使用 **libevent** 和 **libev** 提高网络应用性能

Martin C. Brown, 作家, Freelance

简介： 构建现代的服务器应用程序需要以某种方法同时接收数百、数千甚至数万个事件，无论它们是内部请求还是网络连接，都要有效地处理它们的操作。有许多解决方案，但是 libevent 库和 libev 库能够大大提高性能和事件处理能力。在本文中，我们要讨论在 UNIX® 应用程序中使用和部署这些解决方案所用的基本结构和方法。libev 和 libevent 都可以在高性能应用程序中使用，包括部署在 IBM Cloud 或 Amazon EC2 环境中的应用程序，这些应用程序需要支持大量并发客户端或操作。

简介

许多服务器部署（尤其是 web 服务器部署）面对的最大问题之一是必须能够处理大量连接。无论是通过构建基于云的服务来处理网络通信流，还是把应用程序分布在 IBM Amazon EC 实例上，还是为网站提供高性能组件，都需要能够处理大量并发连接。

一个好例子是，web 应用程序最近越来越动态了，尤其是使用 AJAX 技术的应用程序。如果要部署的系统允许数千客户端直接在网页中更新信息，比如提供事件或问题实时监视的系统，那么提供信息的速度就非常重要了。在网格或云环境中，可能有来自数千客户端的持久连接同时打开着，必须能够处理每个客户端的请求并做出响应。

在讨论 libevent 和 libev 如何处理多个网络连接之前，我们先简要回顾一下处理这类连接的传统解决方案。

处理多个客户端

处理多个连接有许多不同的传统方法，但是在处理大量连接时它们往往会产生问题，因为它们使用的内存或 CPU 太多，或者达到了某个操作系统限制。

使用的主要方法如下：

> 循环：早期系统使用简单的循环选择解决方案，即循环遍历打开的网络连接的列表，判断是否有要读取的数据。这种方法既缓慢（尤其是随着连接数量增加越来越慢），又低效（因为在处理当前连接时其他连接可能正在发送请求并等待响应）。在系统循环遍历每个连接时，其他连接不得不等待。如果有 100 个连接，其中只有一个有数据，那么仍然必须处理其他 99 个连接，才能轮到真正需要处理的连接。
> **poll**、**epoll** 和变体：这是对循环方法的改进，它用一个结构保存要监视的每个连接的数组，当在网络套接字上发现数据时，通过回调机制调用处理函数。poll 的问题是这个结构会非常大，在列表中添加新的网络连接时，修改结构会增加负载并影响性能。
> 选择：`select()` 函数调用使用一个静态结构，它事先被硬编码为相当小的数量（1024 个连接），因此不适用于非常大的部署。

在各种平台上还有其他实现（比如 Solaris 上的 /dev/poll 或 FreeBSD/NetBSD 上的 kqueue），它们在各自的 OS 上性能可能更好，但是无法移植，也不一定能够解决处理请求的高层问题。

上面的所有解决方案都用简单的循环等待并处理请求，然后把请求分派给另一个函数以处理实际的网络交互。关键在于循环和网络套接字需要大量管理代码，这样才能监听、更新和控制不同的连接和接口。

处理许多连接的另一种方法是，利用现代内核中的多线程支持监听和处理连接，为每个连接启动一个新线程。这把责任直接交给操作系统，但是会在 RAM 和 CPU 方面增加相当大的开销，因为每个线程都需要自己的执行空间。另外，如果每个线程都忙于处理网络连接，线程之间的上下文切换会很频繁。最后，许多内核并不适于处理如此大量的活跃线程。

libevent 方法

libevent 库实际上没有更换 `select()`、`poll()` 或其他机制的基础。而是使用对于每个平台最高效的高性能解决方案在实现外加上一个包装器。

为了实际处理每个请求，libevent 库提供一种事件机制，它作为底层网络后端的包装器。事件系统让为连接添加处理函数变得非常简便，同时降低了底层 I/O 复杂性。这是 libevent 系统的核心。

