IBM

使用事件驱动模型实现高效稳定的网络服务器程序

几种网络服务器模型的介绍与比较

围绕如何构建一个高效稳定的网络服务器程序，本文从一个最简单的服务器模型开始，依次介绍了使用多线程的服务器模型、使用非阻塞接口的服务器模型、利用select()接口实现的基于事件驱动的服务器模型，和使用libev事件驱动库的服务器模型。通过比较各个模型，得出事件驱动模型更适合构建高效稳定的网络服务器程序的结论。

顾锋磊 , IBM 中国系统与技术中心软件工程师，2008 年加入 IBM，从事软件开发工作。

2010 年 10 月 14 日

前言

事件驱动为广大的程序员所熟悉，其最为人津津乐道的是在图形化界面编程中的应用；事实上，在网络编程中事件驱动也被广泛使用，并大规模部署在高连接数高吞吐量的服务器程序中，如 http 服务器程序、ftp 服务器程序等。相比于传统的网络编程方式，事件驱动能够极大的降低资源占用，增大服务接待能力，并提高网络传输效率。

关于本文提及的服务器模型，搜索网络可以查阅到很多的实现代码，所以，本文将不拘泥于源代码的陈列与分析，而侧重模型的介绍和比较。使用 libev 事件驱动库的服务器模型将给出实现代码。

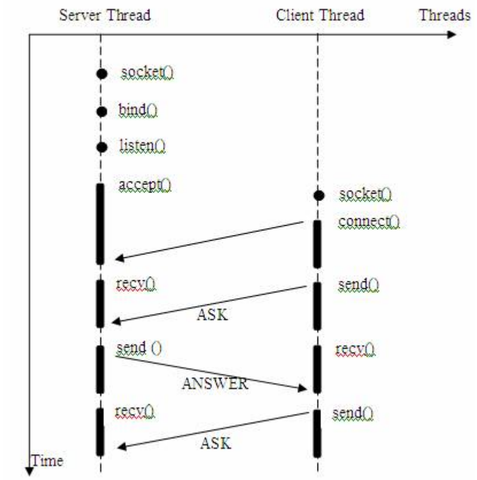
本文涉及到线程 / 时间图例，只为表明线程在各个 IO 上确实存在阻塞时延，但并不保证时延比例的正确性和 IO 执行先后的正确性；另外，本文所提及到的接口也只是笔者熟悉的 Unix/Linux 接口，并未推荐 Windows 接口，读者可以自行查阅对应的 Windows 接口。

阻塞型的网络编程接口

几乎所有的程序员第一次接触到的网络编程都是从 listen()、send()、recv() 等接口开始的。使用这些接口可以很方便的构建服务器 / 客户机的模型。

我们假设希望建立一个简单的服务器程序，实现向单个客户机提供类似于“一问一答”的内容服务。

图 1. 简单的一问一答的服务器 / 客户机模型



我们注意到，大部分的 socket 接口都是阻塞型的。所谓阻塞型接口是指系统调用（一般是 IO 接口）不返回调用结果并让当前线程一直阻塞，只有当该系统调用获得结果或者超时出错时才返回。

实际上，除非特别指定，几乎所有的 IO 接口 (包括 socket 接口) 都是阻塞型的。这给网络编程带来了一个很大的问题，如在调用 send() 的同时，线程将被阻塞，在此期间，线程将无法执行任何运算或响应任何的网络请求。这给多客户机、多业务逻辑的网络编程带来了挑战。这时，很多程序员可能会选择多线程的方式来解决这个问题。

