转 libevent入门

分类： windos network 2013-07-17 12:11 6247人阅读 评论(1) 收藏 举报

花了两天的时间在libevent上，想总结下，就以写简单tutorial的方式吧，貌似没有一篇简单的说明，让人马上就能上手用的。

首先给出官方文档吧： http://libevent.org ，首页有个Programming with Libevent，里面是一节一节的介绍libevent，但是感觉信息量太大了，而且还是英文的-。-（当然，如果想好好用libevent，看看还是很有必要的），还有个Reference，大致就是对各个版本的libevent使用doxgen生成的文档，用来查函数原型和基本用法什么的。

下面假定已经学习过基本的socket编程（socket,bind,listen,accept,connect,recv,send,close），并且对异步/callback有基本认识。

基本的socket编程是阻塞/同步的，每个操作除非已经完成或者出错才会返回，这样对于每一个请求，要使用一个线程或者单独的进程去处理，系统资源没法支撑大量的请求（所谓c10k problem?），例如内存：默认情况下每个线程需要占用2 ~ 8M的栈空间。posix定义了可以使用异步的select系统调用，但是因为其采用了轮询的方式来判断某个fd是否变成active，效率不高[O(n)]，连接数一多，也还是撑不住。于是各系统分别提出了基于异步/callback的系统调用，例如Linux的epoll，BSD的kqueue，Windows的IOCP。由于在内核层面做了支持，所以可以用O(1)的效率查找到active的fd。基本上，libevent就是对这些高效IO的封装，提供统一的API，简化开发。

libevent大概是这样的：

默认情况下是单线程的（可以配置成多线程，如果有需要的话），每个线程有且只有一个event_base，对应一个struct event_base结构体（以及附于其上的事件管理器），用来schedule托管给它的一系列event，可以和操作系统的进程管理类比，当然，要更简单一点。当一个事件发生后，event_base会在合适的时间（不一定是立即）去调用绑定在这个事件上的函数（传入一些预定义的参数，以及在绑定时指定的一个参数），直到这个函数执行完，再返回schedule其他事件。

```
//创建一个event_base
struct event_base *base = event_base_new();
assert(base != NULL);
```

event_base内部有一个循环，循环阻塞在epoll/kqueue等系统调用上，直到有一个/一些事件发生，然后去处理这些事件。当然，这些事件要被绑定在这个event_base上。每个事件对应一个struct event，可以是监听一个fd或者POSIX信号量之类（这里只讲fd了，其他的看manual吧）。struct event使用event_new来创建和绑定，使用event_add来启用：

```
//创建并绑定一个event
struct event *listen_event;
//参数：event_base, 监听的fd，事件类型及属性，绑定的回调函数，给回调函数的参数
listen_event = event_new(base, listener, EV_READ|EV_PERSIST, callback_func, (void*)
base);
//参数：event，超时时间(struct timeval *类型的，NULL表示无超时设置)
event_add(listen_event, NULL);
```

注：libevent支持的事件及属性包括(使用bitfield实现，所以要用 | 来让它们合体)
(a) EV_TIMEOUT: 超时
(b) EV_READ: 只要网络缓冲中还有数据，回调函数就会被触发
(c) EV_WRITE: 只要塞给网络缓冲的数据被写完，回调函数就会被触发
(d) EV_SIGNAL: POSIX信号量，参考manual吧

mysql主从库配置 (3321)
图像编码中的小白问题sp (3253)
nginx结构 (2775)

评论排行
ffmpeg移植改接口调用 | (5)
回调函数中调用类中的非 (2)
Windows 下使用Eclipse (2)
将h.264视频流封装成flv格 (2)
nginx结构 (2)
Pthreads on Microsoft W (1)
rtmp 时间戳问题 (1)
H264通过RTMP发布 V2. (1)
ffmpeg Windows下采集指 (1)
基于HTTP Live Streamir (1)

推荐文章
*在R中使用支持向量机（SVM）进行数据挖掘（上）
* 你不再需要动态网页——编辑-发布-开发分离
*Android性能优化之使用线程池处理异步任务
* Nginx初探
*编译器架构的王者LLVM——（6）多遍翻译的宏翻译系统
* 我的第一个Apple Watch小游戏——猜数字（Swift）

