socket异步编程--libevent的使用 - Simba Yang

这篇文章介绍下libevent在socket异步编程中的应用。在一些对性能要求较高的网络应用程序中，为了防止程序阻塞在socket I/O操作上造成程序性能的下降，需要使用异步编程，即程序准备好读写的函数(或接口)并向系统注册，然后在需要的时候只向系统提交读写的请求之后就继续做自己的事情，实际的读写操作由系统在合适的时候调用我们程序注册的接口进行。异步编程会给一些程序猿带来一些理解和编写上的困难，因为我们通常写的一些简单的程序都是顺序执行的，而异步编程将程序的执行顺序打乱了，有些代码什么情况下执行往往不是太清晰，因此也使得编程的复杂度大大增加。

Note：这里系统这个词使用的不准确，实际上可以是自己封装的异步调用机制，更常见的是一些可用的库，比如libevent,ACE等

想了解libevent的工作原理可以自行查询资料，网上相关的介绍一大堆，也可以自己阅读源码进行分析，本文仅从使用的角度做一个简单的介绍，看如何快速的将libevent引入我们的程序中。任何应用都免不了需要承载其功能的底层OS，libevent也不例外，其内部是通过封装操作系统的IO复用机制实现的，在linux系统上可能是epoll、kqueu之类的，取决于具体的OS所支持的IO复用方式，在我的系统上是epoll，因此可以理解为libevent提供了一个比epoll更为友好的操作接口，将程序猿从网络IO处理的细节中解放出来，使其可以专注于目标问题的处理上。

首先，安装libevent到任意目录下

```
wget http://monkey.org/~provos/libevent-1.4.13-stable.tar.gz
```

```
tar –xzvf libevent-1.4.13-stable.tar.gz
```

```
cd libevent-1.4.13-stable
```

```
./configure --prefix=/home/mydir/libevent
```

```
make && make install
```

现在假定我们要设计一个服务器程序，用于接收客户端的数据，并将接收的数据回写给客户端。下面来构造该程序，由于本仅仅是展示一个Demo，因此程序中将不对错误进行处理，假设所有的调用都成功

```
2 #define PORT 25341
3 #define BACKLOG 5
4 #define MEM_SIZE 1024
5
6 struct event_base* base;
```

```
7
8 int main(int argc, char* argv[])
9 {
10     struct sockaddr_in my_addr;
11     int sock;
12
13     sock = socket(AF_INET, SOCK_STREAM, 0);
14     int yes = 1;
15     setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
16     memset(&my_addr, 0, sizeof(my_addr));
17     my_addr.sin_family = AF_INET;
18     my_addr.sin_port = htons(PORT);
19     my_addr.sin_addr.s_addr = INADDR_ANY;
20     bind(sock, (struct sockaddr*)&my_addr, sizeof(struct sockaddr));
21     listen(sock, BACKLOG);
22
23     struct event listen_ev;
24     base = event_base_new();
25     event_set(&listen_ev, sock, EV_READIEV_PERSIST, on_accept, NULL);
26     event_base_set(base, &listen_ev);
27     event_add(&listen_ev, NULL);
28     event_base_dispatch(base);
29
30     return 0;
31 }
```

第13行说明创建的是一个TCP socket。第15行是服务器程序的通常做法，设置了该选项后，在父子进程模型中，当子进程为客户服务的时候如果父进程退出，可以重新启动程序完成服务的无缝升级，否则在所有父子进程完全退出前再启动程序会在该端口上绑定失败，也即不能完成无缝升级的操作(更多信息可以参考该函数说明或Steven先生的<网络编程>)。第24行用于创建一个事件处理的全局变量，可以理解为这是一个负责集中处理各种出入IO事件的总管家，它负责接收和派发所有输入输出IO事件的信息，这里调用的是函数event_base_new(), 很多程序里这里用的是event_init()，区别就是前者是线程安全的、而后者是非线程安全的，后者在其官方说明中已经被标志为过时的函数、且建议用前者代替，libevent中还有很多类似的函数，比如建议用event_base_dispatch代替event_dispatch，用event_assign代替event_set和event_base_set等，关于libevent接口的详细说明见其官方说明libevent_doc. 第25行说明在listen_en这个事件监听sock这个描述字的读操作，当读消息到达是调用on_accept函数，EV_PERSIST参数告诉系统持续的监听sock上的读事件，如果不加该参数，每次要监听该事件时就要重复的调用26行的event_add函数，从前面的代码可知，sock这个描述字是bind到本地的socket端口上，因此其对应的可读事件自然就是来自客户端的连接到达，我们就可以调用accept无阻塞的返回客户的连接了。第26行将listen_ev注册到base这个事件中，相当于告诉处理IO的管家请留意我的listen_ev上的事件。第27行相当于告诉处理IO的管家，当有我的事件到达时你发给我(调用on_accept函数)，至此对listen_ev的初始化完毕。第28行正式启动libevent的事件处理机制，使系统运行起来，运行程序的话会发现

