阶段一 宠物小精灵的加入

宠物小精灵宏观设计

小精灵基类

- 种族
- 名字
- 等级
- 经验值
- 攻击力
- 防御力
- 生命值
- 速度（攻击间隔）

各个小精灵种族的类

- 种族名字
- 种族类型（高攻击、高防御、高血量、高速度）
- 种族基础攻击力
- 种族基础防御力
- 种族基础生命值
- 种族基础速度
- 种族等级
- 种族经验曲线
- 种族技能

小精灵根据自己种族的经验曲线来决定每升一级如何增加攻击力、防御力、生命值、速度，不同种族经验曲线不同（以函数形式呈现？）

每个值的峰值都是100？HP峰值1000

精灵对战

- 对战过程中，小精灵以攻击时间间隔决定攻击顺序，速度峰值为10，即10回合里能攻击10次；

种族技能

- 每个种族设置4个技能，其中1个普通攻击，3个特殊技能（可以是被动？）
- 普通攻击不限使用次数，特殊技能限制使用次数
- 升级过程中逐渐获得技能，一开始拥有普通攻击和一个特殊技能，6级拥有第三个技能，10级拥有第四个技能
- 技能存在一定的被闪避几率
- 技能存在一定的暴击几率

小精灵类

- 继承自小精灵种族的类
- 增加战斗时属性（战斗时各种属性值会上下浮动，在基本属性上面修改可能会出现问题）
- 小精灵id

精灵类的实现

PokemonBase基类

基类用于表示所有的小精灵。其中根据小精灵的基础种类

```
enum PokemonRace{Pow,Shi,Def,Spe};
```

来决定该小精灵的基础ATK、DFS、HP、SPEED以及经验曲线等。这些属性的访问权限设置为protected以方便派生类访问和基类指针访问。纯虚函数 `attack()` 和 `dodge()` 使精灵可以发动攻击和闪避。

其代码如下

```
class PokemonBase
{
public:
    PokemonBase(PokemonRace race);
    // 虚函数
    virtual void attack(Pokemon &attacker, Pokemon &opponent, int skillIndex,
string &sendbuf)const{}
    virtual bool dodge(int skill_accuraccy,int attack_speed, int opponent_speed,
string &sendbuf) const;

    // getter
    string racename() const { return _racename; }
    PokemonRace race() const { return _race; }
    int baseAtk() const { return _baseAtk; }
    int baseDfs() const { return _baseDfs; }
    int baseHp() const { return _baseHp; }
    int baseSpeed() const { return _baseSpeed; }
    string skillname(int n) const;
    string skilldesc(int n) const;
    int pp(int n) const{ return _pp[n]; }
    growth_curve baseCurve() const { return _baseCurve; }
    growth_curve raceCurve() const { return _raceCurve; }

    void changeRacename(const string & racename)
    {
        _racename = racename;
    }
protected:
    string _racename;
    PokemonRace _race;

    int _baseHp;
    int _baseAtk;
    int _baseDfs;
    int _baseSpeed;
```

```
    string _skillNames[4];
    string _skillDesc[4];

    //baseCurve是每个小精灵的基本成长曲线
    //raceCurve根据种族不同，采用随机数的方式
    growth_curve _baseCurve,_raceCurve;

    // 技能使用次数，第一项默认为INT_MAX；
    int _pp[4];

};
```

Race模板类

Race表示的是四种类型的小精灵再细分的种族，如力量型又可以再细分为：沙瓦朗、火爆猴、腕力。Race类以PokemonBase作为基类，再对PokemonBase类中为初始化的一些属性进行设定，比如成长曲线、技能名、描述等等，还为种族的每个技能提供了计算公式。

这里使用到了无类型模板参数的知识点

template<>中并不一定要是类模板，也可以是int、enum等作为模板的参数，在函数内是可以使用的。但是在Race类中，无类型模板参数int N只起到了区分不同种族的作用，在构造函数和攻击函数中并没有用到N。具体用法：

```
template<int N>
class Race :public PokemonBase
{
public:
    Race();
    void attack(Pokemon &attacker, Pokemon &opponent, int skillIndex, string
&sendbuf) const;
};

// 火爆猴--力量型
template<>
Race<0>::Race() :PokemonBase(Pow)
{
    changeRacename("Primeape");
    _skillNames[0] = "SCRATCH";
    _skillDesc[0] = "Normal attack with low damage";
    _skillNames[1] = "KARATE-CHOP";
    _skillDesc[1] = "Normal attack with high damage";
    _skillNames[2] = "SHOULDER-THROW";
    _skillDesc[2] = "Cause damage and decrease enemy's dfs";
    _skillNames[3] = "SEISIMIC-TOSS";
    _skillDesc[3] = "Cause twice damage of its own atk";
    _pp[0] = INT_MAX;
    _pp[1] = 10;
    _pp[2] = 5;
    _pp[3] = 3;

    //种族成长曲线
    _raceCurve.up_atk = 15;
    _raceCurve.up_dfs = 5;
```

```
    _raceCurve.up_hp = 20;
    _raceCurve.up_speed = 5;

}
```

