Connection.h：

Connection();

构造函数：构造函数内所做的事情就是mysql进行连接的一个初始化的过程。

```
    // 初始化数据库连接
Connection();


Connection::Connection()
{
    _conn = mysql_init(nullptr);
}
```

~Connection();

析构函数：构造函数内所做的事情就是mysql进行关闭连接的一个过程。

```
// 释放数据库连接资源
~Connection();

Connection::~Connection()
{
    if(_conn != nullptr)
        mysql_close(_conn);
}
```

bool connect(string ip, unsigned short port, string user, string password, string dbname);

对数据库进行一个连接，以及mysql_real_connect（）

```
// 连接数据库
bool connect(string ip, unsigned short port, string user, string password,
             string dbname);


bool Connection::connect(string ip, unsigned short port, string user, string
password,
                 string dbname)
{
    MYSQL *p = mysql_real_connect(_conn, ip.c_str(), user.c_str(),
                                  password.c_str(), dbname.c_str(), port,
nullptr, 0);
    return p != nullptr;
}
```

bool update(string sql);

更新操作，进行插入删除以及跟新数据的一个过程。

```
// 更新操作 insert、delete、update
bool update(string sql);

bool Connection::update(string sql)
{
    // 更新操作， insert delete update
    if (mysql_query(_conn, sql.c_str()))
    {
        LOG("更新失败:" + sql);
        return false;
    }
    return true;
}
```

MYSQL_RES *query(string sql);

查询操作 select，对数据库及进行一个查询。

```
// 查询操作 select
MYSQL_RES *query(string sql);

MYSQL_RES* Connection::query(string sql)
{
    //查询操作 select
    if (mysql_query(_conn, sql.c_str()))
    {
        LOG("查询失败:" + sql);
        return nullptr;
    }
    return mysql_use_result(_conn);
}
```

refreshAliveTime()

刷新一下连接的起始的空闲时间点

```
// 刷新一下连接的起始的空闲时间点
void refreshAliveTime(){_alivetime = clock();};
```

getAliveTime()

```
 clock_t getAliveTime() const{return clock() - _alivetime;};
```

两个参数：

```
private:
    MYSQL *_conn;   //表示和MySQL Server的一条连接
    clock_t _alivetime; //记录进入空闲状态后的其实存活时间
```

CommonConnectionPool.h(单例模式)

static ConnectionPool* getConnectionPool();

获取连接池对象实例

```
//获取连接池对象实例
static ConnectionPool* getConnectionPool();

ConnectionPool* ConnectionPool::getConnectionPool()
{
    static ConnectionPool pool; //lock和unlock
    return &pool;
}
```

shared_ptr<Connection> getConnection();

给外部提供接口，从连接池中获取一个可用的空闲链接

```
//给外部提供接口，从连接池中获取一个可用的空闲链接
shared_ptr<Connection> getConnection();

//给外部提供接口，从连接池中获取一个可用的空闲连接
shared_ptr<Connection> ConnectionPool::getConnection()
{
    unique_lock<mutex> lock(_queueMutex);
    while(_connectionQue.empty())
    {
        //sleep
        if(cv_status::timeout == cv.wait_for(lock,
chrono::milliseconds(_connectionTimeout)))
        {
            if(_connectionQue.empty())
            {
                LOG("获取空闲链接超时...获取连接失败");
                return nullptr;
            }
        }
    }
    /*
    shared_ptr智能指针析构时，会把connection资源直接delete掉，相当于调用connection的析构
函数
    connection就被close掉了
    这里需要自定义shared_ptr的释放资源的方式，把connnection直接归还到queue当中
    */
    shared_ptr<Connection> sp(_connectionQue.front(),[&](Connection *pcon){
        //这是在服务器应用线程中调用的，所以一定要考虑队列的线程安全操作
        unique_lock<mutex> lock(_queueMutex);
        pcon->refreshAliveTime();
        _connectionQue.push(pcon);
    });
```

```
    //这种写法主要用于资源管理场景，特别是当你需要在 shared_ptr 被销毁时执行一些自定义的清理或资源再利用操作。
//在这种情况下，Connection 对象被拿出来使用，当它不再需要时，通过自定义删除器，将其重新放回连接池队列中，而不是直接销毁。这保证了连接的复用，同时确保了多线程环境下的安全性。


    _connectionQue.pop();
    //消费完连接以后，通知生产者线程检查一下，如果队列为空了，赶紧进行生产
    cv.notify_all();
    return sp;
}
```

bool loadConfigFile();