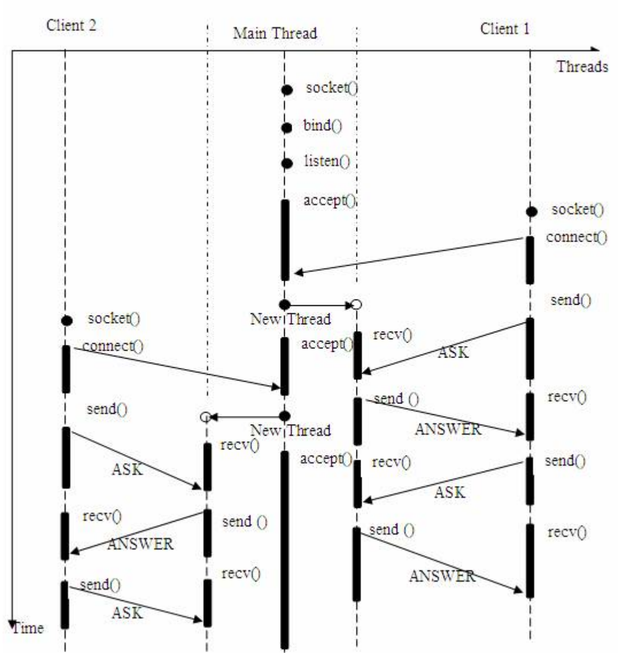
多线程的服务器程序

应对多客户机的网络应用，最简单的解决方式是在服务器端使用多线程（或多进程）。多线程（或多进程）的目的是让每个连接都拥有独立的线程（或进程），这样任何一个连接的阻塞都不会影响其他的连接。

具体使用多进程还是多线程，并没有一个特定的模式。传统意义上，进程的开销要远远大于线程，所以，如果需要同时为较多的客户机提供服务，则不推荐使用多进程；如果单个服务执行体需要消耗较多的 CPU 资源，譬如需要进行大规模或长时间的数据运算或文件访问，则进程较为安全。通常，使用 pthread_create () 创建新线程，fork() 创建新进程。

我们假设对上述的服务器 / 客户机模型，提出更高的要求，即让服务器同时为多个客户机提供一问一答的服务。于是有了如下的模型。

图 2. 多线程的服务器模型



在上述的线程 / 时间图例中，主线程持续等待客户端的连接请求，如果有连接，则创建新线程，并在新线程中提供为前例同样的问答服务。

很多初学者可能不明白为何一个 socket 可以 accept 多次。实际上，socket 的设计者可能特意为多客户机的情况留下了伏笔，让 accept() 能够返回一个新的 socket。下面是 accept 接口的原型：

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

输入参数 s 是从 socket()，bind() 和 listen() 中沿用下来的 socket 句柄值。执行完 bind() 和 listen() 后，操作系统已经开始在指定的端口处监听所有的连接请求，如果有请求，则将该连接请求加入请求队列。调用 accept() 接口正是从 socket s 的请求队列抽取第一个连接信息，创建一个与 s 同类的新的 socket 返回句柄。新的 socket 句柄即是后续 read() 和 recv() 的输入参数。如果请求队列当前没有请求，则 accept() 将进入阻塞状态直到有请求进入队列。

上述多线程的服务器模型似乎完美的解决了为多个客户机提供问答服务的要求，但其实并不尽然。如果要同时响应成百上千路的连接请求，则无论多线程还是多进程都会严重占据系统资源，降低系统对外界响应效率，而线程与进程本身也更容易进入假死状态。

很多程序员可能会考虑使用“线程池”或“连接池”。“线程池”旨在减少创建和销毁线程的频率，其维持一定合理数量的线程，并让空闲的线程重新承担新的执行任务。“连接池”维持连接的缓存池，尽量重用已有的连接、减少创建和关闭连接的频率。这两种技术都可以很好的降低系统开销，都被广泛应用很多大型系统，如 websphere、tomcat 和各种数据库等。

但是，“线程池”和“连接池”技术也只是在一定程度上缓解了频繁调用 IO 接口带来的资源占用。而且，所谓“池”始终有其上限，当请求大大超过上限时，“池”构成的系统对外界的响应并不比没有池的时候效果好多少。所以使用“池”必须考虑其面临的响应规模，并根据响应规模调整“池”的大小。