event_base_dispatch是一个无限循环。

下面是on_accept函数的内容

```
1: void on_accept(int sock, short event, void* arg)
```

```
2: {
```

```
3:     struct sockaddr_in cli_addr;
```

```
4:     int newfd, sin_size;
```

```
5:     // read_ev must allocate from heap memory, otherwise the program would crash from segmant fault
```

```
6:     struct event* read_ev = (struct event*)malloc(sizeof(struct event));;
```

```
7:     sin_size = sizeof(struct sockaddr_in);
```

```
8:     newfd = accept(sock, (struct sockaddr*)&cli_addr, &sin_size);
```

```
9:     event_set(read_ev, newfd, EV_READ|EV_PERSIST, on_read, read_ev);
```

```
10:     event_base_set(base, read_ev);
```

```
11:     event_add(read_ev, NULL);
```

```
12: }
```

第9-12与前面main函数的24-26相同，即在代表客户的描述字newfd上监听可读事件，当有数据到达是调用on_read函数。这里有亮点需要注意，一是read_ev需要从堆里malloc出来，如果是在栈上分配，那么当函数返回时变量占用的内存会被释放，因此事件主循环event_base_dispatch会访问无效的内存而导致进程崩溃(即crash)；第二个要注意的是第9行read_ev作为参数传递给了on_read函数。

下面是on_read函数的内容

```
1: void on_read(int sock, short event, void* arg)
```

```
 2: {
 3:     struct event* write_ev;
 4:     int size;
 5:     char* buffer = (char*)malloc(MEM_SIZE);
 6:     bzero(buffer, MEM_SIZE);
 7:     size = recv(sock, buffer, MEM_SIZE, 0);
 8:     printf("receive data:%s, size:%d\n", buffer, size);
 9:     if (size == 0) {
10:         event_del((struct event*)arg);
11:         free((struct event*)arg);
12:         close(sock);
13:         return;
14:     }
15:     write_ev = (struct event*) malloc(sizeof(struct event));;
16:     event_set(write_ev, sock, EV_WRITE, on_write, buffer);
17:     event_base_set(base, write_ev);
18:     event_add(write_ev, NULL);
19: }
```

第9行，当从socket读返回0标志对方已经关闭了连接，因此这个时候就没必要继续监听该套接口上的事件，由于EV_READ在on_accept函数里是用EV_PERSIST参数注册的，因此要显示的调用event_del函数取消对该事件的监听。第18-21行与on_accept函数的6-11行类似，当可写时调用on_write函数，注意第19行将buffer作为参数传递给了on_write。这段程序还有比较严重的问题，后面进行说明。

on_write函数的实现

```
1 void on_write(int sock, short event, void* arg)
2 {
3     char* buffer = (char*)arg;
4     send(sock, buffer, strlen(buffer), 0);
5
6     free(buffer);
```

```
7 }
```

on_write函数中向客户端回写数据，然后释放on_read函数中malloc出来的buffer。在很多书合编程指导中都很强调资源的所有权，经常要求谁分配资源、就由谁释放资源，这样对资源的管理指责就更明确，不容易出问题，但是通过该例子我们发现在异步编程中资源的分配与释放往往是由不同的所有者操作的，因此也是比较容易出问题的地方。

其实在on_read函数中从socket读取数据后程序就可以直接调用write/send接口向客户回写数据了，因为写事件已经满足，不存在异步不异步的问题，这里进行on_write的异步操作仅仅是为了说明异步编程中资源的管理与释放的问题，另外一方面，直接调用write/send函数向客户端写数据可能导致程序较长时间阻塞在IO操作上，比如socket的输出缓冲区已满，则write/send操作阻塞到有可用的缓冲区之后才能进行实际的写操作，而通过向写事件注册on_accept函数，那么libevent会在合适的时间调用我们的callback函数，(比如对于会引起IO阻塞的情况比如socket输出缓冲区满，则由libevent设计算法来处理，如此当回调on_accept函数时我们在调用IO操作就不会发生真正的IO之外的阻塞)。注：前面括号中是我个人认为一个库应该实现的功能，至于libevent是不是实现这样的功能并不清楚也无意深究。

再来看看前面提到的on_read函数中存在的问题，首先write_ev是动态分配的内存，但是没有释放，因此存在内存泄漏，另外，on_read中进行malloc操作，那么当多次调用该函数的时候就会造成内存的多次泄漏。这里的解决方法是对socket的描述字可以封装一个结构体来保护读、写的事件以及数据缓冲区，整理后的完整代码如下

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <stdio.h>

#include <event.h>
```

```
#define PORT        25341
#define BACKLOG     5
#define MEM_SIZE    1024

struct event_base* base;
struct sock_ev {
    struct event* read_ev;
    struct event* write_ev;
    char* buffer;
};

void release_sock_event(struct sock_ev* ev)
{
    event_del(ev->read_ev);
    free(ev->read_ev);
    free(ev->write_ev);
    free(ev->buffer);
    free(ev);
}

void on_write(int sock, short event, void* arg)
{
    char* buffer = (char*)arg;
    send(sock, buffer, strlen(buffer), 0);

    free(buffer);
}

void on_read(int sock, short event, void* arg)
{
    struct event* write_ev;
    int size;
    struct sock_ev* ev = (struct sock_ev*)arg;
    ev->buffer = (char*)malloc(MEM_SIZE);
    bzero(ev->buffer, MEM_SIZE);
    size = recv(sock, ev->buffer, MEM_SIZE, 0);
    printf("receive data:%s, size:%d\n", ev->buffer, size);
    if (size == 0) {
        release_sock_event(ev);
        close(sock);
```

```
        return;
    }
    event_set(ev->write_ev, sock, EV_WRITE, on_write, ev->buffer);
    event_base_set(base, ev->write_ev);
    event_add(ev->write_ev, NULL);
}

void on_accept(int sock, short event, void* arg)
{
    struct sockaddr_in cli_addr;
    int newfd, sin_size;
    struct sock_ev* ev = (struct sock_ev*)malloc(sizeof(struct sock_ev));
    ev->read_ev = (struct event*)malloc(sizeof(struct event));
    ev->write_ev = (struct event*)malloc(sizeof(struct event));
    sin_size = sizeof(struct sockaddr_in);
    newfd = accept(sock, (struct sockaddr*)&cli_addr, &sin_size);
    event_set(ev->read_ev, newfd, EV_READIEV_PERSIST, on_read, ev);
    event_base_set(base, ev->read_ev);
    event_add(ev->read_ev, NULL);
}

int main(int argc, char* argv[])
{
    struct sockaddr_in my_addr;
    int sock;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    int yes = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(PORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bind(sock, (struct sockaddr*)&my_addr, sizeof(struct sockaddr));
    listen(sock, BACKLOG);

    struct event listen_ev;
    base = event_base_new();
    event_set(&listen_ev, sock, EV_READIEV_PERSIST, on_accept, NULL);
    event_base_set(base, &listen_ev);
    event_add(&listen_ev, NULL);
    event_base_dispatch(base);
```

```
    return 0;
```

```
}
```

程序编译的时候要加 -levent 连接选项，以连接libevent的共享库，但是执行的时候依然爆出如下错误：error while loading shared libraries: libevent-1.4.so.2: cannot open shared object file: No such file or directory， 这个是程序找不到共享库的位置，通过执行echo $LD_LIBRARY_PATH可以看到系统库的环境变量里没有我们安装的路径，即由--prefix制定的路径，执行export LD_LIBRARY_PATH=/home/mydir/libevent/lib/:$LD_LIBRARY_PATH将该路径加入系统环境变量里，再执行程序就可以了。

测试结果