最新评论
libevent入门
chenqiai0: sockaddr_in 好像只有windows下有吧，那你这个不支持其他平台吗？
rtmp 时间戳问题
JAING: 嘿，好牛逼的文章。我做沙发啦。
回调函数中调用类中的非静态成
cugbin: @dragonno1:多线程环境下有解决方案吗？求解答。
BOOST库介绍，安装
itfanr: 虽然写得很乱，但是很赞！
ffmpeg移植改接口调用 |
Jeff-Li: 可以发一份demo吗？jefflee1314@163.com
nginx结构
ustcluoyuanhao: http://www.aosabook.org/en/nginx.html，应该注明原文出处，虽然翻...
CentOS 6 下升级安装Mysql 5.5
hzhxxx: 源码编译是王道啊！！
基于HTTP Live Streaming（HLS
shixingui: 在哪儿抄过来的吧？ffmpeg参数太多了
开源RTSP 流媒体服务器
spy32: VLC也算server？还有ffmpeg
如何用nginx+ffmpeg实现苹果HL
wlg0123: 可以提供下源码吗？wlg0123@126.com

(e) EV_PERSIST: 不指定这个属性的话，回调函数被触发后事件会被删除
(f) EV_ET: Edge-Trigger边缘触发，参考EPOLL_ET

然后需要启动event_base的循环，这样才能开始处理发生的事件。循环的启动使用event_base_dispatch，循环将一直持续，直到不再有需要关注的事件，或者是遇到event_loopbreak()/event_loopexit()函数。

```
//启动事件循环
event_base_dispatch(base);
```

这个函数是阻塞的，等待epoll事件，判断事件类型read/write/error，然后遍历事件队列，找到对该事件感兴趣的event_base，然后依次调用event_base中包含的回调函数！

接下来关注下绑定到event的回调函数callback_func：传递给它的是一个socket fd、一个event类型及属性bit_field、以及传递给event_new的最后一个参数（去上面几行回顾一下，把event_base给传进来了，实际上更多地是分配一个结构体，把相关的数据都撂进去，然后丢给event_new，在这里就能取得到了）。其原型是：

```
typedef void(* event_callback_fn)(evutil_socket_t sockfd, short event_type, void * arg)
```

对于一个服务器而言，上面的流程大概是这样组合的：
1. listener = socket()，bind()，listen()，设置nonblocking(POSIX系统中可使用fcntl设置，windows不需要设置，实际上libevent提供了统一的包装evutil_make_socket_nonblocking)
2. 创建一个event_base
3. 创建一个event，将该socket托管给event_base，指定要监听的事件类型，并绑定上相应的回调函数(及需要给它的参数)。对于listener socket来说，只需要监听EV_READ|EV_PERSIST
4. 启用该事件
5. 进入事件循环
---
6. (异步) 当有client发起请求的时候，调用该回调函数，进行处理。

问题：为什么不在listen完马上调用accept，获得客户端连接以后再丢给event_base呢？这个问题先想想噢。

之前的listen是一个fd,这里的accept后又是一个新的fd,服务器使用新的fd与客户端通信。
这里为什么不能在新的fd上直接调用recv/send???
libevent是单线程的，
如果此时调用recv/send会引起阻塞，
整个流程又变成阻塞同步式的啦。
所以这里新建一个event，设置其对read/write事件敏感，
然后加入到libevent有epoll框架中，
这样就又变回异步事件机制了。

回调函数要做什么事情呢？当然是处理client的请求了。首先要accept，获得一个可以与client通信的sockfd，然后……调用recv/send吗？错！大错特错！如果直接调用recv/send的话，这个线程就阻塞在这个地方了，如果这个客户端非常的阴险（比如一直不发消息，或者网络不好，老是丢包），libevent就只能等它，没法处理其他的请求了——所以应该创建一个新的event来托管这个sockfd。