对应上例中的所面临的可能同时出现的上千甚至上万次的客户端请求，“线程池”或“连接池”或许可以缓解部分压力，但是不能解决所有问题。

总之，多线程模型可以方便高效的解决小规模的服务请求，但面对大规模的服务请求，多线程模型并不是最佳方案。下一章我们将讨论用非阻塞接口来尝试解决这个问题。

非阻塞的服务器程序

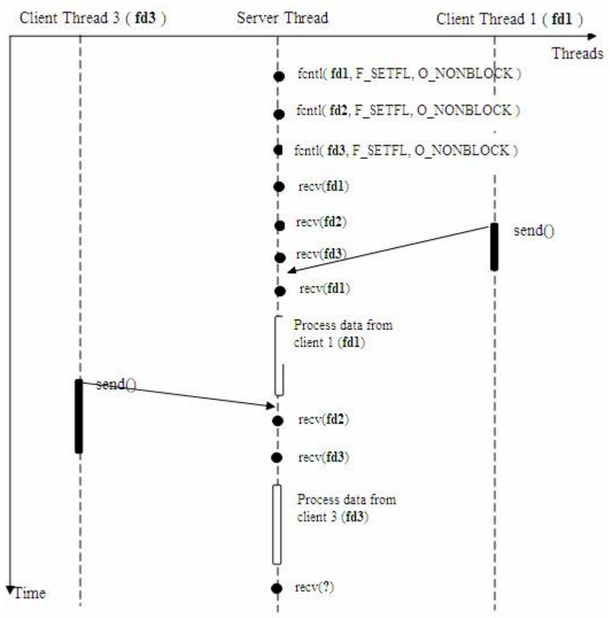
以上面临的很多问题，一定程度是 IO 接口的阻塞特性导致的。多线程是一个解决方案，还一个方案就是使用非阻塞的接口。

非阻塞的接口相比于阻塞型接口的显著差异在于，在被调用之后立即返回。使用如下的函数可以将某句柄 fd 设为非阻塞状态。

```
        fcntl( fd, F_SETFL, O_NONBLOCK );
```

下面将给出只用一个线程，但能够同时从多个连接中检测数据是否送达，并且接受数据。

图 3. 使用非阻塞的接收数据模型



在非阻塞状态下，recv() 接口在被调用后立即返回，返回值代表了不同的含义。如在本例中，

- recv() 返回值大于 0，表示接受数据完毕，返回值即是接受到的字节数；
- recv() 返回 0，表示连接已经正常断开；
- recv() 返回 -1，且 errno 等于 EAGAIN，表示 recv 操作还没执行完成；
- recv() 返回 -1，且 errno 不等于 EAGAIN，表示 recv 操作遇到系统错误 errno。

可以看到服务器线程可以通过循环调用 recv() 接口，可以在单个线程内实现对所有连接的数据接收工作。

但是上述模型绝不被推荐。因为，循环调用 recv() 将大幅度推高 CPU 占用率；此外，在这个方案中，recv() 更多的是起到检测“操作是否完成”的作用，实际操作系统提供了更为高效的检测“操作是否完成“作用的接口，例如 select()。

使用 **select()** 接口的基于事件驱动的服务器模型

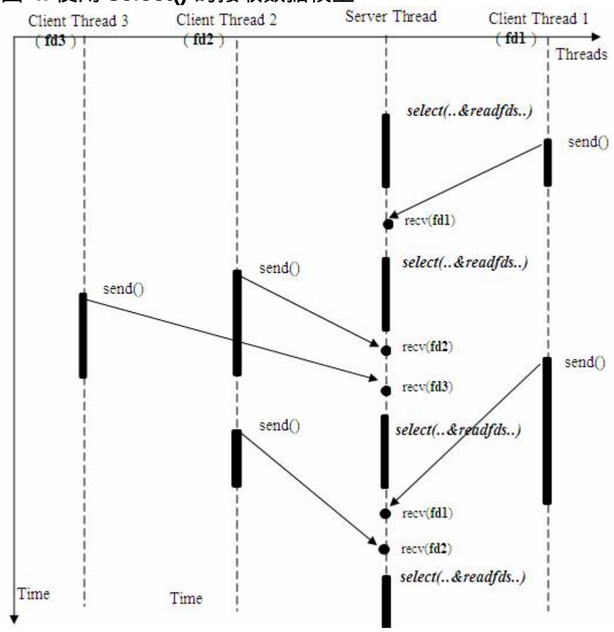
大部分 Unix/Linux 都支持 select 函数，该函数用于探测多个文件句柄的状态变化。下面给出 select 接口的原型：

```
FD_ZERO(int fd, fd_set* fds)
FD_SET(int fd, fd_set* fds)
FD_ISSET(int fd, fd_set* fds)
FD_CLR(int fd, fd_set* fds)
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
       struct timeval *timeout)
```

