public class MatchingPool extends Thread {  
 private static List<Player> *players* = new ArrayList<>();  
 private final ReentrantLock lock = new ReentrantLock();  
 private static RestTemplate *restTemplate*;  
 private final static String *startGameUrl* = "http://127.0.0.1:3000/pk/start/game/";  
  
 @Autowired  
 public void setRestTemplate(RestTemplate restTemplate) {  
 MatchingPool.*restTemplate* = restTemplate;  
 }  
  
 public void addPlayer(Integer userId, Integer rating, Integer botId) {  
 lock.lock();  
 try {  
 *players*.add(new Player(userId, rating, botId, 0));  
 } finally {  
 lock.unlock();  
 }  
 }  
  
 public void removePlayer(Integer userId) {  
 lock.lock();  
 try {  
 List<Player> newPlayers = new ArrayList<>();  
 for (Player player: *players*) {  
 if (!player.getUserId().equals(userId)) {  
 newPlayers.add(player);  
 }  
 }  
 *players* = newPlayers;  
 } finally {  
 lock.unlock();  
 }  
 }  
  
 private void increaseWaitingTime() { *// 将所有当前玩家的等待时间加1* for (Player player: *players*) {  
 player.setWaitingTime(player.getWaitingTime() + 1);  
 }  
 }  
  
 private boolean checkMatched(Player a, Player b) { *// 判断两名玩家是否匹配* int ratingDelta = Math.*abs*(a.getRating() - b.getRating());  
 int waitingTime = Math.*min*(a.getWaitingTime(), b.getWaitingTime());  
 return ratingDelta <= waitingTime \* 10;  
 }  
  
 private void sendResult(Player a, Player b) { *// 返回匹配结果* System.*out*.println("send result: " + a + " " + b);  
 MultiValueMap<String, String> data = new LinkedMultiValueMap<>();  
 data.add("a\_id", a.getUserId().toString());  
 data.add("a\_bot\_id", a.getBotId().toString());  
 data.add("b\_id", b.getUserId().toString());  
 data.add("b\_bot\_id", b.getBotId().toString());  
 *restTemplate*.postForObject(*startGameUrl*, data, String.class);  
 }  
  
 private void matchPlayers() { *// 尝试匹配所有玩家* boolean[] used = new boolean[*players*.size()];  
 for (int i = 0; i < *players*.size(); i ++ ) {  
 if (used[i]) continue;  
 for (int j = i + 1; j < *players*.size(); j ++ ) {  
 if (used[j]) continue;  
 Player a = *players*.get(i), b = *players*.get(j);  
 if (checkMatched(a, b)) {  
 used[i] = used[j] = true;  
 sendResult(a, b);  
 break;  
 }  
 }  
 }  
  
 List<Player> newPlayers = new ArrayList<>();  
 for (int i = 0; i < *players*.size(); i ++ ) {  
 if (!used[i]) {  
 newPlayers.add(*players*.get(i));  
 }  
 }  
 *players* = newPlayers;  
 }  
  
 @Override  
 public void run() {  
 while (true) {  
 try {  
 Thread.*sleep*(1000);  
 lock.lock();  
 try {  
 increaseWaitingTime();  
 matchPlayers();  
 } finally {  
 lock.unlock();  
 }  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 break;  
 }  
 }  
 }  
}

struct Task

{

User user;

string type;

};

struct MessageQueue

{

queue<Task> q;

mutex m;

condition\_variable cv;

}message\_queue;

class Pool

{

public:

void save\_result(int a, int b)

{

printf("Match Result: %d %d\n", a, b);

std::shared\_ptr<TTransport> socket(new TSocket("123.57.47.211", 9090));

std::shared\_ptr<TTransport> transport(new TBufferedTransport(socket));

std::shared\_ptr<TProtocol> protocol(new TBinaryProtocol(transport));

SaveClient client(protocol);

try {

transport->open();

int res = client.save\_data("acs\_0", "6e822f5b", a, b);

if (!res) puts("success");

else puts("failed");

transport->close();

} catch (TException& tx) {

cout << "ERROR: " << tx.what() << endl;