libevent 库的其他组件提供其他功能，包括缓冲的事件系统（用于缓冲发送到客户端/从客户端接收的数据）以及 HTTP、DNS 和 RPC 系统的核心实现。

创建 libevent 服务器的基本方法是，注册当发生某一操作（比如接受来自客户端的连接）时应该执行的函数，然后调用主事件循环 `event_dispatch()`。执行过程的控制现在由 libevent 系统处理。注册事件和将调用的函数之后，事件系统开始自治；在应用程序运行时，可以在事件队列中添加（注册）或删除（取消注册）事件。事件注册非常方便，可以通过它添加新事件以处理新打开的连接，从而构建灵活的网络处理系统。

例如，可以打开一个监听套接字，然后注册一个回调函数，每当需要调用 `accept()` 函数以打开新连接时调用这个回调函数，这样就创建了一个网络服务器。清单 1 所示的代码片段说明基本过程：

清单 1. 打开监听套接字，注册一个回调函数（每当需要调用 `accept()` 函数以打开新连接时调用它），由此创建网络服务器

```
int main(int argc, char **argv)
{
...
    ev_init();

    /* Setup listening socket */

    event_set(&ev_accept, listen_fd, EV_READ|EV_PERSIST, on_accept, NULL);
    event_add(&ev_accept, NULL);

    /* Start the event loop. */
    event_dispatch();
}
```

`event_set()` 函数创建新的事件结构，`event_add()` 在事件队列机制中添加事件，然

event_set() 函数创建新的事件结构，event_add() 在事件队列机制中添加事件。然后，event_dispatch() 启动事件队列系统，开始监听（并接受）请求。

清单 2 给出一个更完整的示例，它构建一个非常简单的回显服务器：

清单 2. 构建简单的回显服务器

```
#include <event.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define SERVER_PORT 8080
int debug = 0;

struct client {
  int fd;
  struct bufferevent *buf_ev;
};

int setnonblock(int fd)
{
  int flags;

  flags = fcntl(fd, F_GETFL);
  flags |= O_NONBLOCK;
  fcntl(fd, F_SETFL, flags);
}

void buf_read_callback(struct bufferevent *incoming,
                       void *arg)
{
  struct evbuffer *evreturn;
  char *req;

  req = evbuffer_readline(incoming->input);
  if (req == NULL)
    return;

  evreturn = evbuffer_new();
  evbuffer_add_printf(evreturn,"You said %s\n",req);
  bufferevent_write_buffer(incoming,evreturn);
  evbuffer_free(evreturn);
  free(req);
}

void buf_write_callback(struct bufferevent *bev,
                        void *arg)
{
}

void buf_error_callback(struct bufferevent *bev,
                        short what,
```

```
                        short what,
                        void *arg)
{
  struct client *client = (struct client *)arg;
  bufferevent_free(client->buf_ev);
  close(client->fd);
  free(client);
}

void accept_callback(int fd,
                     short ev,
                     void *arg)
{
  int client_fd;
  struct sockaddr_in client_addr;
  socklen_t client_len = sizeof(client_addr);
  struct client *client;

  client_fd = accept(fd,
                     (struct sockaddr *)&client_addr,
                     &client_len);
  if (client_fd < 0)
    {
      warn("Client: accept() failed");
      return;
    }

  setnonblock(client_fd);

  client = calloc(1, sizeof(*client));
  if (client == NULL)
    err(1, "malloc failed");
  client->fd = client_fd;

  client->buf_ev = bufferevent_new(client_fd,
                                   buf_read_callback,
                                   buf_write_callback,
                                   buf_error_callback,
                                   client);

  bufferevent_enable(client->buf_ev, EV_READ);
}

int main(int argc,
         char **argv)
{
  int socketlisten;
  struct sockaddr_in addresslisten;
  struct event accept_event;
  int reuse = 1;

  event_init();

  socketlisten = socket(AF_INET, SOCK_STREAM, 0);

  if (socketlisten < 0)
    {
      fprintf(stderr,"Failed to create listen socket");
      return 1;
    }

  memset(&addresslisten, 0, sizeof(addresslisten));
```

```
  addresslisten.sin_family = AF_INET;
  addresslisten.sin_addr.s_addr = INADDR_ANY;
  addresslisten.sin_port = htons(SERVER_PORT);

  if (bind(socketlisten,
          (struct sockaddr *)&addresslisten,
          sizeof(addresslisten)) < 0)
    {
      fprintf(stderr,"Failed to bind");
      return 1;
    }

  if (listen(socketlisten, 5) < 0)
    {
      fprintf(stderr,"Failed to listen to socket");
      return 1;
    }

  setsockopt(socketlisten,
             SOL_SOCKET,
             SO_REUSEADDR,
             &reuse,
             sizeof(reuse));

  setnonblock(socketlisten);

  event_set(&accept_event,
            socketlisten,
            EV_READ|EV_PERSIST,
            accept_callback,
            NULL);

  event_add(&accept_event,
            NULL);

  event_dispatch();

  close(socketlisten);

  return 0;
}
```