Pokemon类

Pokemon类用于创建具体的小精灵的对象。因此它具有一个小精灵的所有属性，分为普通属性（静态）和战时属性（动态）。

所有getter函数都是const成员函数，避免小精灵属性被修改。

Pokemon类有两种构造函数，一种是创建新的小精灵时，只需要知道种族即可，第二种是根据给定属性值构造一个现有的小精灵，通常用于战斗中临时生成一个小精灵副本。

战时属性会随着战斗过程动态变化，因此提供setter和getter函数用于修改属性值，而非战时属性不可随意更改，只能通过升级函数进行修改，因此只提供getter函数。

在Pokemon类中还有一组静态的PokemonBase的常引用，用于发动对应种族的招式以及使用对应种族的成长曲线。而Pokemon的构造函数需要提供种族对应的Index，从而获取该种族的一些基础属性，如非战时属性以及招式等等。因为Race模板类继承于PokemonBase基类，因此PokemonBase的指针可以调用Race中实现的attack()函数。

```
// 每个小精灵动态的属性
class Pokemon
{
public:
    Pokemon(int raceIndex,const string &name = "");
    Pokemon(int pokemonid,int raceIndex, const string &name, int atk, int def,
int HP, int speed, int lv, int ep);
    // 普通状态下的getter和setter
    // const成员函数
    string name()const { return _name; }
    int race()const { return _raceIndex;}
    int atk() const { return _atk; }
    int dfs() const { return _dfs; }
    int hp() const { return _hp; }
    int speed() const { return _speed; }
    int ep()const { return _ep; }
    int lv()const { return _lv; }
    int pokemonid()const { return _pokemonid; }
    void setAtk(int atk);
    void setDfs(int dfs);
    void setHp(int hp);
    void setSpeed(int speed);
    void addEp(int ep);
    // 和种族有关的
    string racename()const { return races[_raceIndex]->racename(); }
    string skillname(int skillIndex) const { return races[_raceIndex]-
>skillname(skillIndex); }
    string skilldesc(int skillIndex) const { return races[_raceIndex]-
>skilldesc(skillIndex); }
    // 战斗状态下的getter
    int battleAtk() const { return _battleAtk; }
    int battleDfs() const { return _battleDfs; }
```

```
    int battleHp() const { return _battleHp; }
    int battleSpeed() const { return _battleSpeed; }
    int battlePp(int i)const { return _battlepp[i]; }
    // 战斗开始前恢复所有属性
    void restoreAll();
    // 攻击,返回伤害值
    void attack(Pokemon& opponent, string &sendbuf,int autofight);
    // 遭受伤害
    void takeDamage(int damage);
    // 吸收经验
    void gainEp(int ep);
private:
    static const PokemonBase *races[RACE_NUM];
    int _raceIndex;
    string _name;
    int _pokemonid;
    int _ep;
    int _lv;
    // 分为正常的属性和战斗状态属性
    // 正常属性
    int _atk;
    int _dfs;
    int _hp;
    int _speed;
    // 战斗状态属性
    int _battleAtk;
    int _battleDfs;
    int _battleHp;
    int _battleSpeed;
    int _battlepp[4];
};
```

对战系统的实现

BattleController类

为对战封装了BattleController类，小精灵的对战在该类内部完成，战斗结果通过startBattleForClient()的返回值得知。

```
class BattleController
{
public:
    BattleController(Pokemon &p1, Pokemon &p2,SOCKET &clientSocket,bool
autofight);
    //void startBattle();
    int startBattleForClient();
    strFunction strfunc;
private:
    Pokemon &_p1;
    Pokemon &_p2;
    int _autofight;
    int _timer1,_timer2;
    SOCKET _clientSocket;
```

```
    string sendbuf;
};
```

出招顺序

在battlecontroller中，对战顺序由内部的两个私有变量_timer1和 _timer2实现。 _timer1设置为 _p2的速度值， _timer2设置为 _p1的速度值。当 _timer1自减为0时，相应的 _p1进行攻击。这样设定逻辑简单，且符合攻击机会与速度成正比，在实际的小精灵对战测试中具有不错的公平性。

```
//双方轮流出招,一方HP为0时停止
while (_p1.battleHp() && _p2.battleHp())
{
    _timer1--, _timer2--;
    if (!_timer1)
    {
        _p1.attack(_p2, _autofight);
        _timer1 = _p2.battleSpeed();
    }
    if (!(_p1.battleHp() && _p2.battleHp()))
        break;
    if (!_timer2)
    {
        _p2.attack(_p1, _autofight);
        _timer2 = _p1.battleSpeed();
    }
    Sleep(BATTLETIMEGAP);
}
```

attack函数

attack函数在Race模板类中定义,根据不同的种族分别定义攻击技能，以某一个种类为例

```
// 火爆猴技能
template<>
void Race<0>::attack(Pokemon &attacker, Pokemon &opponent, int skillIndex
,string& sendbuf) const
{
    cout << attacker.name() << "使用了:" <<  attacker.skillname(skillIndex)
<<"!\n" << endl;
    int atk, dfs, damage,skillPower,lv;
    switch (skillIndex)
    {
    case 0:
        //技能0细节
        skillPower = rand() % attacker.battleAtk()*0.2+
attacker.battleAtk()*0.6;
        atk = attacker.battleAtk();
        dfs = opponent.battleDfs();
        lv = attacker.lv();
        damage = damageFunction(atk, dfs,lv,skillPower);
        opponent.takeDamage(damage);
        break;
    case 1:
        //技能1细节
        break;
```

```
    case 2:
        //技能2细节
        break;
    case 3:
        //技能3细节
        break;
    default:
        break;
    }
}

void Pokemon::attack(Pokemon& opponent,string &sendbuf, int autofight)
{
    // 选择招式
    cout << "轮到" << _name << "出招了！" << endl;
    // 手动选择
    int chosen_skill=0;
    if (!autofight)
    {
        // manual fight
    }
    // 自动选择
    else
    {
        // 找出可用的招式
        bool usable_skill[4] = { 1 };
        for (int i = 1; i < 4; i++)
        {
            if (_battlepp[i] > 0)
            {
                usable_skill[i] = true;
            }
        }
        // 从可用的招式随机选择一个
        do
        {
            chosen_skill = rand() % 4;
        } while (!usable_skill[chosen_skill]);
        _battlepp[chosen_skill]--;
    }
    races[_raceIndex]->attack(*this, opponent, chosen_skill,sendbuf);
}
```

在小精灵进行攻击的时候，使用基类指针races[_raceIndex]来调用虚函数attack

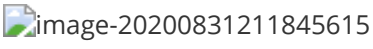
伤害计算以及闪避计算

伤害计算和闪避计算均借鉴了宝可梦的计算方式

其中伤害计算：

伤害值 = [(攻击方的LV×0.4 + 2)×技巧威力×攻击方的攻击（或特攻）能力值÷防御方的防御（或特防）能力值÷50 + 2]×各类修正×(217 ~ 255之间)÷255

命中判定：



简化后的本系统的伤害函数和闪避函数，经过测试后实际效果良好。

```
int damageFunction(int atk, int dfs,int attacker_lv,int skillpower)
{
    // 宠物小精灵伤害公式（攻击方等级 × 2 ÷ 5 + 2） × 技能威力 × 攻击方攻击力 ÷ 防御方防御力 ÷ 50 + 2
    //
    int damage = (attacker_lv + 2)*skillpower*atk / dfs / 10 + 2;
    return damage;
}
// 闪避判定
bool PokemonBase::dodge(int skill_accuraccy ,int attack_speed, int opponent_speed, string &sendbuf)const
{
    // A = B * C / D
    //B由招式的命中决定、C由攻击方速度决定、D由防御方速度决定
    //B根据招式决定，招式命中率越高数值越大，最大为255,传入参数为百分比
    //产生一个1~255之间的随机数，该随机数小于A时则为命中，否则为闪避
    int rand_int = rand() % 255 + 1;
    int B;
    switch (skill_accuraccy)
    {
    case 100:
        B = 255;
        break;
    case 95:
        B = 242;
        break;
    case 90:
        B = 229;
        break;
    case 85:
        B = 216;
        break;
    case 80:
        B = 204;
        break;
    case 75:
        B = 191;
        break;
    case 70:
        B = 78;
        break;
    default:
        break;
    }
    int A = B * attack_speed / opponent_speed;

    if (rand_int >= A)
    {
        cout << "这招居然被闪避了！" << endl;
        sendbuf += "1 0\n";
        return true;
    }
    else
    {
        sendbuf += "0 0\n";
```

```
        return false;
    }
}
```

阶段二 用户的联网

C/S结构宏观设计

要求

- 每个用户需要注册一个账号，用户名全局唯一，不能有任何两个用户名相同，要考虑注册失败的场景时的反馈
- 实现注册、登录、登出功能，均采用C/S模式，客户端和服务端用socket进行通信，服务端保存所有用户的信息
- 每个用户拥有：用户名、拥有的精灵，两个属性。 用户注册成功时，系统自动随机分发三个1级精灵给用户
- 用户可以查看所有成功注册用户拥有的精灵，也可以查看所有当前在线的用户
- 题目主要考察点：socket通信，交互场景反馈，用户信息存储方式，界面交互，其它合理的新颖设计。

Client端

采用QT设计ui，为了方便界面的排版布局，ide采用了QT Creator。使用QTcpSocket作为通信基础。

- 注册（用户名、密码、昵称），用户名唯一，考虑注册失败的反馈
- 登录、登出
- 查看用户的信息
- 修改密码
- 查看小精灵信息
- 展示S端反馈的信息（包括错误信息）

Server端

ide：vs2017，采用Winsock2.h, thread等进行socket编程

- 与数据库交互，存取用户信息，小精灵信息
- 持续运行，不间断地处理C端发来的指令
- 将处理后的结果反馈至C端

数据库

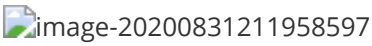
因为只需要本地数据库，因此使用了SQLite3作为数据库进行数据的增删查改。

数据库schema

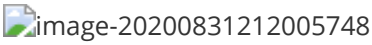
```
Login(
    id integer primary key,
    username text unique not null,
    password text not null
)
```

```
Pokemon(
    id integer primary key,
    userid integer not null,
    name text not null,
    race integer not null,
    atk integer not null,
    dfs integer not null,
    hp integer not null,
    speed integer not null,
    lv integer not null,
    ep integer not null,
)
```

Login



Pokemon



详细设计

Server端

采用了长连接短连接的思想，将Server分为Hub和Endpoint。

短连接：【Hub】建立连接→数据传输→关闭连接

长连接：【Endpoint】建立连接→数据传输→保持连接→...→某一方断开连接。

Hub

Hub主要用于处理登录、注册的信息，并且为登录成功的用户分配Endpoint。线程函数ListenFunc将持续监听来自客户端的信息，线程函数TerminateFunc作为终止函数，当输入任意字符后服务器结束运行。

Hub管理着一个Endpoint的列表，使用vector数据结构实现频繁的插入和删除操作，并使用了互斥量来进行共享数据的保护。

为了处理多用户登录的情况，Hub将为每一个Endpoint开启一个独立的线程，并且为Endpoint分配合适的端口号。当用户登录成功，Hub为该用户建立一个新的Endpoint，并启动该Endpoint分配端口号。新的Endpoint被放入线程池（vector）中，并由Hub中的线程函数EndpointHandler进行监控。当Endpoint对应的用户登出，线程函数EndpointHandler会将该Endpoint从线程池中移除并释放。

服务器的登录、注册端口固定为8888，只能存在一个Hub。

```
class Hub
{
private:
    const int HUB_PORT = 8888;
    SOCKET hubSocket;
    SOCKET clientSocket;
    vector<Endpoint *> endpoints;
    char buf[DEFAULT_BUFLEN];
    sqlite3 *db;
```

```
    mutex hub_mute;
    // 监听线程
    bool running;
    void ListenFunc();
    void TerminateFunc();
public:
    void dealClientInfo(const string &recv);
    void run();
    void endpointHandler(Endpoint *const endpoint);
    Hub() {};
    strFunction strfunc;
};
```

调用Hub::run()，服务器启动，进行socket的初始化，sqlite的初始化以及产生监听线程来监听客户端的信息，并通过Hub::dealClientInfo函数进行请求的处理。

Endpoint

Endpoint类负责实现用户登录之后的所有请求，包括：修改密码、查看用户列表、查看小精灵列表、更改小精灵姓名、登出等。

当客户端登录成功，Online变量被设置为True，通过Endpoint::start()函数初始化socket，从数据库获取用户信息，并返回端口号。客户端再根据这个端口号与Endpoint进行通信。启动后，调用Endpoint::process()来控制监听线程。Endpoint的监听函数与Hub大致相同，在此不再赘述。

当用户登出，running变量和online变量被设置为false，监听线程退出，该Endpoint被释放。

```
class Endpoint
{
private:
    Hub &hub;
    sqlite3 *&db;

    int _PlayerID;
    string _userName;

    string _ip;
    int _port;
    SOCKET endpointSocket;
    SOCKET clientSocket;

    string buf2;

    bool _online;
    bool running;

    mutex endpoint_mute;
public:
    Endpoint(int userId, sqlite3 *&db,Hub &hub);
    ~Endpoint();
    // getter
    int getPlayerID()const { return _PlayerID; }
    bool getOnline()const { return _online; }
    int getPort()const { return _port; }
    int getWinRound()const { return win_round; }
    int getTotalRound()const { return total_round; }
    // 客户端指令处理
```

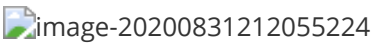
```
    void dealClientInfo(const char* recvbuf);
    void changePokemonName(int pokemonID,string newname);
    void getPokemonList(int PlayerID);
    void changePassword(int PlayerID, const string oldpassword, const string
newpassword);
    void getUserList();
    void logOut();
    void savePokemonToDB(const Pokemon &pokemon);
    void getPokemonInfo(int PokemonID);
    // 控制线程
    void process();
    // 启动endpoint，分配端口号
    int start();
    // 结束endpoint
    void terminateFunc();
    // endpoint处理线程
    void listenFunc();
    strFunction strfunc;
};
```

Client端

Client端所有ui的设计均在QT Creator内完成，基本的信号与槽不在此展示，仅展示一些窗口的设计和运行逻辑

PokemonWidget类

本窗口为客户端的主窗口，除了登录、注册，其它一切操作均在此窗口完成。



窗口初始化时，socket不与服务端连接。当执行完登录操作后，server端向client端发送分配的Endpoint的端口号，端口号通过登录窗口的信号发送给主窗口，主窗口得到端口号后与Endpoint进行连接，连接成功后进入功能界面。信号中还传递了用户的id。

```
void PokemonWidget::login_slot(int endpointport,int playerid)
{
    port = endpointport;
    clientSocket->connectToHost("127.0.0.1",endpointport);
    playerID=playerid;
    // 成功登录，成功连接endpoint
    // 跳转到主界面
    ui->stackedWidget->setCurrentIndex(1);
    changeState(MAIN);
}
```

登录成功后跳转到主界面。界面的跳转通过changeState()函数完成，设置状态枚举变量State来表示当前页面的状态。而采用Qt中的stacked widget模块来同时存储多个页面。当改变状态时，跳转到相应的页面即可。

```
enum State{
 // 显示所有功能案件
    MAIN,
 // 显示登录、注册等按钮
    LOGIN,
```

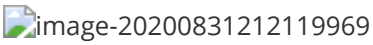
```
    // 显示小精灵列表
    POKEMONLIST,
    // 显示用户列表，并设有查看用户小精灵的按钮
    USERLIST,
    // 登出状态
    LOGOUT,
    // 显示修改密码界面
    CNGPSW,
    // 显示小精灵信息
    POKEMONINFO,
    // 修改小精灵名字
    CNGPONAME
};

// 细节省略
void PokemonWidget::changeState(State state)
{
    this->state=state;
    switch (state) {
    case MAIN:
    case LOGIN:
    case USERLIST:
    case POKEMONLIST:
    case CNGPSW:
    case POKEMONINFO:
    case FIGHT:
    case LOSEPOKEMON:
    default:
        break;
    }
}
```

实现功能的逻辑：通过主窗口的socket将通信请求发给Endpoint，Endpoint进行处理后发回给Client。在主窗口中建立了一个槽，以socket收到数据作为信号，执行PokemonWidget::dealClientInfo()函数，即根据服务器返回的报文展示或处理。

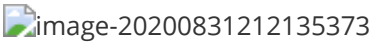
```
// 收到服务器发送的数据的信号
connect(clientSocket,&QTcpSocket::readyRead,this,&PokemonWidget::dealServerMsg);

// 省略细节
void PokemonWidget::dealServerMsg()
{
    QString msg = QString::fromLocal8Bit(clientSocket->read(DEFAULT_BUFLEN));
    string str = msg.toStdString();
    switch (state) {
    case CNGPONAME:
    case USERLIST:
    case POKEMONLIST:
    case POKEMONINFO:
    case ...
    default:
        break;
    }
}
```

一下为某些功能界面截图：

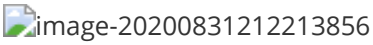
菜单界面



用户列表

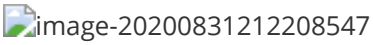


小精灵列表



小精灵详细信息和修改名字

注：修改名字的逻辑是将名字栏设为可修改，用户点击修改名字后就会读取名字栏的内容并发送给服务器。即，即使不修改内容也可以执行修改名字的操作。



LoginWidget类 & RegisterWidget类

LoginWidget与RegisterWidget的实现机制相同，在此仅介绍LoginWidget。

LoginWidget类的数据结构：

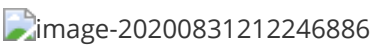
```
class loginWidget : public QWidget
{
    Q_OBJECT
public:
    Ui::loginWidget *ui;
    void dealLoginMsg();
    explicit loginWidget(QWidget *parent = nullptr);
    ~loginWidget();
    strFunction strfunc;
private slots:
    void on_buttonBack_clicked();
    void on_buttonLogin_clicked();
signals:
    void login_signal(int,int);
private:
    int port;
    QTcpSocket *clientSocket;
};
```

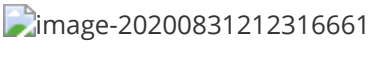
登录窗口与主窗口并不共用一个Socket。当用户输入用户名和密码并成功提交后（按下登录按钮），clientSocket会通过端口8888与Hub相连，发送登录请求，服务器成功处理之后返回一个报文，窗口通过dealLoginMsg进行解析并获得Endpoint的端口号和用户id。最后通过信号将这两个值传递给主窗口。

strFunction类包含了一些基础的字符串检查函数，包括检查用户名、密码是否格式正确。其中用户名和密码只能由英文字母、数字、下划线组成，长度不得大于30。当登录失败会通过QMessageBox进行异常反馈。

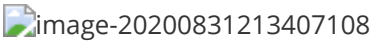
当关闭登录成功后，必须得释放clientSocket，不然进行注册或再次登录或出现socket连接失败的异常。

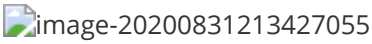
登录界面

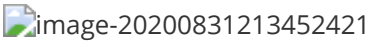




注册界面







通信格式

客户端与服务器的通信格式：

- Hub提供的接口
  - <login> <username> <password>
    - 返回端口号和玩家id为成功，其它情况返回错误信息
  - <logon> <username> <password>
    - 返回“Accept!\n”为成功，其他情况返回错误信息
- Endpoint提供的接口
  - <changePokemonName> <PlayerID> <pokemonid> <newname>
    - 返回“Accept!\n”为成功，其他情况返回错误信息
  - <getPokemonList> <userid>
    - 返回该玩家所有宝可梦的信息
    - 每个宝可梦之间由换行符隔开
    - 每个宝可梦的属性由空格隔开
  - <getUserList>
    - 返回所有玩家的信息
    - 每个玩家之间由换行符隔开
    - 每个玩家的信息由空格隔开
  - <logout>
    - 返回“Accept!\n”为成功，其他情况返回错误信息
  - <changePassword> <id> <oldpassword> <newpassword>
    - 返回“Accept!\n”为成功，其他情况返回错误信息
  - <getPokemonInfo> <pokemonid>
    - 返回该宝可梦的属性
    - 属性之间由空格隔开
- 如果都不符合，则服务器返回字符串“recv INVALID request!”

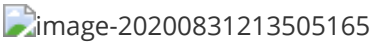
主程序

```
int main()
{
    srand(time(NULL));
    Hub hub;
    hub.run();
    return 0;
}
```

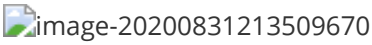
运行结果

客户端运行结果已展示，在此仅展示服务器运行结果

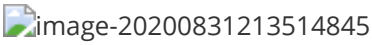
启动服务器



登录成功



其它指令



阶段三 联网对战

要求

- 已经登录的在线用户可以和服务器进行虚拟决斗，决斗分两种：升级赛和决斗赛，两种比赛都能增长宠物经验值。服务器上有一个虚拟精灵的列表，用户可以挑选其中任意一个进行比赛（升级赛或者决斗赛）。另外决斗赛中用户胜出可以直接获得该战胜的的精灵，失败则系统从用户的精灵中随机选三个（不够三个精灵的情况就选择他所有的精灵），然后由用户选一个送出
    - 升级赛 只是用户用来增加精灵经验值，规则开发者自定
    - 累积多少经验值升一级，规则开发者自定
    - 决斗赛的上述规则同升级赛，只是额外还可以赢得宠物一个
- 用户如果没有精灵（比如总是失败，已经全部送出去），则系统会随机放给给他一个初级精灵
- 请让你的系统自动模拟每场比赛的每次出招。另外，为了增加不确定性，可以加入概率闪避攻击和暴击伤害机制
    - 比赛的过程和结果由系统根据上述规则自动模拟完成，要求结果具有一定的随机性
- 用户增加新功能，可以查看某个用户的胜率
- 用户增加新属性，为宠物个数徽章（金银铜）和高级宠物徽章（金银铜），分别根据拥有的宠物个数的多少和拥有高级宠物（15级）个数的多少颁发
- 如有界面设计可酌情加分，如有新颖设计可酌情加分
- 题目考察点：客户端与服务器数据交互（可采用多进程或异步通信或其他方法均可），并发请求处理，类的方法设计，伤害计算方法设计

在线对战系统设计

客户端部分

客户端在原有基础上，STATE增加`FIGHT,LOSEPOKEMON`两个状态，分别表示小精灵战斗状态，决斗模式失败丢弃小精灵的状态。

增加新的结构体FightController和枚举 FightState:

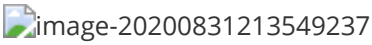
```
enum FightState{NOFIGHT=100,LEVELUPMODE,DUELMODE};
struct FightController
{
    int myPokemonid;
    int opponentPokemonid;
    FightState mode;        //三种战斗情况
    bool initFight;         //初始化战斗界面的条件
};
```

FightController用于判断是否初始化战斗界面（第一次战斗前以及每次战斗结束后初始化）、战斗结束后的结算方式（升级模式只返回QMessageBox,对决模式失败要跳转到给出小精灵的界面），以及存储对战双方的id。

客户端战斗流程设计如下：

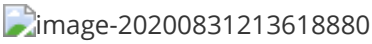
- 选择小精灵的界面继承自查看精灵列表的界面，跳转到精灵列表后根据`FightController`的`mode`判断是否可以选择小精灵出战。如果是`NOFIGHT`则出战按钮不可用，其它为可用。

选择自己的精灵出战，会显示"就决定是你了"