}

}

bool check\_match(uint32\_t i, uint32\_t j)

{

auto a = users[i], b = users[j];

int dt = abs(a.score - b.score);

int a\_max\_dif = wt[i] \* 50;

int b\_max\_dif = wt[j] \* 50;

return dt <= a\_max\_dif && dt <= b\_max\_dif;

}

void match()

{

for (uint32\_t i = 0; i < wt.size(); i ++ )

wt[i] ++ ; // 等待秒数 + 1

while (users.size() > 1)

{

bool flag = true;

for (uint32\_t i = 0; i < users.size(); i ++ )

{

for (uint32\_t j = i + 1; j < users.size(); j ++ )

{

if (check\_match(i, j))

{

auto a = users[i], b = users[j];

users.erase(users.begin() + j);

users.erase(users.begin() + i);

wt.erase(wt.begin() + j);

wt.erase(wt.begin() + i);

save\_result(a.id, b.id);

flag = false;

break;

}

}

if (!flag) break;

}

if (flag) break;

}

}

void add(User user)

{

users.push\_back(user);

wt.push\_back(0);

}

void remove(User user)

{

for (uint32\_t i = 0; i < users.size(); i ++ )

if (users[i].id == user.id)

{

users.erase(users.begin() + i);

wt.erase(wt.begin() + i);

break;

}

}

private:

vector<User> users;

vector<int> wt; // 等待时间, 单位：s

}pool;

class MatchHandler : virtual public MatchIf {

public:

MatchHandler() {

// Your initialization goes here

}

int32\_t add\_user(const User& user, const std::string& info) {

// Your implementation goes here

printf("add\_user\n");

unique\_lock<mutex> lck(message\_queue.m);

message\_queue.q.push({user, "add"});

message\_queue.cv.notify\_all();

return 0;

}

int32\_t remove\_user(const User& user, const std::string& info) {

// Your implementation goes here

printf("remove\_user\n");

unique\_lock<mutex> lck(message\_queue.m);

message\_queue.q.push({user, "remove"});

message\_queue.cv.notify\_all();

return 0;

}

};

class MatchCloneFactory : virtual public MatchIfFactory {

public:

~MatchCloneFactory() override = default;

MatchIf\* getHandler(const ::apache::thrift::TConnectionInfo& connInfo) override

{

std::shared\_ptr<TSocket> sock = std::dynamic\_pointer\_cast<TSocket>(connInfo.transport);

/\*cout << "Incoming connection\n";

cout << "\tSocketInfo: " << sock->getSocketInfo() << "\n";

cout << "\tPeerHost: " << sock->getPeerHost() << "\n";

cout << "\tPeerAddress: " << sock->getPeerAddress() << "\n";

cout << "\tPeerPort: " << sock->getPeerPort() << "\n";\*/

return new MatchHandler;

}

void releaseHandler(MatchIf\* handler) override {

delete handler;

}

};

void consume\_task()

{

while (true)

{

unique\_lock<mutex> lck(message\_queue.m);

if (message\_queue.q.empty())

{

// message\_queue.cv.wait(lck);

lck.unlock();

pool.match();

sleep(1);

}

else

{

auto task = message\_queue.q.front();

message\_queue.q.pop();

lck.unlock();

if (task.type == "add") pool.add(task.user);

else if (task.type == "remove") pool.remove(task.user);

}

}

}

int main(int argc, char \*\*argv) {

TThreadedServer server(

std::make\_shared<MatchProcessorFactory>(std::make\_shared<MatchCloneFactory>()),

std::make\_shared<TServerSocket>(9090), //port

std::make\_shared<TBufferedTransportFactory>(),

std::make\_shared<TBinaryProtocolFactory>());

cout << "Start Match Server" << endl;

thread matching\_thread(consume\_task);

server.serve();

return 0;

}

条件变量是一种特殊的变量，它可以让一个线程在某个条件成立之前等待

* wait：这个方法会让当前线程等待，直到条件变量被通知。
* notify\_one：这个方法会唤醒一个等待的线程。
* notify\_all：这个方法会唤醒所有等待的线程。