下面讨论各个函数及其操作：

`main()`：主函数创建用来监听连接的套接字，然后创建 `accept()` 的回调函数以便通过事件处理函数处理每个连接。

`accept_callback()`：当接受连接时，事件系统调用此函数。此函数接受到客户端的连接；添加客户端套接字信息和一个 bufferevent 结构；在事件结构中为客户端套接字上的读/写/错误事件添加回调函数；作为参数传递客户端结构（和嵌入的 eventbuffer 和客户端套接字）。每当对应的客户端套接字包含读、写或错误操作时，调用对应的回调函数。

`buf_read_callback()`：当客户端套接字有要读的数据时调用它。作为回显服务，此函数把 "you said..." 写回客户端。套接字仍然打开，可以接受新请求。

`buf_write_callback()`：当有要写的数据时调用它。在这个简单的服务中，不需要此函数，所以定义是空的。

`buf_error_callback()`：当出现错误时调用它。这包括客户端中断连接。在出现错误的所有场景中，关闭客户端套接字，从事件列表中删除客户端套接字的事件条目，释放客户端结构的内存。

`setnonblock()`：设置网络套接字以开放 I/O。

当客户端连接时，在事件队列中添加新事件以处理客户端连接；当客户端中断连接时删除事件。在幕后，libevent 处理网络套接字，识别需要服务的客户端，分别调用对应的函数。

为了构建这个应用程序，需要编译 C 源代码并添加 libevent 库：`$ gcc -o basic basic.c -levent`。

从客户端的角度来看，这个服务器仅仅把发送给它的任何文本发送回来（见 清单 3）。

清单 3. 服务器把发送给它的文本发送回来

```
$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello!
You said Hello!
```

这样的网络应用程序非常适合需要处理多个连接的大规模分布式部署，比如 IBM Cloud 系统。

很难通过简单的解决方案观察处理大量并发连接的情况和性能改进。可以使用嵌入的 HTTP 实现帮助了解可伸缩性。

使用内置的 HTTP 服务器

如果希望构建本机应用程序，可以使用一般的基于网络的 libevent 接口；但是，越来越常见的场景是开发基于 HTTP 协议的应用程序，以及装载或动态地重新装载信息的网页。如果使用任何 AJAX 库，客户端就需要 HTTP，即使您返回的信息是 XML 或 JSON。

libevent 中的 HTTP 实现并不是 Apache HTTP 服务器的替代品，而是适用于与云和 web 环境相关联的大规模动态内容的实用解决方案。例如，可以在 IBM Cloud 或其他解决方案中部署基于 libevent 的接口。因为可以使用 HTTP 进行通信，服务器可以与其他组件集成。

要想使用 libevent 服务，需要使用与主要网络事件模型相同的基本结构，但是还必须处理网络接口，HTTP 包装器会替您处理。这使整个过程变成四个函数调用（初始化、启动 HTTP 服务器、设置 HTTP 回调函数和进入事件循环），再加上发送回数据的回调函数。清单 4 给出一个非常简单的示例：

清单 4. 使用 libevent 服务的简单示例

```
#include <sys/types.h>

#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>

#include <event.h>
#include <evhttp.h>

void generic_request_handler(struct evhttp_request *req, void *arg)
{
  struct evbuffer *returnbuffer = evbuffer_new();

  evbuffer_add_printf(returnbuffer, "Thanks for the request!");
  evhttp_send_reply(req, HTTP_OK, "Client", returnbuffer);
  evbuffer_free(returnbuffer);
  return;
}

int main(int argc, char **argv)
{
  short           http_port = 8081;
  char           *http_addr = "192.168.0.22";
  struct evhttp *http_server = NULL;

  event_init();
  http_server = evhttp_start(http_addr, http_port);
  evhttp_set_gencb(http_server, generic_request_handler, NULL);

  fprintf(stderr, "Server started on port %d\n", http_port);
  event_dispatch();

  return(0);
}
```

应该可以通过前面的示例看出代码的基本结构，不需要解释。主要元素是 `evhttp_set_gencb()` 函数（它设置当收到 HTTP 请求时要使用的回调函数）和 `generic_request_handler()` 回调函数本身（它用一个表示成功的简单消息填充响应缓冲区）。

HTTP 包装器提供许多其他功能。例如，有一个请求解析器，它会从典型的请求中提取出查询参数（就像处理 CGI 请求一样）。还可以设置在不同的请求路径中要触发的处理函数。通过设置不同的回调函数和处理函数，可以使用路径 '/db/' 提供到数据库的接口，或使用 '/memc' 提供到 memcached 的接口。

libevent 工具包的另一个特性是支持通用计时器。可以在指定的时间段之后触发事件。可以通过结合使用计时器和 HTTP 实现提供轻量的服务，从而自动地提供文件内容，在修改文件内容时更新返回的数据。例如，以前要想在新闻频发的活动期间提供即时更新服务，前端 web 应用程序就需要定期重新装载新闻稿，而现在可以轻松地提供内容。整个应用程序（和 web 服务）都在内存中，因此响应非常快。

这就是 清单 5 中的示例的主要用途：

清单 5. 使用计时器在新闻频发的活动期间提供即时更新服务

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <event.h>
#include <evhttp.h>

#define RELOAD_TIMEOUT 5
#define DEFAULT_FILE "sample.html"

char *filedata;
time_t lasttime = 0;
char filename[80];
int counter = 0;

void read_file()
{
  int size = 0;
  char *data;
  struct stat buf;

  stat(filename,&buf);

  if (buf.st_mtime > lasttime)
    {
      if (counter++)
        fprintf(stderr,"Reloading file: %s",filename);
      else
        fprintf(stderr,"Loading file: %s",filename);

      FILE *f = fopen(filename, "rb");
      if (f == NULL)
        {
          fprintf(stderr,"Couldn't open file\n");
          exit(1);
        }

      fseek(f, 0, SEEK_END);
      size = ftell(f);
      fseek(f, 0, SEEK_SET);
      data = (char *)malloc(size+1);
      fread(data, sizeof(char), size, f);
      filedata = (char *)malloc(size+1);
      strcpy(filedata,data);
      fclose(f);


      fprintf(stderr," (%d bytes)\n",size);
      lasttime = buf.st_mtime;
    }
}

void load_file()
{
  struct event *loadfile_event;
  struct timeval tv;

  read_file();

