#include <assert.h>

#include "sys/socket.h"

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include "EventLoop.h"

// 初始化

struct EventLoop\* eventLoopInit()

{

return eventLoopInitEx(NULL);

}

// 写数据

void takeWakeup(struct EventLoop\* evLoop)

{

const char\* msg = "我要当上海贼王！！！";

write(evLoop->socketPair[0], msg, strlen(msg));

}

// 读数据 ---- 目的不是读数据，而是触发读事件， 就能对epoll\_wait解除阻塞

int readLocalMessage(void\* arg)

{

struct EventLoop\* evLoop = (struct EventLoop\*)arg;

char buf[256];

read(evLoop->socketPair[1], buf, sizeof(buf));

return 0;

}

struct EventLoop\* eventLoopInitEx(const char\* threadName)

{

struct EventLoop\* evLoop = (struct EventLoop\*)malloc(sizeof(struct EventLoop));

evLoop->isQuit = false;

evLoop->threadID = pthread\_self();

pthread\_mutex\_init(&evLoop->mutex, NULL);

strcpy(evLoop->threadName, threadName == NULL ? "MainThread" : threadName);

evLoop->dispatcher = &EpollDispatcher;

evLoop->dispatcherData = evLoop->dispatcher->init();

// 链表初始化

evLoop->head = evLoop->tail = NULL;

// map

evLoop->channelMap = channelMapInit(128);

// 增加一个主线程唤醒子线程的 文件描述符

// AF\_UNIX表示本地通信 evLoop->socketPair传出参数，包含两个文件描述符用于本地通信

int ret = socketpair(AF\_UNIX, SOCK\_STREAM, 0, evLoop->socketPair);

if (ret == -1)

{

perror("socketpair");

exit(0);

}

// 制定规则：evLoop->socketPair【0】发数据 【1】接收数据

struct Channel\* channel = channelInit(evLoop->socketPair[1], ReadEvent, readLocalMessage, NULL, NULL, evLoop);

// 将channel添加到任务队列中

eventLoopAddTask(evLoop, channel, ADD);

return evLoop;

}

// 启动反应堆模型

int eventLoopRun(struct EventLoop\* evLoop)

{

// 断言，false 终止并输出错误信息

assert(evLoop != NULL);

// 取出事件分发和检测模型

struct Dispatcher\* dispatcher = evLoop->dispatcher;

// 比较线程ID是否正常

if (evLoop->threadID != pthread\_self())

{

return -1;

}

// 循环进行事件处理

while (!evLoop->isQuit)

{

// dispatch 检测文件描述符 并 处理读写事件（两条路径

dispatcher->dispatch(evLoop, 2);

eventLoopProcessTask(evLoop);

}

return 0;

}

// 处理激活的文件fd

int eventActivate(struct EventLoop\* evLoop, int fd, int event)

{

if (fd < 0 || evLoop == NULL)

{

return -1;

}

// 取出channel

struct Channel\* channel = evLoop->channelMap->list[fd];

assert(channel->fd == fd);

if (event & ReadEvent && channel->readCallback)

{

channel->readCallback(channel->arg);

}

if (event & WriteEvent && channel->writeCallback)

{

channel->writeCallback(channel->arg);

}

return 0;

}

// 添加任务到任务队列

int eventLoopAddTask(struct EventLoop\* evLoop, struct Channel\* channel, int type)

{

// 加锁 ， 保护共享资源

pthread\_mutex\_lock(&evLoop->mutex);

// 创建新节点

struct ChannelElement\* node = (struct ChannelElement\*)malloc(sizeof(struct ChannelElement));

node->channel = channel;

node->type = type;

node->next = NULL;

// 链表为空

if (evLoop->head == NULL)

{

evLoop->head = evLoop->tail = node;

}

else

{

evLoop->tail->next = node;

evLoop->tail = node;

}

pthread\_mutex\_unlock(&evLoop->mutex);

// 处理节点

/\* 前提条件，当前eventLopp属于子线程

细节： 1. 对于链表节点的添加，可能是当前线程也可能是其它线程（主线程

1）修改fd事件，当前子线程发起，当前子线程处理

2）添加新的fd，添加任务节点的操作由主线程发起

2. 不能让主线程处理任务队列，需要由当前的子线程去处理

\*/

if (evLoop->threadID == pthread\_self())

{

// 当前子线程

eventLoopProcessTask(evLoop);

}

else

{

// 主线程 -- 告诉子线程处理任务队列中的任务

// 1. 子线程在工作， 2.子线程被阻塞了

takeWakeup(evLoop);

}

return 0;

}

// 处理任务队列中的任务

int eventLoopProcessTask(struct EventLoop\* evLoop)

{

pthread\_mutex\_lock(&evLoop->mutex);

// 取出头节点

struct ChannelElement\* head = evLoop->head;

while (head != NULL)

{

struct Channel\* channel = head->channel;

if (head->type == ADD)

{

// 添加

eventLoopAdd(evLoop, channel);

}

else if (head->type == DELETE)

{

// 删除

eventLoopRemove(evLoop, channel);

}

else if (head->type == MODIFY)

{

// 修改

eventLoopModify(evLoop, channel);

}

struct ChannelElement\* tmp = head;

head = head->next;

free(tmp);

}

evLoop->head = evLoop->tail = NULL;

pthread\_mutex\_unlock(&evLoop->mutex);

return 0;

}

// 处理dispatcher中的节点

int eventLoopAdd(struct EventLoop\* evLoop, struct Channel\* channel)

{

int fd = channel->fd;

struct ChannelMap\* channelMap = evLoop->channelMap;

if (fd >= channelMap->size)

{

// 没有足够的空间存储键值对 fd - channel ==》扩容

if (!makeMapRoom(channelMap, fd, sizeof(struct Channel\*)));

{

return -1;

}

}

// 找到fd'对应的数组元素位置，并存储

if (channelMap->list[fd] == NULL)

{

channelMap->list[fd] == channel;

evLoop->dispatcher->add(channel, evLoop);

}

return 0;

}

int eventLoopRemove(struct EventLoop\* evLoop, struct Channel\* channel)

{

int fd = channel->fd;

struct ChannelMap\* channelMap = evLoop->channelMap;

if (fd >= channelMap->size)

{

return -1;

}

return evLoop->dispatcher->remove(channel, evLoop);

}

int eventLoopModify(struct EventLoop\* evLoop, struct Channel\* channel)

{

int fd = channel->fd;

struct ChannelMap\* channelMap = evLoop->channelMap;

if (channelMap->list[fd] == NULL)

{

return -1;

}

return evLoop->dispatcher->modify(channel, evLoop);

}

// 释放channel

int destroyChannel(struct EventLoop\* evLoop, struct Channel\* channel)

{

// 删除channel 和 fd的对应关系

evLoop->channelMap->list[channel->fd] = NULL;

// 关闭fd

close(channel->fd);

// 释放channel

free(channel);

return 0;

}