从配置文件中加载配置项

```
bool ConnectionPool::loadConfigFile()
{
    FILE *pf = fopen("/home/fdl/桌面/ConnectionPool/src/mysql.conf", "r");
    if(pf == NULL)
    {
        LOG("mysql.init file is not exist!");
        return false;
    }

    while(!feof(pf))
    {
        char line[1024] = {0};
        fgets(line,1024,pf);
        string str = line;
        int idx = str.find('=',0);
        if(idx == -1) //无效的配置项
        {
            continue;
        }
        int endidx = str.find('\n',idx);
        string key = str.substr(0,idx);
        string value = str.substr(idx+1,endidx - idx -1);

        if(key == "ip")
        {
            _ip = value;
        }else if(key == "port")
        {
            _port = atoi(value.c_str());
        }else if(key == "username")
        {
            _username = value;
        }else if(key == "dbname"){
            _dbname = value;
        }
        else if(key == "password")
        {
            _password = value;
        }else if(key ==  "initSize")
```

```
        {
            _initSize = atoi(value.c_str());
        }else if(key == "maxSize")
        {
            _maxSize = atoi(value.c_str());
        }else if(key == "maxIdleTime")
        {
            _maxIdleTime = atoi(value.c_str());
        }else if(key == "connectionTimeout")
        {
            _connectionTimeout = atoi(value.c_str());
        }

        cout << endl;
    }
    return true;
}


#数据库连接池的配置文件
ip=127.0.0.1
port=3306
username=root
password=root
dbname=user
initSize=50
maxSize=1024
#最大空闲时间默认单位是秒
maxIdleTime=10
#连接超时最大时间ms
connectionTimeOut=100
```

ConnectionPool（）

构造函数，就是对于一些对象的值从配置中读取

```
ConnectionPool::ConnectionPool()
{
    //加载配置项
    if(!loadConfigFile())
    {
        return;
    }
    //创建初始数量的连接
    for(int i =0; i < _initSize; ++i)
    {
        Connection *p = new Connection();
        p->connect(_ip, _port,_username,_password,_dbname);
        p->refreshAliveTime(); //刷新一下开始空闲的起始时间
        _connectionQue.push(p);
        _connectionCnt++;
    }

    //启动一个新的线程，作为一个连接的生产者
    thread produce(std::bind(&ConnectionPool::produceConnectionTask,this));
```

```
    produce.detach();

    //启动一个新的定时线程，扫描多余的空闲连接，超过maxIdleTime时间的空闲连接，进行多余的连接回收
    thread scanner(std::bind(&ConnectionPool::scannerConnectionTask,this));
    scanner.detach();
}
```

produceConnectionTask();

运行在独立的线程中，专门负责生产新链接

```
//运行在独立的线程中，专门负责生产新链接
void produceConnectionTask();

void ConnectionPool::produceConnectionTask()
{
    for(;;)
    {
        unique_lock<mutex> lock(_queueMutex);
        while(!_connectionQue.empty())
        {
            cv.wait(lock);//队列不空，此处生产线程进入等待状态
        }
        //连接数量到达上限，继续创建新的连接
        if(_connectionCnt <= _maxSize)
        {
            Connection *p = new Connection();
            p->connect(_ip, _port,_username,_password,_dbname);
            p->refreshAliveTime();
            _connectionQue.push(p);
            _connectionCnt++;
        }
        //通知消费者线程，可以消费连接了
        cv.notify_all();
    }
}
```

scannerConnectionTask()

```
//扫描多余的空闲连接，超过maxIdleTime时间的空闲连接，进行多余的连接回收
void ConnectionPool::scannerConnectionTask()
{
    for(;;)
    {
        //通过sleep模拟定时效果
        this_thread::sleep_for(chrono::seconds(_maxIdleTime));
        //扫描整个队列，释放多余的连接
        unique_lock<mutex> lock(_queueMutex);
        while(_connectionCnt > _initSize)
        {
            Connection *p = _connectionQue.front();
            if(p->getAliveTime() >= (_maxIdleTime*1000))
            {
```

```
                _connectionQue.pop();
                _connectionCnt--;
                delete p; //调用~Connection()释放连接
            }else{
                break;//队头的连接没有超过_maxIdleTime,其他连接肯定也没有超过
            }
        }
    }
}
```

参数:

```
    string _ip; //mysql的ip地址
    unsigned short _port;   //mysql的端口号 3306
    string _username;   //mysql登陆用户名
    string _password;   //mysql登录密码
    string _dbname;     //数据库名字（不是数据表）
    int _initSize;      //连接池的初始连接量
    int _maxSize;       //连接池的最大连接量
    int _maxIdleTime;   //连接池的最大空闲时间
    int _connectionTimeout; //连接池获取连接的超时时间

    queue<Connection*> _connectionQue; //存储mysql连接的队列
    mutex _queueMutex; //维护连接队列的线程安全互斥锁
    atomic_int _connectionCnt; //记录连接所创建的connection连接的总数量
    condition_variable cv; //设置条件变量，用于连接生产线程和消费线程的通信
```