这里，fd_set 类型可以简单的理解为按 bit 位标记句柄的队列，例如要在某 fd_set 中标记一个值为 16 的句柄，则该 fd_set 的第 16 个 bit 位被标记为 1。具体的置位、验证可使用 FD_SET、FD_ISSET 等宏实现。在 select() 函数中，readfds、writefds 和 exceptfds 同时作为输入参数和输出参数。如果输入的 readfds 标记了 16 号句柄，则 select() 将检测 16 号句柄是否可读。在 select() 返回后，可以通过检查 readfds 有否标记 16 号句柄，来判断该“可读”事件是否发生。另外，用户可以设置 timeout 时间。

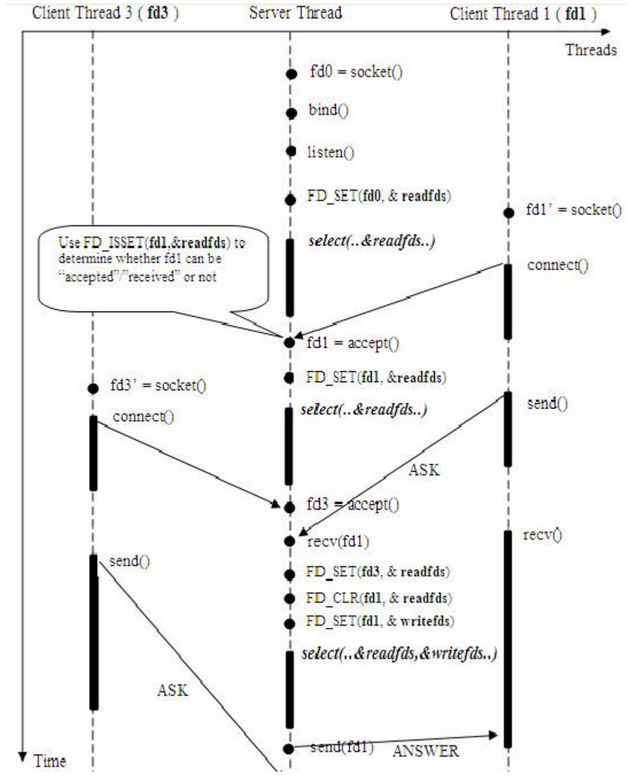
下面将重新模拟上例中从多个客户端接收数据的模型。

图 4. 使用 select() 的接收数据模型



上述模型只是描述了使用 select() 接口同时从多个客户端接收数据的过程；由于 select() 接口可以同时对多个句柄进行读状态、写状态和错误状态的探测，所以可以很容易构建为多个客户端提供独立问答服务的服务器系统。

图 5. 使用 select() 接口的基于事件驱动的服务器模型



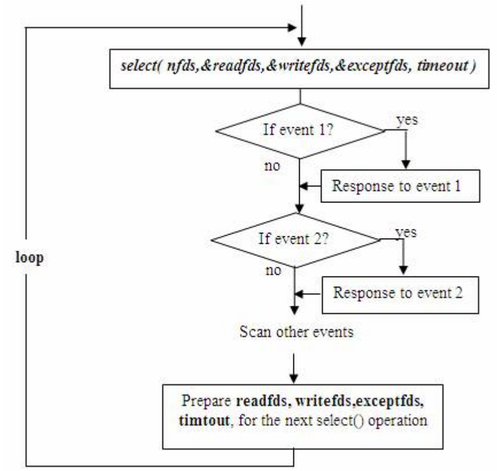
这里需要指出的是，客户端的一个 connect() 操作，将在服务器端激发一个“可读事件”，所以 select() 也能探测来自客户端的 connect() 行为。

上述模型中，最关键的地方是如何动态维护 select() 的三个参数 readfds、writefds 和 exceptfds。作为输入参数，readfds 应该标记所有的需要探测的“可读事件”的句柄，其中永远包括那个探测 connect() 的那个“母”句柄；同时，writefds 和 exceptfds 应该标记所有需要探测的“可写事件”和“错误事件”的句柄 (使用 FD_SET() 标记)。

作为输出参数，readfds、writefds 和 exceptfds 中的保存了 select() 捕捉到的所有事件的句柄值。程序员需要检查的所有的标记位 (使用 FD_ISSET() 检查)，以确定到底哪些句柄发生了事件。

