多线程服务器的常用编程模型

陈硕 (giantchen_AT_gmail)
Blog.csdn.net/Solstice
2009 Feb 12

本文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，归纳了进程间通讯与线程同步的最佳实践，以期用简单规范的方式开发多线程程序。

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 x64 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），机器之间用百兆或千兆以太网连接。这大概是目前民用 PC 服务器的主流配置。

本文不涉及 Windows 系统，不涉及人机交互界面（无论命令行或图形）；不考虑文件读写（往磁盘写 log 除外），不考虑数据库操作，不考虑 Web 应用；不考虑低端的单核主机或嵌入式系统，不考虑手持式设备，不考虑专门的网络设备，不考虑高端的 >=32 核 Unix 主机；只考虑 TCP，不考虑 UDP，也不考虑除了局域网络之外的其他数据收发方式（例如串并口、USB口、数据采集板卡、实时控制等）。

有了以上这么多限制，那么我将要谈的“网络应用程序”的基本功能可以归纳为“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，我放到另一篇博客里讨论。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，本文不考虑虚拟化的场景，当我说“两个进程不在同一台机器上"，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。

本文假定读者已经有多线程编程的知识与经验，这不是一篇入门教程。

本文承蒙 Milo Yip 先生审读，在此深表谢意。当然，文中任何错误责任均在我。

目 录

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”。本文的进程指的是 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。

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 书把“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死否死了（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（管登陆的、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 a 的活儿挪给 b 做）、**退休**（a 要修复 bug，先别给他派新活儿，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，距今不过十余年，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；signal 的语意也大为复杂化。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，我认为是为了更好地发挥对称多路处理 (SMP) 的效能。在 SMP 之前，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。）如果只有一个执行单元，一个 CPU，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。

2 典型的单线程服务器编程模型

UNP3e 对此有很好的总结（第 6 章：IO 模型，第 30 章：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，我知道的有：

- lighttpd，单线程服务器。（nginx 估计与之类似，待查）
- libevent/libev
- ACE，Poco C++ libraries（QT 待查）
- Java NIO (Selector/SelectableChannel), Apache Mina, Netty (Java)
- POE (Perl)
- Twisted (Python)

相反，boost::asio 和 Windows I/O Completion Ports 实现了 Proactor 模式，应用面似乎要窄一些。当然，ACE 也实现了 Proactor 模式，不表。

在“non-blocking IO + IO multiplexing”这种模型下，程序的基本结构是一个事件循环 (event loop)：（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
  } else {
    处理到期的 timers
    if (retval > 0) {
      处理 IO 事件
    }
  }
}
```

当然，select(2)/poll(2) 有很多不足，Linux 下可替换为 epoll，其他操作系统也有对应的高性能替代品（搜 c10k problem）。

Reactor 模型的优点很明显，编程简单，效率也不错。不仅网络读写可以用，连接的建立（connect/accept）甚至 DNS 解析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 即是这样，它内部的 fdevent 结构十分精妙，值得学习。（这里且不考虑用阻塞 IO 这种次优的方案。）

当然，实现一个优质的 Reactor 不是那么容易，我也没有用过坊间开源的库，这里就不推荐了。

3 典型的多线程服务器的线程模型

这方面我能找到的文献不多，大概有这么几种：

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 1 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。

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

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），代码框架跟第 2 节一样。

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。
  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 (TCP connection) 注册到那个线程的 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线程中；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 Reactor 上，程序里有多个 Reactor，每个线程至多有一个 Reactor。

多线程程序对 Reactor 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，这个 loop 必须得是线程安全的。

线程池

不过，对于没有 IO 光有计算任务的线程，使用 event loop 有点浪费，我会用有一种补充方案，即用 blocking queue 实现的任务队列(TaskQueue)：

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

void worker_thread()
{
  while (!quit) {
    boost::function<void()> 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：

```
启动容量为 N 的线程池：
int N = num_of_computing_threads;
for (int i = 0; i < N; ++i) {
  create_thread(&worker_thread);  // 伪代码：启动线程
}
```

使用起来也很简单：