  RELOAD_TIMEOUT
```

```
  tv.tv_sec = RELOAD_TIMEOUT;
  tv.tv_usec = 0;

  loadfile_event = malloc(sizeof(struct event));

  evtimer_set(loadfile_event,
              load_file,
              loadfile_event);

  evtimer_add(loadfile_event,
              &tv);
}

void generic_request_handler(struct evhttp_request *req, void *arg)
{
  struct evbuffer *evb = evbuffer_new();

  evbuffer_add_printf(evb, "%s",filedata);
  evhttp_send_reply(req, HTTP_OK, "Client", evb);
  evbuffer_free(evb);
}

int main(int argc, char *argv[])
{
  short          http_port = 8081;
  char          *http_addr = "192.168.0.22";
  struct evhttp *http_server = NULL;

  if (argc > 1)
    {
      strcpy(filename,argv[1]);
      printf("Using %s\n",filename);
    }
  else
    {
      strcpy(filename,DEFAULT_FILE);
    }

  event_init();

  load_file();

  http_server = evhttp_start(http_addr, http_port);
  evhttp_set_gencb(http_server, generic_request_handler, NULL);

  fprintf(stderr, "Server started on port %d\n", http_port);
  event_dispatch();
}
```

这个服务器的基本原理与前面的示例相同。首先，脚本设置一个 HTTP 服务器，它只响应对基本 URL 主机/端口组合的请求（不处理请求 URI）。第一步是装载文件 (`read_file()`)。在装载最初的文件时和在计时器触发回调时都使用此函数。

`read_file()` 函数使用 `stat()` 函数调用检查文件的修改时间，只有在上一次装载之后修改了文件的情况下，它才重新读取文件的内容。此函数通过调用 `fread()` 装载文件数据，把数据复制到另一个结构中，然后使用 `strcpy()` 把数据从装载的字符串转移到全局字符串中。

`load_file()` 函数是触发计时器时调用的函数，它通过调用 `read_file()` 装载内容，然后

`load_file()` 函数是触发计时器时调用的函数。它通过调用 `read_file()` 装载内容，然后使用 RELOAD_TIMEOUT 值设置计时器，作为尝试装载文件之前的秒数。libevent 计时器使用 timeval 结构，允许按秒和毫秒指定计时器。计时器不是周期性的；当触发计时器事件时设置它，然后从事件队列中删除事件。

使用与前面的示例相同的格式编译代码：`$ gcc -o basichttpfile basichttpfile.c -levent`。

现在，创建作为数据使用的静态文件；默认文件是 sample.html，但是可以通过命令行上的第一个参数指定任何文件（见 清单 6）。

清单 6. 创建作为数据使用的静态文件

```
$ ./basichttpfile
Loading file: sample.html (8046 bytes)
Server started on port 8081
```

现在，程序可以接受请求了，重新装载计时器也启动了。如果修改 sample.html 的内容，应该会重新装载此文件并在日志中记录一个消息。例如，清单 7 中的输出显示初始装载和两次重新装载：

清单 7. 输出显示初始装载和两次重新装载

```
$ ./basichttpfile
Loading file: sample.html (8046 bytes)
Server started on port 8081
Reloading file: sample.html (8047 bytes)
Reloading file: sample.html (8048 bytes)
```

注意，要想获得最大的收益，必须确保环境没有限制打开的文件描述符数量。可以使用 ulimit 命令修改限制（需要适当的权限或根访问）。具体的设置取决与您的 OS，但是在 Linux® 上可以用 `-n` 选项设置打开的文件描述符（和网络套接字）的数量：

清单 8. 用 `-n` 选项设置打开的文件描述符数量

通过指定数字提高限制：`$ ulimit -n 20000`。

可以使用 Apache Bench 2 (ab2) 等性能基准测试应用程序检查服务器的性能。可以指定并发查询的数量以及请求的总数。例如，使用 100,000 个请求运行基准测试，并发请求数量为 1000 个：`$ ab2 -n 100000 -c 1000 http://192.168.0.22:8081/`。

使用服务器示例中所示的 8K 文件运行这个示例系统，获得的结果为大约每秒处理 11,000 个请求。请记住，这个 libevent 服务器在单一线程中运行，而且单一客户端不太可能给服务器造成压

力，因为它还受到打开请求的方法的限制。尽管如此，在交换的文档大小适中的情况下，这样的处理速率对于单线程应用程序来说仍然令人吃惊。

使用其他语言的实现

尽管 C 语言很适合许多系统应用程序，但是在现代环境中不经常使用 C 语言，脚本语言更灵活、更实用。幸运的是，Perl 和 PHP 等大多数脚本语言是用 C 编写的，所以可以通过扩展模块使用 libevent 等 C 库。

例如，清单 9 给出 Perl 网络服务器脚本的基本结构。`accept_callback()` 函数与 清单 1 所示核心 libevent 示例中的 accept 函数相同。

清单 9. Perl 网络服务器脚本的基本结构