在老版本libevent上的实现，比较罗嗦[如果不想详细了解的话，看下一部分]。
对于服务器希望先从client获取数据的情况，大致流程是这样的：
1. 将这个sockfd设置为nonblocking
2. 创建2个event:
   event_read，绑上sockfd的EV_READ|EV_PERSIST，设置回调函数和参数（后面提到的struct）
   event_write，绑上sockfd的EV_WRITE|EV_PERSIST，设置回调函数和参数（后面提到的struct）
3. 启用event_read事件
---
4. (异步) 等待event_read事件的发生，调用相应的回调函数。这里麻烦来了：回调函数用recv读入的数据，不能直接用send丢给sockfd了事——因为sockfd是nonblocking的，丢给它的话，不能保证正确（为什么呢？）。所以需要一个自己管理的缓存用来保存读入的数据中（在accept以后就创建一个struct，作为第2步回调函数的arg传进来），在合适的时间（比如遇到换行符）启用event_write事件【event_add(event_write, NULL)】，等待EV_WRITE事件的触发
---
5. (异步) 当event_write事件的回调函数被调用的时候，往sockfd写入数据，然后删除event_write事件【event_del(event_write)】，等待event_read事件的下一次执行。
以上步骤比较晦涩，具体代码可参考官方文档里面的【Example: A low-level ROT13 server with Libevent】

在读回调函数中，使用recv收到数据，
如果能判断协议的话，此时处理后，
就需要给对方一个回应，但是依然不能
使用send()直接发送，要不又就变成阻塞
同步式通信了。
所以libevent在这里引入了写缓冲区，
将需要发送的数据先放入写缓冲区中，
同时对fd注册写敏感事件和写回调函数，
当epoll检测到可写时，调用回调函数完成
真正的写操作！！！

我草为了搞成异步非阻塞操作，还真是麻烦！

由于需要自己管理缓冲区，且过程晦涩难懂，并且不兼容于Windows的IOCP，所以libevent2开始，提供了bufferevent这个神器，用来提供更加优雅、易用的API。struct bufferevent内建了两个event(read/write)和对应的缓冲区【struct evbuffer *input, *output】，并提供相应的函数用来操作缓冲区（或者直接操作bufferevent）。每当有数据被读入input的时候，read_cb函数被调用；每当output被输出完的时候，write_cb被调用；在网络IO操作出现错误的情况（连接中断、超时、其他错误），error_cb被调用。于是上一部分的步骤被简化为：
1. 设置sockfd为nonblocking
2. 使用bufferevent_socket_new创建一个struct bufferevent *bev，关联该sockfd，托管给event_base
3. 使用bufferevent_setcb(bev, read_cb, write_cb, error_cb, (void *)arg)将EV_READ/EV_WRITE对应的函数
4. 使用bufferevent_enable(bev, EV_READ|EV_WRITE|EV_PERSIST)来启用read/write事件

---

5. (异步)

在read_cb里面从input读取数据，处理完毕后塞到output里(会被自动写入到sockfd)

在write_cb里面（需要做什么吗？对于一个echo server来说，read_cb就足够了）

在error_cb里面处理遇到的错误

*. 可以使用bufferevent_set_timeouts(bev, struct timeval *READ, struct timeval *WRITE)来设置读写超时, 在error_cb里面处理超时。

*. read_cb和write_cb的原型是

void read_or_write_callback(struct bufferevent *bev, void *arg)

error_cb的原型是

void error_cb(struct bufferevent *bev, short error, void *arg) //这个是event的标准回调函数原型

可以从bev中用libevent的API提取出event_base、sockfd、input/output等相关数据，详情RTFM~

于是代码简化到只需要几行的read_cb和error_cb函数即可：

```
void read_cb(struct bufferevent *bev, void *arg) {
    char line[256];
    int n;
    evutil_socket_t fd = bufferevent_getfd(bev);
    while (n = bufferevent_read(bev, line, 256), n > 0)
        bufferevent_write(bev, line, n);
}

void error_cb(struct bufferevent *bev, short event, void *arg) {
    bufferevent_free(bev);
}
```

于是一个支持大并发量的echo server就成型了！下面附上无注释的echo server源码，110行，多抄几遍，就能完全弄懂啦！更复杂的例子参见官方文档里面的【Example: A simpler ROT13 server with Libevent】