```
boost::function<void()> task = boost::bind(&Foo::calc, this);
taskQueue.post(task);
```

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 5 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，我的另一篇博客《当析构函数遇到多线程——C++ 中线程安全的对象回调》详细讨论了这个问题 http://blog.csdn.net/Solstice/archive/2010/01/22/5238671.aspx

除了任务队列，还可以用 blocking_queue<T> 实现数据的消费者-生产者队列，即 T 的是数据类型而非函数对象，queue 的消费者(s)从中拿到数据进行处理。这样做比 task queue 更加 specific 一些。

blocking_queue<T> 是多线程编程的利器，它的实现可参照 Java 5 util.concurrent 里的(Array|Linked)BlockingQueue，通常 C++ 可以用 deque 来做底层的容器。Java 5 里的代码可读性很高，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。（我没有用过免费的库，这里就不乱推荐了，有兴趣的同学可以试试 Intel Threading Building Blocks 里的 concurrent_queue<T>。）

归纳

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

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

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，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。

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，具体的考虑点容我以后再谈。

这里没有谈线程的退出，留待下一篇 blog“多线程编程反模式”探讨。

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。

4 进程间通信与线程间通信

Linux 下进程间通信 (IPC) 的方式数不胜数，光 UNPv2 列出的就有：pipe、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，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。

5 进程间通信

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（比如共享内存效率最高，但再怎么着也不能高效地共享两台机器的内存），限制了 scalability。

在编程上，TCP sockets 和 pipe 都是一个文件描述符，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，pipe 是单向的 (Linux)，进程间双向通讯还得开两个文件描述符，不方便；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讯模型下，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/Selector 时用来异步唤醒 select (或等价的 poll/epoll) 调用（Sun JVM 在 Linux 就是这么做的）。

TCP port 是由一个进程独占，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，自然就没法工作了），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，这样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，我以后在讲服务端的日期与时间处理的时候还会谈到心跳协议的设计。

与其他 IPC 相比，TCP 协议的一个自然好处是“可记录可重现”，tcpdump/Wireshark 是解决两个进程间协议/状态争端的好帮手。

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，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。这将是我下一篇 blog 的主题，目前我推荐 Google Protocol Buffers。

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。我问，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？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”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。为我所不取。再说了，你舍得让几万块买来的 SQL Server 和你的程序分享机器资源吗？产品里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，还可以在此之上构建 RPC/REST/SOAP 之类的上层通信协议，这超过了本文的范围。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。

本文不具体讲 Reactor 方式下的网络编程，其实这里边有很多值得注意的地方，比如带 back off 的 retry connecting，用优先队列来组织 timer 等等，留作以后分析吧。

6 线程间同步

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，偶尔用一用读写锁；
4. 不自己编写 lock-free 代码，不去凭空猜测“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。

互斥器 (mutex)

互斥器 (mutex) 恐怕是使用得最多的同步原语，粗略地说，它保护了临界区，一个时刻最多只能有一个线程在临界区内活动。（请注意，我谈的是 pthreads 里的 mutex，不是 Windows 里的重量级跨进程 Mutex。）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。
- 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）。这样我们保证在同一个函数里加锁和解锁，避免在 foo() 里加锁，然后跑到 bar() 里解锁。
- 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

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，不会因异常而忘记解锁。

Mutex 恐怕是最简单的同步原语，安照上面的几条原则，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招到并修复。

跑题：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，它们的惟一区别在于：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 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。具体的例子：

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

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

void traverse()
{
  MutexLockGuard lock(mutex);
  for (auto it = foos.begin(); it != foos.end(); ++it) { // 用了 0x 新写法
    it->doit();
  }
}
```

post() 加锁，然后修改 foos 对象； traverse() 加锁，然后遍历 foos 数组。将来有一天，Foo::doit() 间接调用了 post() （这在逻辑上是错误的），那么会很有戏剧性的：

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（(gdb) thread apply all bt），只要每个函数不是特别长，很容易看出来是怎么死的。（另一方面支持了函数不要写过长。）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，让验尸官的日子好过些。

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。

对于 a)，仿造前面的办法能比较容易地排错。对于 b)，如果 pthreads 提供 isLocked() 就好办，可以写成：

```
void postWithLockHold(const Foo& f)
{
  assert(mutex.isLocked());  // 目前只是一个愿望
  // ...
}
```