```
my $server = IO::Socket::INET->new(
    LocalAddr       => 'localhost',
    LocalPort       => 8081,
    Proto           => 'tcp',
    ReuseAddr       => SO_REUSEADDR,
    Listen          => 1,
    Blocking        => 0,
    ) or die $@;

my $accept = event_new($server, EV_READ|EV_PERSIST, \&accept_callback);

$main->add;

event_mainloop();
```

用这些语言编写的 libevent 实现通常支持 libevent 系统的核心，但是不一定支持 HTTP 包装器。因此，对脚本编程的应用程序使用这些解决方案会比较复杂。有两种方法：要么把脚本语言嵌入到基于 C 的 libevent 应用程序中，要么使用基于脚本语言环境构建的众多 HTTP 实现之一。例如，Python 包含功能很强的 HTTP 服务器类 (httplib/httplib2)。

应该指出一点：在脚本语言中没有什么东西是无法用 C 重新实现的。但是，要考虑到开发时间的限制，而且与现有代码集成可能更重要。

libev 库

与 libevent 一样，libev 系统也是基于事件循环的系统，它在 `poll()`、`select()` 等机制的本机实现的基础上提供基于事件的循环。到我撰写本文时，libev 实现的开销更低，能够实现更好的基准测试结果。libev API 比较原始，没有 HTTP 包装器，但是 libev 支持在实现中内置更多事件类型。例如，一种 evstat 实现可以监视多个文件的属性变动，可以在 清单 4 所示的 HTTP 文件解决方案中使用它。

但是，libevent 和 libev 的基本过程是相同的。创建所需的网络监听套接字，注册在执行期间要调用的事件，然后启动主事件循环，让 libev 处理过程的其余部分。

例如，可以使用 Ruby 接口按照与清单 1 相似的方式提供回显服务器，见 清单 10。

清单 10. 使用 Ruby 接口提供回显服务器

```
require 'rubygems'
require 'rev'

PORT = 8081

class EchoServerConnection < Rev::TCPSocket
  def on_read(data)
    write 'You said: ' + data
  end
end

server = Rev::TCPServer.new('192.168.0.22', PORT, EchoServerConnection)
server.attach(Rev::Loop.default)

