libevent简介.md

# 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

* 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

* 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

* 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;  
}