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。

=== 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很多 race condition，请参考拙作《当析构函数遇到多线程——C++ 中线程安全的对象回调》

http://blog.csdn.net/Solstice/archive/2010/01/22/5238671.aspx 。请注意这里的 class 命名与那篇文章有所不同。我现在认为 MutexLock 和 MutexLockGuard 是更好的名称。

性能注脚：Linux 的 pthreads mutex 采用 futex 实现，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。

条件变量

条件变量 (condition variable) 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() 即是条件变量（它们以容易用错著称）。条件变量只有一种正确使用的方式，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

写成代码是：

```
MutexLock mutex;
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 blocking，不会与 enqueue 死锁
  }
  assert(!queue.empty());
  int top = queue.front();
  queue.pop_front();
  return top;
}
```

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）

写成代码是：

```
void enqueue(int x)
{
  MutexLockGuard lock(mutex);
  queue.push_back(x);
  cond.notify();
}
```

上面的 dequeue/enqueue 实际上实现了一个简单的 unbounded BlockingQueue。

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 或 CountDownLatch。

读写锁与其他

读写锁 (Reader-Writer lock)，读写锁是个优秀的抽象，它明确区分了 read 和 write 两种行为。需要注意的是，reader lock 是可重入的，writer lock 是不可重入（包括不可提升 reader lock）的。这正是我说它“优秀”的主要原因。

遇到并发读写，如果条件合适，我会用《借 shared_ptr 实现线程安全的 copy-on-write》http://blog.csdn.net/Solstice/archive/2008/11/22/3351751.aspx 介绍的办法，而不用读写锁。当然这不是绝对的。

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。

说一句大逆不道的话，如果程序里需要解决如“哲学家就餐”之类的复杂 IPC 问题，我认为应该首先考察几个设计，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）？能不能把“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，在 Linux 下正确模拟 WaitForMultipleObjects 不是普通程序员该干的。

封装 MutexLock、MutexLockGuard 和 Condition

本节把前面用到的 MutexLock、MutexLockGuard、Condition classes 的代码列出来，前面两个 classes 没多大难度，后面那个有点意思。

MutexLock 封装临界区（Critical secion），这是一个简单的资源类，用 RAII 手法[CCS:13][1]封装互斥器的创建与销毁。临界区在 Windows 上是 CRITICAL_SECTION，是可重入的；在 Linux 下是 pthread_mutex_t，默认是不可重入的。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 classes 应该能在纸上默写出来，没有太多需要解释的：

```
#include <pthread.h>
#include <boost/noncopyable.hpp>

class MutexLock : boost::noncopyable
{
 public:
```

[1] 《C++ 编程规范》/《C++ Coding Standards》, by Herb Sutter and Andrei Alexandrescu, 2005. 条款 13.

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

  ~MutexLock()
  { pthread_mutex_destroy(&mutex_); }

  void lock()  // 程序一般不主动调用
  { pthread_mutex_lock(&mutex_); }

  void unlock()  // 程序一般不主动调用
  { pthread_mutex_unlock(&mutex_); }

  pthread_mutex_t* getPthreadMutex()  // 仅供 Condition 调用，严禁自己调用
  { return &mutex_; }

 private:
  pthread_mutex_t mutex_;
};

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);  // 没有变量名，产生一个临时对象又马上销毁了，没有锁住临界区
  // 正确写法是 MutexLockGuard lock(mutex);

  // 临界区
}
```

这里 MutexLock 没有提供 trylock() 函数，因为我没有用过它，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。

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，于是我人肉把模板具现化 (instantiate) 了。此外一种更激进的写法是，把 lock/unlock 放到 private 区，然后把 Guard 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：a) 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。b) 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() 是个不错的思路。

以上两点改进留作练习。

Condition class 的实现有点意思。

Pthreads condition variable 允许在 wait() 的时候指定 mutex，但是我想不出什么理由一个 condition variable 会和不同的 mutex 配合使用。Java 的 intrinsic condition 和 Conditon class 都不支持这么做，因此我觉得可以放弃这一灵活性，老老实实一对一好了。相反 boost::thread 的 condition_varianle 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 有五六种： 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“灵活”的库我只得三揖绕道而行。这些 class 名字也很无厘头，为什么不老老实实用 reader_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 classes 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 Condition 简单地封装了 pthread cond var，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:
  Condition(MutexLock& mutex) : mutex_(mutex)
  { pthread_cond_init(&pcond_, NULL); }

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

  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)
   : count_(count),
     mutex_(),
     condition_(mutex_)
  { }

 private:
  int count_;
  MutexLock mutex_;  // 顺序很重要
  Condition condition_;
};
```