puts "Listening on localhost:#{PORT}"
Rev::Loop.default.run
```

Ruby 实现尤其出色，因为它为许多常用的网络解决方案提供了包装器，包括 HTTP 客户端、OpenSSL 和 DNS。其他脚本语言实现包括功能全面的 Perl 和 Python 实现，您可以试一试。

结束语

libevent 和 libev 都提供灵活且强大的环境，支持为处理服务器端或客户端请求实现高性能网络（和其他 I/O）接口。目标是以高效（CPU/RAM 使用量低）的方式支持数千甚至数万个连接。在本文中，您看到了一些示例，包括 libevent 中内置的 HTTP 服务，可以使用这些技术支持基于 IBM Cloud、EC2 或 AJAX 的 web 应用程序。

参考资料

学习

C10K problem 对处理 10,000 个连接的问题做了精彩的概述。

IBM Cloud Computing 网站提供不同云实现的相关信息。

阅读 系统管理工具包: 标准化您的 UNIX 命令行工具（Martin Brown，developerWorks，2006 年 5 月），学习如何跨多台机器使用相同的命令。

让 UNIX 和 Linux 一起工作（Martin Brown，developerWorks，2006 年 4 月）讲解如何让传统的 UNIX 发行版和 Linux 一起工作。

揭秘云计算（Brett McLaughlin，developerWorks，2009 年 3 月）：帮助您根据自己的应用程序需求选择最好的云计算平台。

阅读 用 Amazon Web Services 进行云计算（Prabhakar Chaganti，developerWorks，2008 年 7 月）：详细讲解如何使用 Amazon Web Services。

可以通过 developerWorks Cloud Computing Resource Center 使用适用于 Amazon EC2 平台的 IBM 产品。

developerWorks Cloud Computing Resource Center 使用适用于 Amazon EC2 平台的 IBM 产品。

在 developerWorks 的 云开发人员资源 中，发现和共享应用程序和服务开发人员构建其云部署项目的知识和经验。

AIX and UNIX 专区：developerWorks 的“AIX and UNIX 专区”提供了大量与 AIX 系统管理的所有方面相关的信息，您可以利用它们来扩展自己的 UNIX 技能。

AIX and UNIX 新手入门：访问“AIX and UNIX 新手入门”页面可了解更多关于 AIX 和 UNIX 的内容。

AIX and UNIX 专题汇总：AIX and UNIX 专区已经为您推出了很多的技术专题，为您总结了很多热门的知识点。我们在后面还会继续推出很多相关的热门专题给您，为了方便您的访问，我们在这里为您把本专区的所有专题进行汇总，让您更方便的找到您需要的内容。

AIX and UNIX 下载中心：在这里你可以下载到可以运行在 AIX 或者是 UNIX 系统上的 IBM 服务器软件以及工具，让您可以提前免费试用他们的强大功能。

IBM Systems Magazine for AIX 中文版：本杂志的内容更加关注于趋势和企业级架构应用方面的内容，同时对于新兴的技术、产品、应用方式等也有很深入的探讨。IBM Systems Magazine 的内容都是由十分资深的业内人士撰写的，包括 IBM 的合作伙伴、IBM 的主机工程师以及高级管理人员。所以，从这些内容中，您可以了解到更高层次的应用理念，让您在选择和应用 IBM 系统时有一个更好的认识。

在 developerWorks 播客 上收听面向软件开发人员的有趣访谈和讨论。

developerWorks 技术活动和网络广播：随时关注 developerWorks 技术活动和网络广播。

获得产品和技术

获取 libev 库，包括下载和文档。

获取 libevent 库。

ruby libev (rev) 库和文档。

Memcached 是用于存储和处理数据的 RAM 缓存（其核心使用 libevent，也可以使用其他 libevent 服务器）。

使用 IBM 试用软件 改进您的下一个开放源码开发项目，这些软件可以通过下载或从 DVD 获得。

讨论

参与 developerWorks 博客 并加入 developerWorks 社区。

加入 My developerWorks 中文社区

参与 AIX 和 UNIX® 论坛：

- AIX 论坛
- AIX for developers 论坛
- 集群系统管理
- IBM Support Assistant 论坛
- 性能工具论坛
- 虚拟化论坛
- 更多 AIX 和 UNIX 论坛

关于作者

Martin Brown 成为专业作家已有八年多的时间了。他是题材广泛的众多书籍和文章的作者。他的专业技术涉及各种开发语言和平台 — Perl、 Python、Java、JavaScript、Basic、Pascal、Modula-2、C、C++、Rebol、Gawk、 Shellscript、Windows、Solaris、Linux、BeOS、Mac OS/X 等等 — 还涉及 Web 编程、系统管理和集成。Martin 是 Microsoft 的主题专家（SME），并且是 ServerWatch.com、LinuxToday.com 和 IBM developerWorks 的定期投稿人，他还是 Computerworld、The Apple Blog 和其他站点的正式博客。您可以通过他的 Web 站点 http://www.mcslp.com 与他联络。

为本文评分

平均分 (41个评分)

评论