。这一切都是在浪费人的时间，使用更好的图形化工具并不能减少浪费；相反，我认为增加了浪费。

另外一个常见的工作场景，早上来到办公室，`update` 一下代码，然后扫一眼 `diff output` 看看别人昨天动了哪些文件，改了哪些代码。这就是一两条命令的事，几秒就能结束战斗。如果用图形化的工具，得一个个点击文件 `diff` 的链接或打开新 `tab` 来看文件的 `side-by-side` 比较（不这么做的话就看不到足够多的上下文，跟看 `diff output` 无异），然后上下翻动页面去看别人到底改了什么。说实话，我觉得这么做效率并不比 `diff` 高。

12.7 再探 `std::string`

Scott Meyers 在《Effective STL》[ESTL] 第 15 条提到 `std::string` 有多种实现方式，归纳起来有三类，而每类又有多种变化。

- 1. 无特殊处理（eager copy），采用类似 `std::vector` 的数据结构。现在很少有实现采用这种方式。
- 2. Copy-on-Write（COW）。g++ 的 `std::string` 一直采用这种方式实现²⁷。
- 3. 短字符串优化（SSO），利用 `string` 对象本身的空间来存储短字符串。Visual C++ 用的是这种实现方式。

表 12-1 总结了我知道的各个库的 `string` 实现方式和 `string` 对象分别在 32-bit/64-bit x86 系统中的大小。

表 12-1

库	32-bit	64-bit	实现方式
g++ <code>std::string</code>	4	8	COW
<code>__gnu_cxx::__sso_string</code>	24	32	SSO
<code>__gnu_cxx::__rc_string</code>	4	8	COW
clang <code>libc++</code>	12	24	SSO
SGI STL	12	24	eager copy
STLPort	24	48	SSO
Apache <code>libstdcxx</code>	4	8	COW
Visual C++ 2010	28/32	40/48	SSO

²⁷ `libstdc++` 的 `std::string` 是 Nathan Myers 的手笔。

Visual C++ 的 `string` 的大小跟编译模式有关，表 12-1 中小的那个数字是 `release` 编译，大的是 `debug` 编译。因此 `debug` 库和 `release` 库不能混用。除此之外，其他库的 `string` 大小是固定的。

以下分别介绍这几种实现方式的代码骨架和数据结构示意图，无论哪种实现方式都要保存三个数据：1. 字符串本身（`char[]`），2. 字符串的长度（`size`），3. 字符串的容量（`capacity`）。

12.7.1 直接拷贝（eager copy）

类似 `std::vector` 的“三指针”结构。代码骨架（省略模板）如下，数据结构示意图如图 12-1 所示。

```

// http://www.sgi.com/tech/stl/string
// Class invariants:
// (1) [start, finish) is a valid range.
// (2) Each iterator in [start, finish) points to a valid object
//     of type value_type.
// (3) *finish is a valid object of type value_type; in particular,
//     it is value_type().
// (4) [finish + 1, end_of_storage) is a valid range.

// (5) Each iterator in [finish + 1, end_of_storage) points to
//     uninitialized memory.

// Note one important consequence: a string of length n must manage
// a block of memory whose size is at least n + 1.

class string
{
public:
    const_pointer data() const { return start; }
    iterator begin()          { return start; }
    iterator end()             { return finish; }
    size_type size() const     { return finish - start; }
    size_type capacity() const { return end_of_storage - start; }

private:
    char* start;
    char* finish;
    char* end_of_storage;
};

```

eager copy string 1

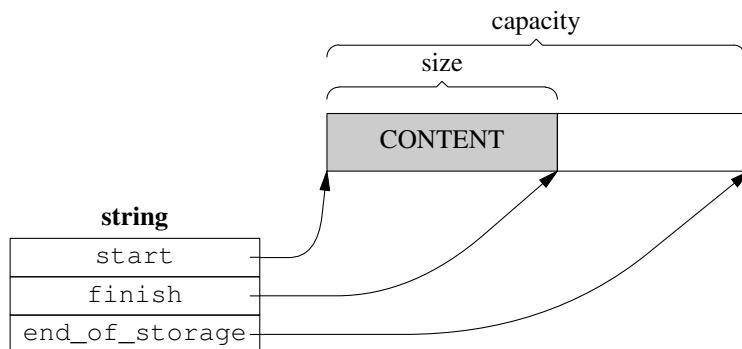


图 12-1

对象的大小是 3 个指针，在 32-bit 中是 12 字节，在 64-bit 中是 24 字节。

Eager copy string 的另一种实现方式是把后两个成员变量替换成整数，表示字符串的长度和容量，代码骨架如下，数据结构示意图如图 12-2 所示。

```
class string
{
public:
    const_pointer data() const { return start; }
    iterator begin()          { return start; }
    iterator end()            { return start + size_; }
    size_type size() const    { return size_; }
    size_type capacity() const { return capacity_; }

private:
    char* start;
    size_t size_;
    size_t capacity_;
};
```

eager copy string 2

eager copy string 2

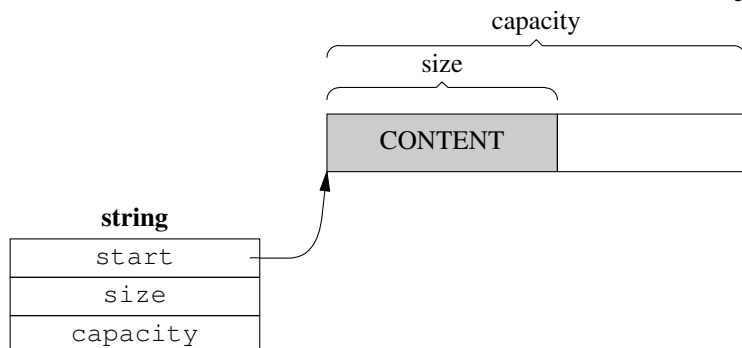


图 12-2

这种做法并没有多大的改变，因为 `size_t` 和 `char*` 是一样大的。但是，我们通常用不到单个几百兆字节的字符串²⁸，那么可以再改变一下长度和容量的类型（从 64-bit 整数改成 32-bit 整数），这样在 64-bit 下可以减小对象的大小，如图 12-3 所示。