main.cpp

```
#include <iostream>
using namespace std;
#include "Connection.h"
#include "CommonConnectionPool.h"

int main()
{
    // 数据库操作
    //  Connection conn;
    //  string sql = "insert into user (name,age,sex) values('John',18,'male')";
    //  //sql语句的组装方法
    //  // char sql[1024] = {0};
    //  // sprintf(sql, "insert into user(name, age, sex) values('%s', %d,
'%s')",
    //  //         "zhangsan",20,"male");
    //  conn.connect("127.0.0.1",3306,"root","root","user");
    //  conn.update(sql);

    // ConnectionPool *cp = ConnectionPool::getConnectionPool();
    // cp->loadConfigFile();

    //单线程插入数据
    // clock_t begin = clock();
    // ConnectionPool *cp = ConnectionPool::getConnectionPool();
```

```
    // for (int i = 0; i < 1000; ++i)
    // {
    //     //Connection conn;
    //     char sql[1024] = {0};
    //     sprintf(sql, "insert into user(name, age, sex) values('%s', %d,
'%s')",
    //             "zhangsan", 20, "male");
    //     // conn.connect("127.0.0.1", 3306, "root", "root", "user");
    //     // conn.update(sql);
    //     //ConnectionPool *cp = ConnectionPool::getConnectionPool();
    //     shared_ptr<Connection> sp = cp -> getConnection();
    //     sp->update(sql);
    // }
    // clock_t end = clock();
    // cout<< (end - begin) << "ms" << endl;



    //多线程插入数据
    clock_t begin = clock();
    ConnectionPool *cp = ConnectionPool::getConnectionPool();
    thread t1([&](){
        for (int i = 0; i < 250; ++i)
        {
            //Connection conn;
            char sql[1024] = {0};
            sprintf(sql, "insert into user(name, age, sex) values('%s', %d,
'%s')",
                    "zhangsan", 20, "male");
            shared_ptr<Connection> sp = cp -> getConnection();
            sp->update(sql);
        }
    });
    thread t2([&](){
        for(int i = 0; i < 250; ++i)
        {
            //Connection conn;
            char sql[1024] = {0};
            sprintf(sql, "insert into user(name, age, sex) values('%s', %d,
'%s')",
                    "zhangsan", 20, "male");
            shared_ptr<Connection> sp = cp -> getConnection();
            sp->update(sql);
        }
    });
    thread t3([&](){
        for (int i = 0; i < 250; ++i)
        {
            //Connection conn;
            char sql[1024] = {0};
            sprintf(sql, "insert into user(name, age, sex) values('%s', %d,
'%s')",
                    "zhangsan", 20, "male");
            shared_ptr<Connection> sp = cp -> getConnection();
            sp->update(sql);
        }
```

```
    });
    thread t4([&](){
        for (int i = 0; i < 250; ++i)
        {
            //Connection conn;
            char sql[1024] = {0};
            sprintf(sql, "insert into user(name, age, sex) values('%s', %d,
'%s')",
                    "zhangsan", 20, "male");
            shared_ptr<Connection> sp = cp -> getConnection();
            sp->update(sql);
        }
    });

    t1.join();
    t2.join();
    t3.join();
    t4.join();

    clock_t end = clock();
    cout<< (end - begin) << "ms" << endl;

    return 0;
}
```

条件变量示例：

```
#include <iostream>
#include <thread>
#include <queue>
#include <mutex>
#include <condition_variable>

std::queue<int> q;                  // 共享队列
std::mutex mtx;                     // 互斥锁用于保护共享资源
std::condition_variable cv;         // 条件变量用于线程间的同步
bool done = false;                  // 生产者是否完成的标志

// 生产者函数
void producer(int count) {
    for (int i = 1; i <= count; ++i) {
        std::unique_lock<std::mutex> lock(mtx);
        q.push(i);                  // 将数据放入队列
        std::cout << "Produced: " << i << std::endl;
        cv.notify_one();            // 通知等待的消费者
    }
    // 设置完成标志并通知所有消费者
    {
        std::unique_lock<std::mutex> lock(mtx);
```

```
        done = true;
    }
    cv.notify_all();
}

// 消费者函数
void consumer() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, []{ return !q.empty() || done; }); // 等待条件变量

        if (!q.empty()) {
            int value = q.front();
            q.pop();
            std::cout << "Consumed: " << value << std::endl;
        } else if (done) {
            break;
        }
    }
}

int main() {
    std::thread prod(producer, 10); // 生产者线程
    std::thread cons1(consumer);    // 消费者线程 1
    std::thread cons2(consumer);    // 消费者线程 2

    prod.join();  // 等待生产者完成
    cons1.join(); // 等待消费者1完成
    cons2.join(); // 等待消费者2完成

    return 0;
}
```