请允许我再次强调，虽然本节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-Helgrind 等能识别 pthreads 调用，并依据 happens-before 关系 [Lamport 1978] 分析程序有无 data race。

线程安全的 Singleton 实现

研究 Signleton 的线程安全实现的历史你会发现很多有意思的事情，一度人们认为 Double checked locking 是王道，兼顾了效率与正确性。后来有神牛指出由于乱序执行的影响，DCL 是靠不住的。（这个又让我想起了 SQL 注入，十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。）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（ http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf ）。接下来 Java 5 修订了内存模型，并增强了 volatile 的语义，这下 DCL (with volatile) 又是安全的了。然而 C++ 的内存模型还在修订中，C++ 的 volatile 目前还不能（将来也难说）保证 DCL 的正确性（只在 VS2005+ 上有效）。

其实没那么麻烦，在实践中用 pthread once 就行：

```
#include <pthread.h>

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

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

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

template<typename T>
pthread_once_t Singleton<T>::ponce_ = PTHREAD_ONCE_INIT;

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

上面这个 Singleton 没有任何花哨的技巧，用 pthread_once_t 来保证 lazy-initialization 的线程安全。使用方法也很简单：

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

当然，这个 Singleton 没有考虑对象的销毁，在服务器程序里，这不是一个问题，因为当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）。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。

归纳

- 进程间通信首选 TCP sockets
- 线程同步的四项原则
- 使用互斥器的条件变量的惯用手法 (idiom)，关键是 RAII

用好这几样东西，基本上能应付多线程服务端开发的各种场合，只是或许有人会觉得性能没有发挥到极致。我认为，先把程序写正确了，再考虑性能优化，这在多线程下任然成立。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。

7 总结

在现代的多核计算背景下，线程是不可避免的。尽管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，往往比重头实现一个多线程的程序更难。

掌握同步原语和它们的适用场合时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，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，a) TCP 不是效率最高的 IPC，b) 我提倡正确加锁而不是自己编写 lock-free 算法（使用原子操作除外）。在程序的复杂度和性能之前取得平衡，并经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）。下一篇“多线程编程的反模式”会考察伸缩性方面的常见错误，我认为在分布式系统中，伸缩性 (scalability) 比单机的性能优化更值得投入精力。

这篇文章记录了我目前对多线程编程的理解，用文中介绍的手法，我能解决自己面临的全部多线程编程任务。如果文章的观点与您不合，比如您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），只要您理由充分，但行无妨。

这篇文章本来还有两节“多线程编程的反模式”与“多线程的应用场景”，考虑到字数已经超过一万了，且听下回分解吧 :-)

后文预览：Sleep 反模式

我认为 sleep 只能出现在测试代码中，比如写单元测试的时候。（涉及时间的单元测试不那么好写，短的如一两秒钟可以用 sleep，长的如一小时一天得想其他办法，比如把算法提出来并把时间注入进去。）产品代码中线程的等待可分为两种：一种是无所事事的时候（要么等在 select/poll/epoll 上。要么等在 condition variable 上，等待 BlockingQueue /CountDownLatch 亦可归入此类），一种是等着进入临界区（等在 mutex 上）以便继续处理。在程序的正常执行中，如果需要等待一段时间，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。如果多线程的安全性和效率要靠代码主动调用 sleep 来保证，这是设计出了问题。等待一个事件发生，正确的做法是用 select 或 condition variable 或（更理想地）高层同步工具。当然，在 GUI 编程中会有主动让出 CPU 的做法，比如调用 sleep(0) 来实现 yield。