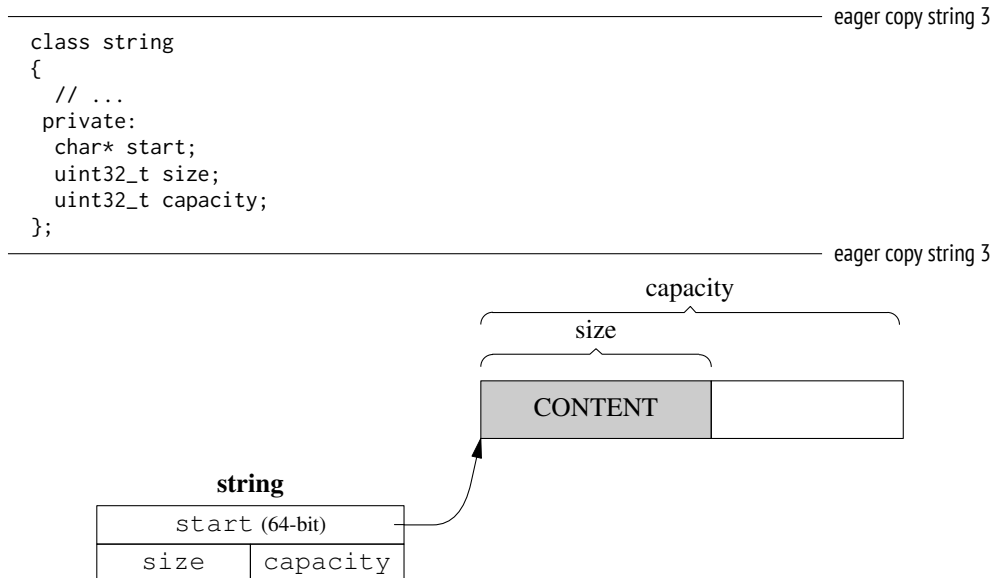
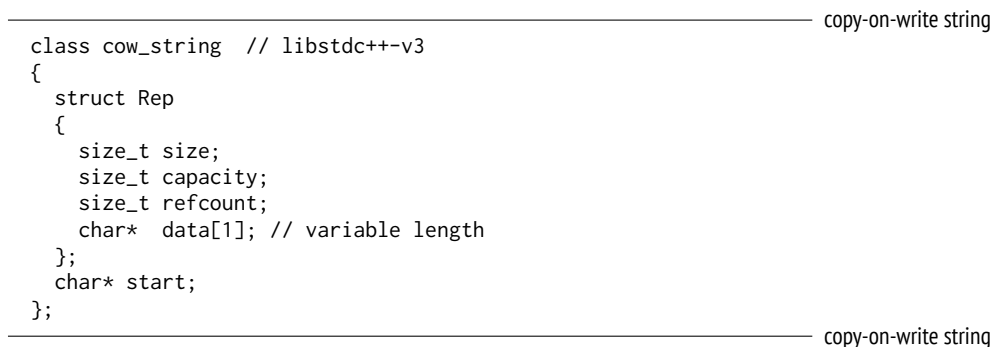


图 12-3

新的 `string` 结构在 64-bit 中是 16 字节，比原来的 24 字节小了一些。

12.7.2 写时复制（copy-on-write）

`string` 对象里只放一个指针，如图 12-4 所示。值得一提的是 COW 对多线程不友好，Andrei Alexandrescu 提倡在多核时代应该改用 eager copy string。^[Alex10]



²⁸ 如果真的用到了，就继续使用 `std::string` 或 `std::vector<char>` 好了。

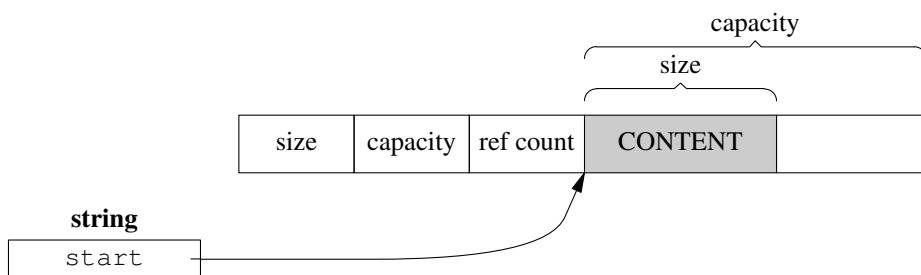


图 12-4

这种数据结构没啥好说的，在 64-bit 中似乎也没有优化空间。另外 COW 的操作复杂度不一定符合直觉，它拷贝字符串是 $O(1)$ 时间，但是拷贝之后的第一次 `operator[]` 有可能是 $O(N)$ 时间。²⁹

12.7.3 短字符串优化（SSO）

`string` 对象比前面两个都大，因为有本地缓冲区（local buffer）。

```
class sso_string // __gnu_ext::__sso_string
{
    char* start;
    size_t size;
    static const int kLocalSize = 15;
    union
    {
        char buffer[kLocalSize+1];
        size_t capacity;
    } data;
};
```

short-string-optimized string

short-string-optimized string

内存布局如图 12-5（左图）所示。如果字符串比较短（通常的阈值是 15 字节），那么直接存放在对象的 `buffer` 里，如图 12-5（右图）所示。`start` 指向 `data.buffer`。

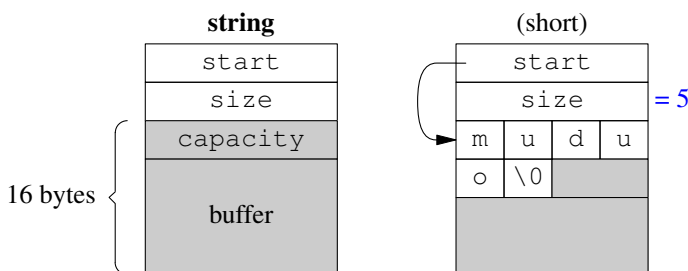


图 12-5

²⁹ <http://coolshell.cn/articles/1443.html>

如果字符串超过 15 字节，那么就变成类似图 12-2 的 eager copy 2 结构，start 指向堆上分配的空间（见图 12-6）。

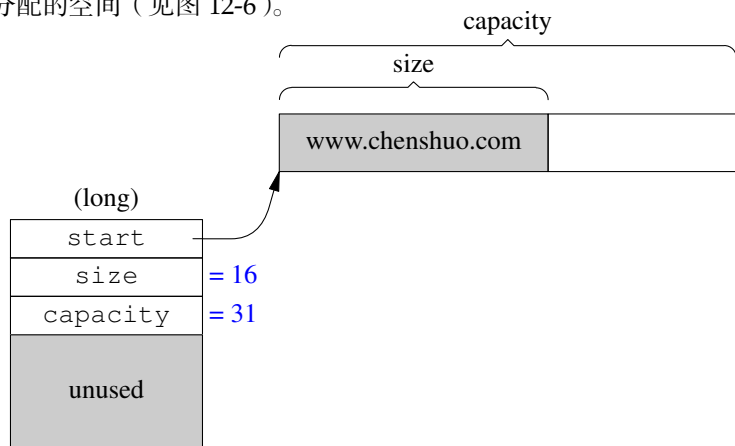


图 12-6

短字符串优化的实现方式不止一种，主要区别是把那三个指针/整数中的哪一个与本地缓冲重合。例如《Effective STL》[ESTL] 第 15 条展现的“实现 D”是将 buffer 与 start 指针重合，这正是 Visual C++ 的做法。而 STLPort 的 string 是将 buffer 与 end_of_storage 指针重合。

SSO string 在 64-bit 中有一个小小的优化空间：如果允许字符串 max_size() 不大于 4GiB 的话，我们可以用 32-bit 整数来表示长度和容量，这样同样是 32 字节的 string 对象，local buffer 可以增大至 19 字节。

```

class sso_string // optimized for 64-bit
{
    char* start;
    uint32_t size;

    static const int kLocalSize = sizeof(void*) == 8 ? 19 : 15;

    union
    {
        char buffer[kLocalSize+1];
        uint32_t capacity;
    } data;
};

```

short-string-optimized string 2

内存布局如图 12-7 所示。

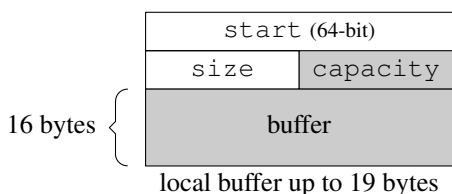


图 12-7

llvm/clang/libc++ 采用了与众不同的 SSO 实现，空间利用率最高。其 local buffer 几乎与三个指针/整数完全重合，在 64-bit 上对象大小是 24 字节，本地缓冲区可达 22 字节。数据结构如图 12-8 所示。

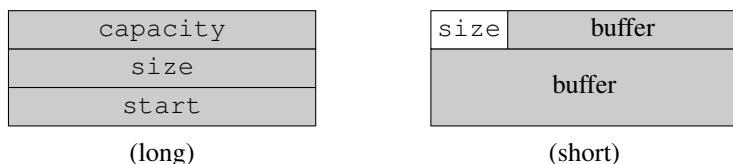


图 12-8

它用一个 bit 来区分是长字符串还是短字符串，然后用位操作和掩码（mask）来取重叠部分的数据，因此实现是 SSO 里最复杂的³⁰，如图 12-9 所示。

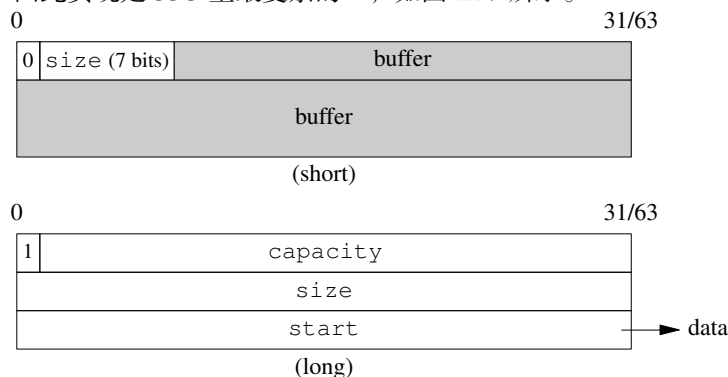


图 12-9

Andrei Alexandrescu 建议^[Alex10] 针对不同的应用负载选用不同的 string，对于短字符串，用 SSO string；对于中等长度的字符串，用 eager copy；对于长字符串，用 COW。具体分界点需要靠 profiling 来确定，选用合适的字符串可能提高 10% 的整体性能。

从实现的复杂度上看，eager copy 是最简单的，SSO 稍微复杂一些，COW 最难。性能也各有千秋，见 Petr Ovtchenkov 写的《Comparison of Strings Implementations

³⁰ <http://llvm.org/viewvc/llvm-project/libcxx/trunk/include/string?view=markup>

in C++ language》³¹。我准备自己写一个 non-standard³² non-template³³ 的 string 库（位于 recipes/string）作为练手，计划采用 eager copy 3 和 sso 2 的数据结构。

注：C++03/98 标准没有规定 string 中的字符是连续存储的，但是《Generic Programming and the STL》的作者 Matthew Austern 指出：现在所有的 std::string 实现都是连续存储的，因此建议在新标准中明确规定下来³⁴。

12.8 用 STL algorithm 轻松解决几道算法面试题

C++ STL 的 algorithm 配合自定义的 functor（仿函数、函数对象）可以轻松解决不少面试题，代码简洁，正确性也容易验证。本节仍旧采用 C++03 的 functor 写法，没有采用 C++11 的 Lambda 表达式写法，尽管后者会简洁得多。完整代码及测试用例见 recipes/algorithm。

12.8.1 用 next_permutation() 生成排列与组合

本小节的内容源自 10 年前我写的一篇博客³⁵，这篇博客还找到了 Visual C++ 7.0 的 STL 的一个疑似 bug（或者叫 feature）。生成排列、组合、整数划分的具体算法见 Donald Knuth 的《The Art of Computer Programming, Volume 4A》³⁶ 第 7.2.1 节。本处只给出使用 STL 的实现代码。