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <assert.h>

#include <event2/event.h>
#include <event2/bufferevent.h>

#define LISTEN_PORT 9999
#define LISTEN_BACKLOG 32

void do_accept(evutil_socket_t listener, short event, void *arg);
void read_cb(struct bufferevent *bev, void *arg);
void error_cb(struct bufferevent *bev, short event, void *arg);
void write_cb(struct bufferevent *bev, void *arg);

int main(int argc, char *argv[])
{
    int ret;
    evutil_socket_t listener;
    listener = socket(AF_INET, SOCK_STREAM, 0);
    assert(listener > 0);
    evutil_make_listen_socket_reuseable(listener);

    struct sockaddr_in sin;
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = 0;
    sin.sin_port = htons(LISTEN_PORT);

    if (bind(listener, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return 1;
    }

    if (listen(listener, LISTEN_BACKLOG) < 0) {
        perror("listen");
        return 1;
    }

    printf ("Listening...\n");
```

这应该是使用setsockopt设置了REUSE标志位

```
    evutil_make_socket_nonblocking(listener);

    struct event_base *base = event_base_new();
    assert(base != NULL);
    struct event *listen_event;
    listen_event = event_new(base, listener, EV_READ|EV_PERSIST, do_accept, (void*)
base);
    event_add(listen_event, NULL);
    event_base_dispatch(base);

    printf("The End.");
    return 0;
}

void do_accept(evutil_socket_t listener, short event, void *arg)
{
    struct event_base *base = (struct event_base *)arg;
    evutil_socket_t fd;
    struct sockaddr_in sin;
    socklen_t slen;
    fd = accept(listener, (struct sockaddr *)&sin, &slen);
    if (fd < 0) {
        perror("accept");
        return;
    }
    if (fd > FD_SETSIZE) {
        perror("fd > FD_SETSIZE\n");
        return;
    }

    printf("ACCEPT: fd = %u\n", fd);

    struct bufferevent *bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FRE
E);
    bufferevent_setcb(bev, read_cb, NULL, error_cb, arg);
    bufferevent_enable(bev, EV_READ|EV_WRITE|EV_PERSIST);
}

void read_cb(struct bufferevent *bev, void *arg)
{
#define MAX_LINE    256
    char line[MAX_LINE+1];
    int n;
    evutil_socket_t fd = bufferevent_getfd(bev);

    while (n = bufferevent_read(bev, line, MAX_LINE), n > 0) {
        line[n] = '\0';
        printf("fd=%u, read line: %s\n", fd, line);

        bufferevent_write(bev, line, n);
    }
}

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

void error_cb(struct bufferevent *bev, short event, void *arg)
{
    evutil_socket_t fd = bufferevent_getfd(bev);
    printf("fd = %u, ", fd);
    if (event & BEV_EVENT_TIMEOUT) {
        printf("Timed out\n"); //if bufferevent_set_timeouts() called
    }
    else if (event & BEV_EVENT_EOF) {
        printf("connection closed\n");
    }
    else if (event & BEV_EVENT_ERROR) {
        printf("some other error\n");
    }
    bufferevent_free(bev);
}
```

所有的event必须依赖于一个event_base,感觉这个event_base就是一个事件容器。

声明fd对read事件敏感，当发生时执行回调do_accept()函数。

添加事件到libevent的事件队列/链表

开启事件分发，其中就是调用epoll等待事件发生

获取到一个新的fd用于与客户端通信

此处依然不能使用read/write来读写，避免又回到同步阻塞模式。
所以在新的fd上新建一个event，依然挂到之前的event_base上。
新建的event对read/write/error敏感，设置相应的回调函数。
然后由libevent的事件分发通过epoll阻塞等待事件的发生。

在回调函数中依然不能使用read()来读取，防止阻塞，
所以libevent提供了输入缓冲区，由它读入数据到缓冲区（在单独线程中）
然后再通过自定义的输入接口读写数据，阻止阻塞住，
写数据也不能直接调用write()防止阻塞，
而是通过libevent提供的自定义接口先将数据写入到libevent的输出缓冲区，
再由libevent适时执行真正的write()操作！

--

顶 0 踩 0

猜你在找

- Windows Server 2012 IIS 管理
- Windows Server 2012 Hyper-v 管理
- Windows Server 2012 组策略管理
- Windows Server 2012 存储和文件管理
- Windows Server 2012 R2 RMS 服务管理

查看评论

1楼 chenqiai0 2015-11-10 17:17发表

sockaddr_in 好像只有windows下有吧，那你这个不支持其他平台吗？

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap