Libevent 简介

Libevent是一款事件驱动的网络开发包，由于采用C语言开发体积小巧，跨平台，速度极快。大量开源项目使用了Libevent比如谷歌的浏览器和分布式的高速缓存系统memcached。libevent支持kqueue,select,poll,epoll,iocp。内部事件机制完全独立于公开事件API，libevent支持跨平台可以在Linux，*BSD，MacOSX,Solaris,Windows等平台上编译。

学习条件：具有一定的C/C++基础，熟悉Linux

环境搭建

- [x] 配置zlib库

```
# 1. 解压zlib 1.2.11
tar xvf zlib-1.2.11.tar.gz
# 2. 编译
cd zlib-1.2.11/
./configure
make
make install
```

- [x] 配置openssl库

```
# 1. 解压openssl-1.1.1.tar.gz
tar xvf openssl-1.1.1.tar.gz
# 2. 编译
cd openssl-1.1.1/
./configure
make
make install
```

- [x] 配置libevent环境

```
# 1.加压liebevent 2.1.8
unzip libevent-master.zip
# 2. 编译
cd libevent-master/
./autogen.sh
./configure
make
make install
# 3.将动态路来连接到 /usr/lib 下或者执行以下  ldconfig
sudo ln -s /usr/local/lib/libevent-2.2.so.1 /usr/lib/libevent-2.2.so.1
```

实战实例

创建`event_base`

仅仅实现创建上下文

```
/***
 *  创建event base
 * */

#include <event2/event.h>
#include <iostream>
using namespace std;
int main()
{

    std::cout << "test libevent!\n";
    //创建libevent的上下文
    event_base * base = event_base_new();
    if (base)
    {
        cout << "event_base_new success!" << endl;
    }
    return 0;
}
```

创建`test_server`

`test_server`中说明了如何使用`libevent`创建一个`socket`监听

`evconnlistener_new_bind`一个接口完成了`socket`的创建，绑定和监听。

```
/***
 *  创建event base
 * */

#include <event2/event.h>
#include <iostream>
#include <signal.h>
#include <event2/listener.h>
#include <string.h>
#include "event_interface.h"

using namespace std;

/**
   A callback that we invoke when a listener has a new connection.

   @param listener The evconnlistener
   @param fd The new file descriptor
   @param addr The source address of the connection
   @param socklen The length of addr
```

```
   @param user_arg the pointer passed to evconnlistener_new()
 */
void listen_cb(struct evconnlistener * evConnListener, evutil_socket_t
evUtilSockFd, struct sockaddr * sockAddr, int socklen, void *data)
{
    cout << "listen cb is called" << endl;
}

int main(int argc, char *argv[])
{
    //1. 忽略管道信号,发送数据给已关闭的socket
    //一些socket程序莫名宕掉的原因
    if(signal(SIGPIPE, SIG_IGN) == SIG_ERR)
    {
        cout << "ignal pipe signal" << endl;
    }

    std::cout << "test libevent!\n";
    //创建libevent的上下文
    event_base * base = event_base_new();
    if (!base)
    {
        cout << "event_base_new failed." << endl;
        return -1;
    }
    else
    {
        cout << "event_base_new success!" << endl;
    }

    //监听端口
    //socket, bind, listen
    sockaddr_in sockIn;
    memset(&sockIn, 0, sizeof(sockIn));
    sockIn.sin_family = AF_INET;
    sockIn.sin_port = htons(SERVER_PORT);
    /* 地址没有指定因为对sockIn进行了了memset, 地址赋值为0代表着可以为任意可以用的地址 */

    struct evconnlistener *pEvListener = evconnlistener_new_bind(base, /*
libevent的上下文 */
                listen_cb, /* 接收到连接的回调 */
                base, /* 回调函数参数 */
                LEV_OPT_REUSEABLE|LEV_OPT_CLOSE_ON_FREE, /* 地址重用,
evconnlistenner关闭同时关闭socket */
                10, /* 连接队列的大小, 对应的listen函数 */
                (sockaddr *)&sockIn,    /* 绑定地址和端口 */
                sizeof(sockIn)
                );

    //事件分发处理
    if(base)
        event_base_dispatch(base);

    if(pEvListener)
        evconnlistener_free(pEvListener);

    if(base)
    event_base_free(base);
```

```
    return 0;
}
```

创建`test_conf`

`test_conf`主要是实现了，测试当前系统中支持的方法类型和事件特征的支持情况。

```
support methods
epoll
poll
select

EV_FEATURE_ET events are supported.
EV_FEATURE_O1 events are supported.
EV_FEATURE_FDS events are not supports.
EV_FEATURE_EARLY_CLOSE events are supported.
event base new with config sucess
```

```
#include <event2/event.h>
#include <event2/thread.h>
#include <event2/listener.h>
#include <signal.h>
#include <iostream>
#include <string.h>
#include "event_interface.h"

using namespace std;

int main()
{
    //忽略管道信号，发送数据给已关闭的socket
    if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        return 1;

    //创建配置上下文
    event_config *config = event_config_new();
    //显示支持的网络模式
    const char **methods =  event_get_supported_methods();
    cout << "support methods " << endl;
    for(int i = 0; methods[i] != NULL; i++)
    {
        cout << methods[i] << endl;
    }
    //设置特征，确认特征时候生效
    //这个features在linux中设置没有效果，因为linux中本来就是支持ET模式的，边缘触发模式
    // 设置了EV_FEATURE_FDS其他特征嗯就无法设置
    //也就是所支持了EV_FEATURE_FDS  其他的特征都是无法支持的
    int ret =
    event_config_require_features(config, EV_FEATURE_ET|EV_FEATURE_EARLY_CLOSE);
    if(OK != ret)
```

```
    {
        cerr << "event config require features failed." << endl;
        return ERROR;
    }
    //初始化libevent上下文
    event_base *base =  event_base_new_with_config(config);

    //config一旦配置好就不需要在使用了
    event_config_free(config);

    if(!base)
    {
        cerr << "event base new with config failed!" << endl;
        //首次失败就创建一个base取默认值，若是再次失败就返回失败
        base = event_base_new();
        if(!base)
        {
            cerr << "event base new failed." << endl;
            return ERROR;
        }
    }
    else
    {
        //确认特征那些生效
        int f = event_base_get_features(base);
        if(f&EV_FEATURE_ET)
        {
            cout << "EV_FEATURE_ET events are supported." << endl;
        }
        else
        {
            cout << "EV_FEATURE_ET events are not supports." << endl;
        }

        if(f&EV_FEATURE_O1)
        {
            cout << "EV_FEATURE_O1 events are supported." << endl;
        }
        else
        {
            cout << "EV_FEATURE_O1 events are not supports." << endl;
        }

        if(f&EV_FEATURE_FDS)
        {
            cout << "EV_FEATURE_FDS events are supported." << endl;
        }
        else
        {
            cout << "EV_FEATURE_FDS events are not supports." << endl;
        }

        if(f&EV_FEATURE_EARLY_CLOSE)
        {
            cout << "EV_FEATURE_EARLY_CLOSE events are supported." << endl;
        }
        else
        {
```

```
            cout << "EV_FEATURE_EARLY_CLOSE events are not supports." << endl;
        }
        cout << "event base new with config sucess" << endl;
        event_base_free(base);
    }

    return 0;
}
```