生成 N 个不同元素的全排列

这是 next_permutation() 的基本用法，把元素从小到大放好（即字典序最小的排列），然后反复调用 next_permutation() 就行了。

```
6 int main()
7 {
8     int elements[] = { 1, 2, 3, 4 };
9     const size_t N = sizeof(elements)/sizeof(elements[0]);
10    std::vector<int> vec(elements, elements + N);
```

recipes/algorithm/permutation.cc

³¹ <http://complement.sourceforge.net/compare.pdf>

³² C++ 标准库的 string 有很多设计缺陷，见 Herb Sutter 的《Exceptional C++ Style》第 37~40 条。

³³ 见 Steve Donovan 写的《Overdoing C++ Templates》（<http://blog.csdn.net/myan/article/details/1915>）。

³⁴ <http://www.open-std.org/JTC1/SC22/WG21/docs/lwg-defects.html#530>

³⁵ <http://blog.csdn.net/Solstice/article/details/2059>

³⁶ <http://cs.utsa.edu/~wagner/knuth/>

```

11
12     int count = 0;
13     do
14     {
15         std::cout << ++count << ": ";
16         std::copy(vec.begin(), vec.end(),
17                 std::ostream_iterator<int>(std::cout, ", "));
18         std::cout << std::endl;
19     } while (next_permutation(vec.begin(), vec.end()));
20 }

```

recipes/algorithm/permutation.cc

整个程序最关键的就是 L19。输出的前几行如下：

```

1: 1, 2, 3, 4,
2: 1, 2, 4, 3,
3: 1, 3, 2, 4,
4: 1, 3, 4, 2,
5: 1, 4, 2, 3,
6: 1, 4, 3, 2,
7: 2, 1, 3, 4,
8: 2, 1, 4, 3,
9: 2, 3, 1, 4,
// 一共 24 行

```

类似的代码还能生成多重排列，比如 2 个 a、3 个 b 的全部排列，代码见 permutation2.cc。输出如下：

```

1: a, a, b, b, b,
2: a, b, a, b, b,
3: a, b, b, a, b,
4: a, b, b, b, a,
5: b, a, a, b, b,
6: b, a, b, a, b,
7: b, a, b, b, a,
8: b, b, a, a, b,
9: b, b, a, b, a,
10: b, b, b, a, a,

```

注： $\frac{5!}{2! \times 3!} = 10$

思考：能不能把 do {} while () 循环换成 while () {} 循环？

生成从 N 个元素中取出 M 个的所有组合

题目： 输出从 7 个不同元素中取出 3 个元素的所有组合。

思路： 对序列 {1, 1, 1, 0, 0, 0, 0} 做全排列。对于每个排列，输出数字 1 对应的位置上的元素。代码如下：

Linux 多线程服务端编程：使用 muduo C++ 网络库

```

7  int main()
8  {
9      int values[] = { 1, 2, 3, 4, 5, 6, 7 };
10     int elements[] = { 1, 1, 1, 0, 0, 0, 0 };
11     const size_t N = sizeof(elements)/sizeof(elements[0]);
12     assert(N == sizeof(values)/sizeof(values[0]));
13     std::vector<int> selectors(elements, elements + N);
14
15     int count = 0;
16     do
17     {
18         std::cout << ++count << ": ";
19         for (size_t i = 0; i < selectors.size(); ++i)
20         {
21             if (selectors[i])
22             {
23                 std::cout << values[i] << ", ";
24             }
25         }
26         std::cout << std::endl;
27     } while (prev_permutation(selectors.begin(), selectors.end()));
28 }

```

recipes/algorithm/combinations.cc

注意，为了照顾输出顺序，L27 用的是 `prev_permutation()`。程序输出如下：

```

1: 1, 2, 3,
2: 1, 2, 4,
3: 1, 2, 5,
4: 1, 2, 6,
// 省略若干行
33: 4, 5, 7,
34: 4, 6, 7,
35: 5, 6, 7,

```

可见完整地输出了 $C(7,4) = 35$ 种组合。

12.8.2 用 `unique()` 去除连续重复空白

孟岩在谈《C++ 程序设计原理与实践》³⁷ 时曾说：“比如对我来说，C++ 这个语言最强的地方在于它的模板技术提供了足够复杂的程序库开发机制，可以把复杂性高度集中在程序库里。做得好的话，在应用代码部分我连一个 `for` 循环都不用写，犯错误的机会就少，效率还不打折扣，关键是看着代码心里爽。”这几小节可算是他这番话的一个注脚。C++11 有了 Lambda 表达式，Scott Meyers 提倡的“Prefer algorithm calls to hand-written loops”就更容易落实了³⁸。

³⁷ <http://blog.csdn.net/hzbooks/article/details/5767169>

³⁸ <http://drdobbs.com/184401446>

题目 给你一个字符串，要求原地（in-place）把相邻的多个空格替换为一个³⁹。例如，输入 "a_b"，输出 "a_b"；输入 "aaa_bbb_"，输出 "aaa_bbb_"。

这道题目不难，手写的话也就是单重循环，复杂度是 $O(N)$ 时间和 $O(1)$ 空间。这里展示用 `std::unique()` 的解法，思路很简单：`std::unique()` 的作用是去除相邻的重复元素，我们只要把“重复元素”定义为“两个元素都是空格”即可。注意所有针对区间的 STL algorithm 都只能调换区间内元素的顺序，不能真正删除容器内的元素，因此需要 L17。关键代码如下：

```

5 struct AreBothSpaces
6 {
7     bool operator()(char x, char y) const
8     {
9         return x == ' ' && y == ' ';
10    }
11 };
12
13 void removeContinuousSpaces(std::string& str)
14 {
15     std::string::iterator last
16         = std::unique(str.begin(), str.end(), AreBothSpaces());
17     str.erase(last, str.end());
18 }