上述模型主要模拟的是“一问一答”的服务流程，所以，如果 select() 发现某句柄捕捉到了“可读事件”，服务器程序应及时做 recv() 操作，并根据接收到的数据准备好待发送数据，并将对应的句柄值加入 writefds，准备下一次的“可写事件”的 select() 探测。同样，如果 select() 发现某句柄捕捉到“可写事件”，则程序应及时做 send() 操作，并准备好下一次的“可读事件”探测准备。下图描述的是上述模型中的一个执行周期。

图 6. 一个执行周期



这种模型的特征在于每一个执行周期都会探测一次或一组事件，一个特定的事件会触发某个特定的响应。我们可以将这种模型归类为“事件驱动模型”。

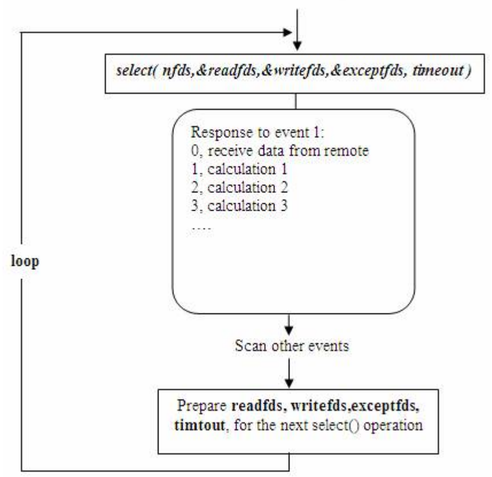
相比其他模型，使用 select() 的事件驱动模型只用单线程（进程）执行，占用资源少，不消耗太多 CPU，同时能够为多客户端提供服务。如果试图建立一个简单的事件驱动的服务器程序，这个模型有一定的参考价值。

但这个模型依旧有着很多问题。

首先，select() 接口并不是实现“事件驱动”的最好选择。因为当需要探测的句柄值较大时，select() 接口本身需要消耗大量时间去轮询各个句柄。很多操作系统提供了更为高效的接口，如 linux 提供了 epoll，BSD 提供了 kqueue，Solaris 提供了 /dev/poll …。如果需要实现更高效的服务器程序，类似 epoll 这样的接口更被推荐。遗憾的是不同的操作系统特供的 epoll 接口有很大差异，所以使用类似于 epoll 的接口实现具有较好跨平台能力的服务器会比较困难。

其次，该模型将事件探测和事件响应夹杂在一起，一旦事件响应的执行体庞大，则对整个模型是灾难性的。如下例，庞大的执行体 1 的将直接导致响应事件 2 的执行体迟迟得不到执行，并在很大程度上降低了事件探测的及时性。

图 7. 庞大的执行体对使用 **select()** 的事件驱动模型的影响



幸运的是，有很多高效的事件驱动库可以屏蔽上述的困难，常见的事件驱动库有 libevent 库，还有作为

libevent 替代者的 libev 库。这些库会根据操作系统的特点选择最合适的事件探测接口，并且加入了信号 (signal) 等技术以支持异步响应，这使得这些库成为构建事件驱动模型的不二选择。下章将介绍如何使用 libev 库替换 select 或 epoll 接口，实现高效稳定的服务器模型。

使用事件驱动库 libev 的服务器模型

Libev 是一种高性能事件循环 / 事件驱动库。作为 libevent 的替代作品，其第一个版本发布与 2007 年 11 月。Libev 的设计者声称 libev 拥有更快的速度，更小的体积，更多功能等优势，这些优势在很多测评中得到了证明。正因为其良好的性能，很多系统开始使用 libev 库。本章将介绍如何使用 Libev 实现提供问答服务的服务器。

（事实上，现存的事件循环 / 事件驱动库有很多，作者也无意推荐读者一定使用 libev 库，而只是为了说明事件驱动模型给网络服务器编程带来的便利和好处。大部分的事件驱动库都有着与 libev 库相类似的接口，只要明白大致的原理，即可灵活挑选合适的库。）

与前章的模型类似，libev 同样需要循环探测事件是否产生。Libev 的循环体用 ev_loop 结构来表达，并用 ev_loop( ) 来启动。

```
        void ev_loop( ev_loop* loop, int flags )
```

Libev 支持八种事件类型，其中包括 IO 事件。一个 IO 事件用 ev_io 来表征，并用 ev_io_init() 函数来初始化：

```
        void ev_io_init(ev_io *io, callback, int fd, int events)
```

初始化内容包括回调函数 callback，被探测的句柄 fd 和需要探测的事件，EV_READ 表“可读事件”，EV_WRITE 表“可写事件”。

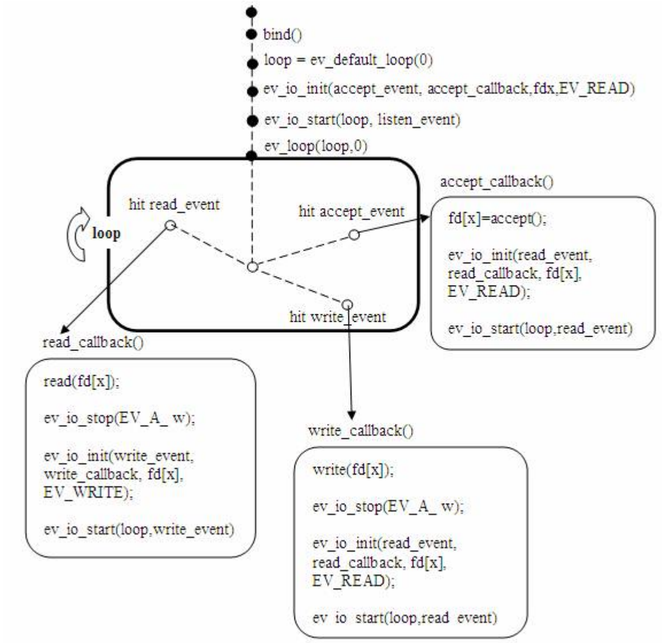
现在，用户需要做的仅仅是在合适的时候，将某些 ev_io 从 ev_loop 加入或剔除。一旦加入，下个循环即会检查 ev_io 所指定的事件有否发生；如果该事件被探测到，则 ev_loop 会自动执行 ev_io 的回调函数 callback()；如果 ev_io 被注销，则不再检测对应事件。

无论某 ev_loop 启动与否，都可以对其添加或删除一个或多个 ev_io，添加删除的接口是 ev_io_start() 和 ev_io_stop()。

```
        void ev_io_start( ev_loop *loop, ev_io* io )
        void ev_io_stop( EV_A_* )
```

由此，我们可以容易得出如下的“一问一答”的服务器模型。由于没有考虑服务器端主动终止连接机制，所以各个连接可以维持任意时间，客户端可以自由选择退出时机。

图 8. 使用 libev 库的服务器模型



上述模型可以接受任意多个连接，且为各个连接提供完全独立的问答服务。借助 libev 提供的事件循环 / 事件驱动接口，上述模型有机会具备其他模型不能提供的高效率、低资源占用、稳定性好和编写简单等特点。

由于传统的 web 服务器，ftp 服务器及其他网络应用程序都具有“一问一答”的通讯逻辑，所以上述使用 libev 库的“一问一答”模型对构建类似的服务器程序具有参考价值；另外，对于需要实现远程监视或远程遥控的应用程序，上述模型同样提供了一个可行的实现方案。

总结

本文围绕如何构建一个提供“一问一答”的服务器程序，先后讨论了用阻塞型的 socket 接口实现的模型，使用多线程的模型，使用 select() 接口的基于事件驱动的服务器模型，直到使用 libev 事件驱动库的服务器模型。文章对各种模型的优缺点都做了比较，从比较中得出结论，即使用“事件驱动模型”可以的实现更为高效稳定的服务器程序。文中描述的多种模型可以为读者的网络编程提供参考价值。

参考资料

学习

- libev 官网提供了 libev 的源码和文档。
- libevent 官网提供了 libevent 的源码和文档。
- 《 UNIX 环境高级编程》，作者 W. Richard Stevens & Stephen A. Rago。
- 在 developerWorks Linux 专区 寻找为 Linux 开发人员（包括 Linux 新手入门）准备的更多参考资料，查阅我们 最受欢迎的文章和教程。
- 在 developerWorks 上查阅所有 Linux 技巧 和 Linux 教程。
- 随时关注 developerWorks 技术活动和网络广播。

讨论

- 欢迎加入 My developerWorks 中文社区。