```

recipes/algorithm/removeContinuousSpaces.cc

recipes/algorithm/removeContinuousSpaces.cc

12.8.3 用 {make, push, pop}_heap() 实现多路归并

题目 用一台 4GiB 内存的机器对磁盘上的单个 100GB 文件排序。⁴⁰

这种单机外部排序题目的标准思路是先分块排序，然后多路归并成输出文件。多路归并很容易用 heap 排序实现，比方说要归并已经按从小到大的顺序排好序的 32 个文件，我们可以构造一个 32 元素的 min heap，每个元素是 `std::pair<Record, FILE*>`。然后每次取出堆顶的元素，将其 Record 写入输出文件；如果 FILE* 还可读，就读入一条 Record，再向 heap 中添加 `std::pair<Record, FILE*>`。这样当 heap 为空的时候，多路归并就完成了。注意在这个过程中 heap 的大小通常会慢慢变小，因为有可能某个输入文件已经全部读完了。

这种方法比传统的二路归并要节省很多遍磁盘读写，假如用教科书上的二路归并来做外部排序⁴¹，那么我们要先读一遍这 32 个文件，两两归并输出 16 个稍大的已

³⁹ 来自 <https://gist.github.com/2227226>。

⁴⁰ 题目改编自 <http://blog.csdn.net/pennyliang/article/details/7073777>。

⁴¹ 这种教科书有可能是在大型机还在使用磁带外存的时候写成的。

排序中间文件；然后再读一遍这 16 个中间文件，两两归并输出 8 个更大的中间文件；如此往复，最后归并两个已经排好序的大文件，输出最终的结果。读者可以算算这比直接多路归并要多读写多少遍磁盘。

完整的外部排序代码见 `recipes/esort/sort02.cc` 及其改进版 `sort{03,04}.cc`。这里展示一个内存里的多路归并，以说明基本思路。

```

39  File mergeN(const std::vector<File>& files)
40  {
41      File output;
42      std::vector<Input> inputs;
43
44      for (size_t i = 0; i < files.size(); ++i) {
45          Input input(&files[i]);
46          if (input.next()) {
47              inputs.push_back(input);
48          }
49      }
50
51      std::make_heap(inputs.begin(), inputs.end());
52      while (!inputs.empty()) {
53          std::pop_heap(inputs.begin(), inputs.end());
54          output.push_back(inputs.back().value);
55
56          if (inputs.back().next()) {
57              std::push_heap(inputs.begin(), inputs.end());
58          } else {
59              inputs.pop_back();
60          }
61      }
62
63      return output;
64  }

```

recipes/algorithm/mergeN.cc

`L44~L51` 构造一个 binary heap，`L52` 开始的 while 循环反复取出堆顶元素（`L53` `std::pop_heap()` 会把堆顶元素放到序列末尾，即 `inputs.back()` 处），`L54` 把取出的元素（当前最小值）输出。`L56~L60` 从堆顶元素所属的文件读入下一条记录，如果成功，就把它放回堆中（`L57`）。当循环结束的时候，堆为空，说明每个文件都读完了。其中用到的 `Input` 类型定义如下。

```

typedef int Record;
typedef std::vector<Record> File;

struct Input
{
    Record value;
    const File* file;

```

recipes/algorithm/mergeN.cc

```

explicit Input(const File* f);
bool next();

bool operator<(const Input& rhs) const
{
    // make_heap to build min-heap, for merging
    return value > rhs.value;
}
};

```

recipes/algorithm/mergeN.cc

以上是多路归并的实现，再来考虑第一阶段分块排序的流水线设计。先做一个简化的假设：普通机械硬盘的顺序读写速度是 100MB/s，既然可用内存为 4GB，那么分块（chunk）的大小就选定为 1GB，这样读入和写出一个分块均耗时 10 秒。再假设在内存中排序 1GB 数据耗时 10 秒。为了编程方便，磁盘 IO 用阻塞方式。按照这些假设，如果用单线程的方式实现外部排序，第一阶段的耗时是 $30N$ 秒，其中 N 是分块数目。对一个 6GB 的文件排序，单线程程序（sort02.cc）的执行过程如图 12-10 所示，第一阶段将耗时 180 秒（只画出前 120 秒）。内存消耗为 1GB。

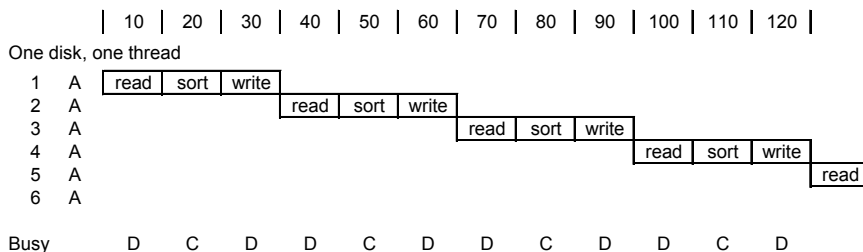


图 12-10

注意到，在程序执行时，要么 CPU 繁忙，要么硬盘繁忙（Busy 行的 D 表示磁盘，C 表示 CPU），资源并没有充分利用起来。为了加快排序速度，我们考虑用多线程，让计算和 IO 重叠，减少整体运行时间。注意这里我们不能简单地起多个进程，每个进程分别排序一个 chunk，因为这样势必会造成多个进程争抢磁盘 IO，而机械硬盘的随机读取比顺序读取慢得多。

一种解决办法是把 IO 放入一个单独的线程，避免争抢，然后用另外的线程(s)来排序内存中的数据块。换句话说，一个线程做 IO（由于只有一块硬盘，那么不必使用多个 IO 线程），再用一个线程池做计算，以实现 IO 和计算重叠。我们预计这种方式完成分块排序将会耗时 120 秒，比单线程快 33%。预计执行流程（流水线）如图 12-11 所示。

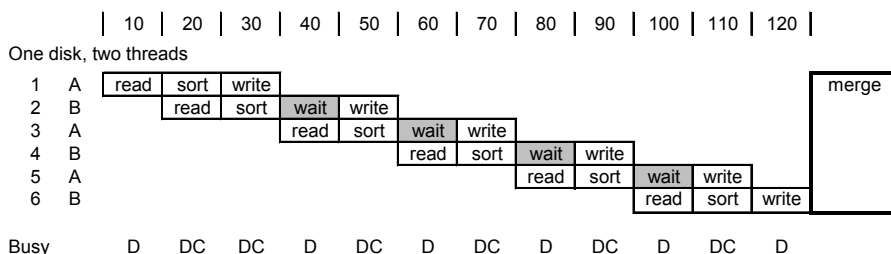


图 12-11

注意同一时刻磁盘要么顺序读，要么顺序写，避免反复寻道的开销。这种方案会让 CPU 和磁盘同时繁忙，提高了资源利用率，内存消耗为 2GB。这种思路的代码见 [sort03.cc](#)。图 12-12 是一次实际运行的情况，方块的宽度与时间成正比。这里实际的磁盘和 CPU 的速度比前面的假设要快，因此第一阶段总耗时 90 秒。

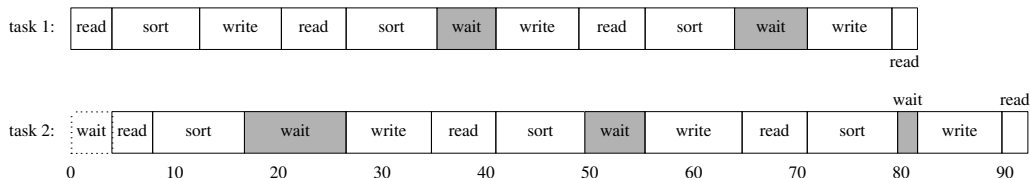


图 12-12

注意到 CPU 的吞吐量（每秒排序 100MB 数据）大于单块磁盘吞吐量（读写 100MB 共耗时 2 秒），因此仍然会出现 CPU 等待 IO 的情况。如果有不止一块磁盘，可以重新设计流水线，进一步压缩运行时间。比方说把输入数据全部放在 S 盘（source），把分块排序的中间结果放到 T 盘（temporary），这样两块磁盘一读一写，可以相互重叠。在归并阶段，自然可以从 T 盘读数据写到 S 盘。这需要用到两个 IO 线程，每个磁盘配一个 IO 线程，确保每个磁盘都是顺序访问的，以保证吞吐量。这种方案的分块排序预计用时 80 秒，预计执行流程如图 12-13 所示，比第一种快 50% 以上，内存消耗也增长到 3GB。（这种方案的实现留作练习。）

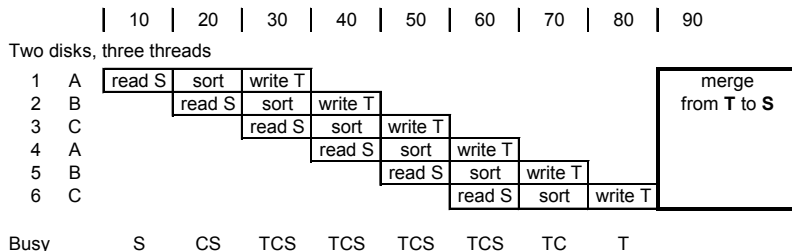


图 12-13

还有一个简单的优化措施：最后的两三个排序结果不必写入磁盘，而是直接在内存中参与多路归并，这样大约可以再节约 10 秒。

类似的题目：有 a 、 b 两个文件，大小各是 100GB 左右，每行长度不超过 1kB，这两个文件有少量（几百个）重复的行，要求用一台 4GiB 内存的机器找出这些重复行。

解这道题目有两个方向，一是 hash，把 a 、 b 两个文件按行的 hash 取模分成几百个小文件，每个小文件都在 1GB 以内，然后对 a_1 、 b_1 求交集 c_1 ，对 a_2 、 b_2 求交集 c_2 ，这样就能在内存里解决了。

第二个思路是外部排序，但是跟前面完整的外部排序不同，我们并不需要得到两个已排序的文件（ a' 和 b' ）再求交集，只需要把 a 分块排序成 100 个小文件，再把 b 分块排序成 100 个小文件。剩下的工作就是一边读这些小文件，一边在内存中同时归并出 a' 和 b' ，一边求出交集。内存中的两个多路归并需要两个 heap，分别对应 a 和 b 的小文件 (s)。内存中的运算流程如图 12-14 所示。

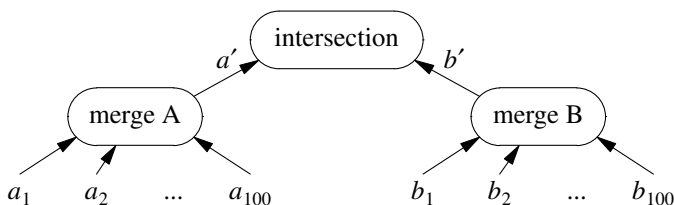


图 12-14

代码写起来估计比单个 heap 归并要复杂一些，特别是 C++ 不支持类似 C# 的 yield 关键字来方便地实现迭代。假如 C++ 有 yield，那么“求交集”这一步我们直接调用 `std::set_intersection()` 并配合适当的迭代器就行了，但是在没有 yield 的情况下要实现这样的迭代器恐怕要费事得多，因为每个迭代器要维护更多的状态。这算是 coroutine 的一个使用场景。

上面两种解法的代价都是额外 200GB 磁盘空间，请读者思考有没有大大节省磁盘空间的做法。另外一个延伸的题目是：有几个巨大的文本文件，每行存放一个查询 (query)，将所有 query 按出现次数排序（代码 <https://gist.github.com/4009225>）。

12.8.4 用 partition() 实现“重排数组，让奇数位于偶数前面”

`std::partition()` 的作用是把符合条件的元素放到区间首部，不符合条件的元素放到区间后部，我们只需把“符合条件”定义为“元素是奇数”就能解决这道题。复杂度是 $O(N)$ 时间和 $O(1)$ 空间。为节省篇幅，`isOdd()` 直接做成了函数，而不是函数对象，缺点是有可能阻碍编译器实施 inlining。

```

5  bool isOdd(int x)
6  {
7      return x % 2 != 0; // x % 2 == 1 is WRONG
8  }
9
10 void moveOddsBeforeEvens()
11 {
12     int oddeven[] = { 1, 2, 3, 4, 5, 6 };
13     std::partition(oddeven, oddeven+6, &isOdd);
14     std::copy(oddeven, oddeven+6, std::ostream_iterator<int>(std::cout, ", "));
15     std::cout << std::endl;
16 }

```

输出如下，注意确实满足“奇数位于偶数之前”，但奇数元素之间的相对位置有变化，偶数元素亦是如此。

1, 5, 3, 4, 2, 6,

如果题目要求改成“调整数组顺序使奇数位于偶数前面，并且保持奇数的先后顺序不变，偶数的先后顺序不变”，解决办法也一样简单，改用 `std::stable_partition()` 即可，代码及输出如下：

```

int oddeven[] = { 1, 2, 3, 4, 5, 6 };
std::stable_partition(oddeven, oddeven+6, &isOdd);
std::copy(oddeven, oddeven+6, std::ostream_iterator<int>(std::cout, ", "));
std::cout << std::endl;
// 输出 1, 3, 5, 2, 4, 6,

```

注意，`stable_partition()` 的复杂度较特殊：在内存充足的情况下，开辟与原数组一样大的空间，复杂度是 $O(N)$ 时间和 $O(N)$ 空间；在内存不足的情况下，要做 in-place 位置调换，复杂度是 $O(N \log N)$ 时间和 $O(1)$ 空间。

类似的题目还有“调整数组顺序使负数位于非负数前面”，读者应能举一反三。

12.8.5 用 `lower_bound()` 查找 IP 地址所属的城市

题目 已知 N 个 IP 地址区间和它们对应的城市名称，写一个程序，能从 IP 地址找到它所在的城市。注意这些 IP 地址区间互不重叠。

这道题目的 naïve 解法是 $O(N)$ ，借助 `std::lower_bound()` 可以轻易做到 $O(\log N)$ 查找，代价是事先做一遍 $O(N \log N)$ 的排序。如果区间相对固定而查找很频繁，这么做是值得的。

基本思路是按 IP 区间的首地址排好序，再进行二分查找。比如说有两个区间 [300, 500]、[600, 750]，分别对应北京和香港两个城市，那么 `std::lower_bound()` 查找 299、300、301、499、500、501、599、600、601、749、750、751 等“IP 地址”返回的迭代器如图 12-15 所示。

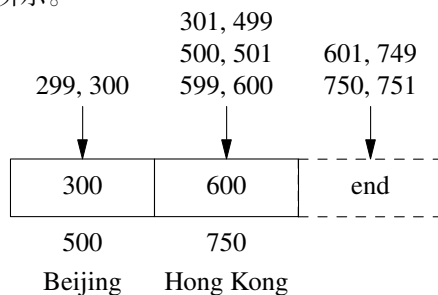


图 12-15

我们需要对返回的结果微调（L28~L32），使得迭代器 `it` 所指的区间是唯一有可能包含该 IP 地址的区间，如图 12-16 所示。

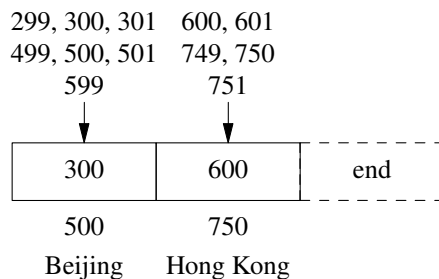


图 12-16

最后判断一下 IP 地址是否位于这个区间就行了（L34）。完整代码如下，为了简化，“城市”用整数表示，-1 表示未找到。另外，这个实现对于整个 IP 地址空间都是正确的，即便区间中包括 [255.255.255.0, 255.255.255.255] 这种边界条件。

```

7 struct IPRange
8 {
9     uint32_t startIp; // inclusive
10    uint32_t endIp;    // inclusive
11    int value;         // >= 0
12
13    bool operator<(const IPRange& rhs) const
14    {
15        return startIp < rhs.startIp;
16    }
17 };
18
```

recipes/algorithm/iprange.cc

```

19 // REQUIRE: ranges is sorted.
20 int findIpValue(const std::vector<IPrange>& ranges, uint32_t ip)
21 {
22     int result = -1;
23
24     if (!ranges.empty()) {
25         IPrange needle = { ip, 0, 0 };
26         std::vector<IPrange>::const_iterator it
27             = std::lower_bound(ranges.begin(), ranges.end(), needle);
28         if (it == ranges.end()) {
29             --it;
30         } else if (it != ranges.begin() && it->startIp > ip) {
31             --it;
32         }
33
34         if (it->startIp <= ip && it->endIp >= ip) {
35             result = it->value;
36         }
37     }
38     return result;
39 }

```

recipes/algorithm/iprange.cc

说明：如果 IP 地址区间有重复，那么我们通常要用线段树⁴²来实现高效的查询。另外，在真实的场景中，IP 地址区间通常适用专门的 longest prefix match 算法，这会比本节的通用算法更快。

小结

想到正确的思路是一码事，写出正确的、经得起推敲的代码是另一码事。例如 §12.8.4 用 $(x \% 2 != 0)$ 来判断 int x 是否为奇数，如果写成 $(x \% 2 == 1)$ 就是错的，因为 x 可能是负数，负数的取模运算的关窍见 §12.3。常见的错误还包括误用 char 的值作为数组下标（面试题目：统计文件中每个字符出现的次数），但是没有考虑 char 可能是负数，造成访问越界。有的人考虑到了 char 可能是负数，因此先强制转型为 unsigned int 再用作下标，这仍然是错的。正确的做法是强制转型为 unsigned char 再用作下标，这涉及 C/C++ 整型提升的规则，就不详述了。这些细节往往是面试官的考察点⁴³。本节给出的解法在正确性方面应该是没问题的；在效率方面，可以说在 Big-O 意义下是最优的，但不一定是运行最快的。

另外，面试题的目的可能就是让你动手实现一些 STL 算法，例如求两个有序集合的交集（set_intersection()）、洗牌（random_shuffle()）等等，这就不属于本

⁴² http://en.wikipedia.org/wiki/Segment_tree

⁴³ 工作 5 年以来，我面试过近百人，因此这番话是从面试官的角度说的。

节所讨论的范围了。从“算法”本身的难度上看，我个人把 STL algorithm 分为三类，面试时要求手写的往往是第二类算法。

- 容易，即闭着眼睛一想就知道是如何实现的，自己手写一遍的难度跟 `strlen()` 和 `strcpy()` 差不多。这类算法基本上就是遍历一遍输入区间，对每个元素做些判断或操作，一个 `for` 循环就解决问题。一半左右的 STL algorithm 属于此类，例如 `for_each()`、`transform()`、`accumulate()` 等等。
- 较难，知道思路，但是要写出正确的实现要考虑清楚各种边界条件。例如 `merge()`、`unique()`、`remove()`、`random_shuffle()`⁴⁴、`lower_bound()`、`partition()` 等等，三成左右的 STL algorithm 属于此类。
- 难，要在一个小时内写出正确的、健壮的实现基本不现实，例如 `sort()`⁴⁵、`nth_element()`、`next_permutation()`、`inplace_merge()` 等等，约有两成 STL algorithm 属于此类。

注意，“容易”级别的算法是指写出正确的实现很容易，但不一定意味着写出高效的实现也同样容易，例如 `std::copy()` 拷贝 POD 类型的效率可媲美 `memcpy()`，这需要一点模板技巧。

以上分类纯属个人主观看法，或许别人有不同的分类法，例如把 `remove()` 归入简单，把 `next_permutation()` 归入较难，把 `lower_bound()` 归入难等。

⁴⁴ 要考虑随机数生成器的状态空间（http://en.wikipedia.org/wiki/Fisher-Yates_shuffle#Potential_sources_of_bias）。

⁴⁵ 快速排序是本科生数据结构课上就有的内容，但是其工业强度的实现是足以在顶级期刊上发论文的。

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 Stevens 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 8000 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。

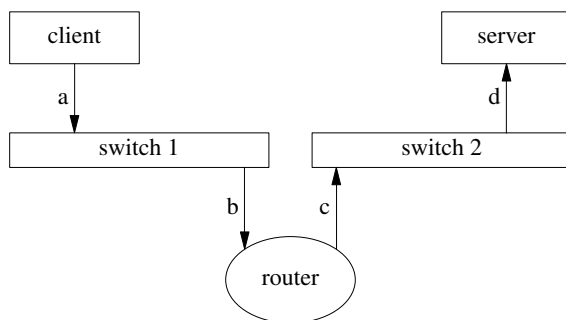


图 A-1

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://noahdavidson.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，

Linux 多线程服务端编程：使用 muduo C++ 网络库

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_CPU_Caches.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 TR1²⁸ 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。

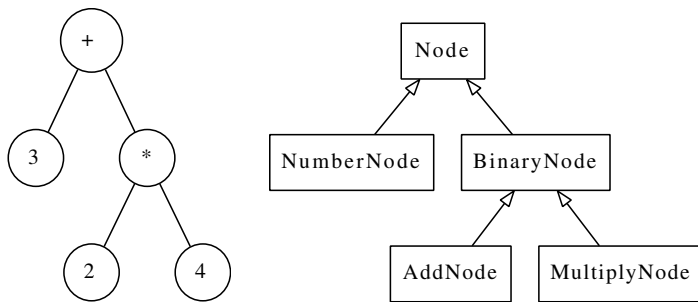


图 B-1

³³ C++ 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。

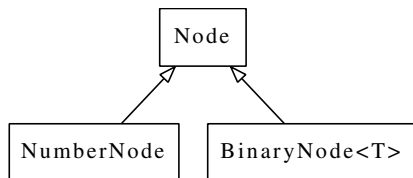


图 B-2

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++ 程序中完全可以完全不写 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/qiuzu/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++11⁵⁷，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 soXXXXXX 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什么事情？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，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。

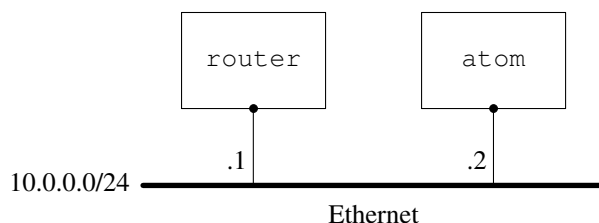


图 D-1

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。

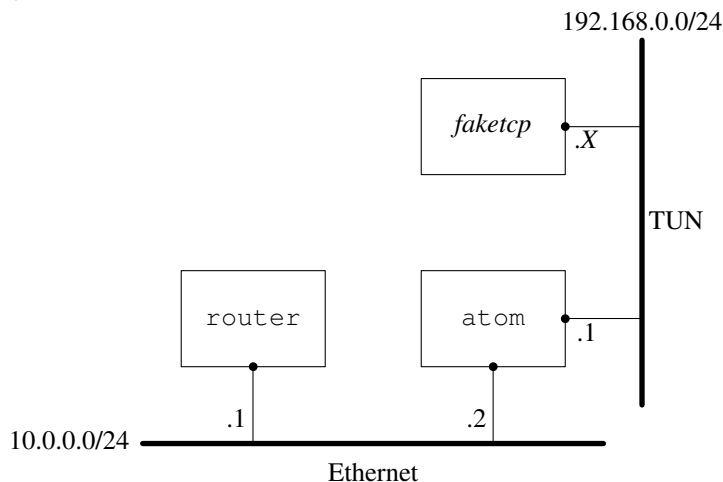


图 D-2

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 faketcip 了。代码见 recipes/faketcip/icmpecho.cc。

其中响应 ICMP echo request 的函数是 icmp_input()，位于 recipes/faketcip/faketcip.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

```
allocated 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/faketcip/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 faketcip 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 连接都被拒绝了。

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/faketcip/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 faketcip 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/faketcip/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 `faketcip` 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/faketcip/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


```
allocated 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. 基峰出版社, 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. Kernighan 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