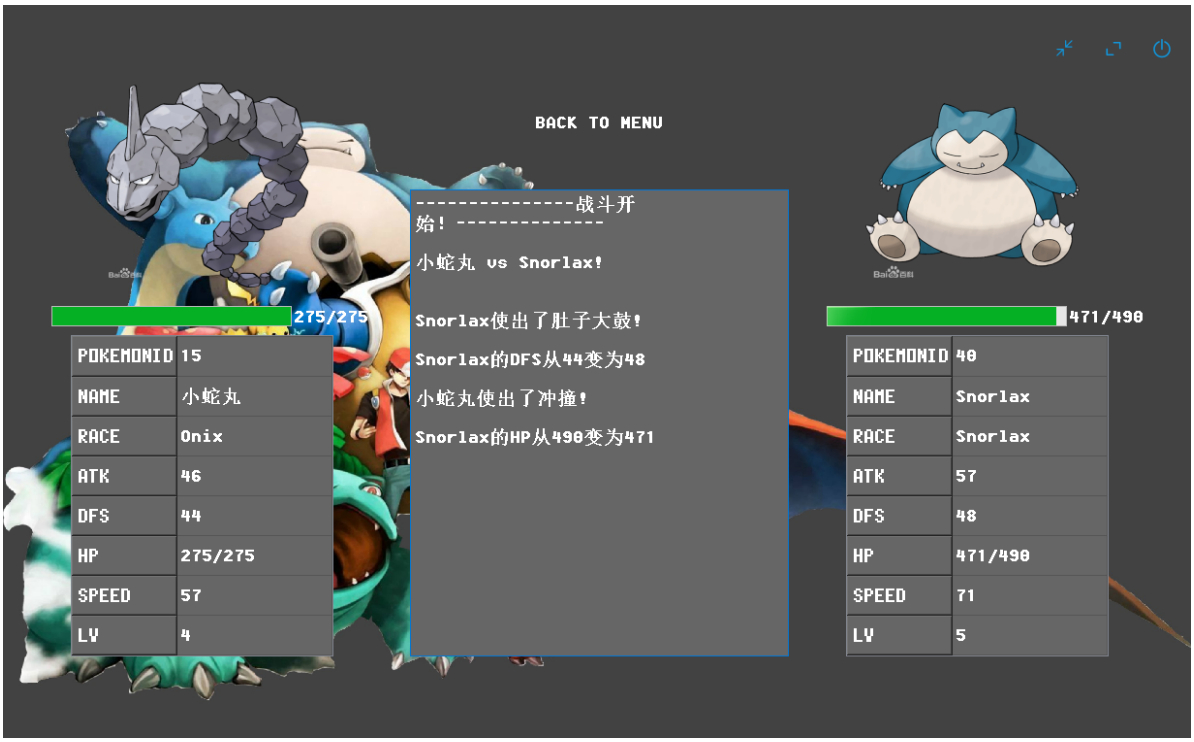


选择成功会有弹窗提示

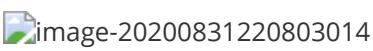
 **选择对手小精灵出战，会显示"和我决斗吧"**

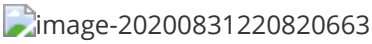


- 升级模式的对手可以选择任意小精灵（玩家小精灵+系统小精灵），决斗模式的对手只能是系统小精灵。决斗失败后，自己的某一只小精灵会变为系统小精灵
- 对战界面：左右各为小精灵信息的展板，中间的TextBrowser是战斗信息的输出模块。



- 若是决斗模式失败，则状态变为`LOSEPOKEMON`，若小精灵只剩一只，会在给出小精灵后随机为玩家分配一名1级小精灵。





服务器部分

该版本服务器主要添加对战功能，Endpoint添加下列接口：

- <battle> <pokemonid1> <pokemonid2>
  - pokemonid1为发起攻击的小精灵
  - 升级战与决斗战在服务器端没有区别，在客户端区分区别
  - 返回值为对战细节信息，将在通信协议设计内讲解
- <changePokemonMaster> <pokemonid> <masterid>
  - 将小精灵pokemonid的主人修改为玩家masterid
  - 成功返回”Accept.\n“，其他情况返回错误信息
- <losePokemon> <playerID>
  - 返回被选择的几个小精灵的信息
  - 若小精灵只有一个，则为用户添加一个1级精灵

BattleController、attack函数都进行了修改，都在参数中添加了一个string，已完成在小精灵执行攻击的过程中产生通信报文。

通信协议设计

```
/*
* 战斗通信格式
* <PlayerRound 0|1|2> <skillName> <dodge 0|1> <critical strike 0|1> + '\n'
* 本方小精灵信息
* <pokemonid> <name> <race>
* <battleatk> <battledfs> <battlehp> <battlespeed> <lv> <maxhp>
* <pp1> <pp2> <pp3> + '\n'
* 敌方小精灵信息
* <pokemonid> <name> <race>
* <battleatk> <battledfs> <battlehp> <battlespeed> <lv> <maxhp>
* <pp1> <pp2> <pp3>
*/

/*
* 战斗结算格式
* <PlayerRound 3>
* <winner 1|2>
* <pokemon1EP> <pokemon2EP>
*/
```

- PlayerRound 为0，1，2为战斗过程通信，PlayerRound为0、1代表本轮玩家的小精灵出招、对方小精灵出招，为2代表初始化战斗，只传输小精灵初始信息，为3代表战斗结束，进行结算
- 闪避、暴击为1表示成功，0表示失败
- 其他数据，客户端负责解析以及展示
- 因为是自动战斗，不需要玩家选择出招，因此出招次数不在客户端展示

战斗同步问题

客户端向服务端发送<battle> id1 id2发起战斗，服务端返回两个玩家的信息。

为了使战斗与展示同步，战斗控制器BattleController的构造函数中应该包含参数Endpoint 的Socket，每次执行完攻击之后，在战斗控制器内部调用Socket::recv来阻塞战斗流程

当进入战斗流程，服务器每次发送战斗报文给客户端，之后调用Socket::recv来监听客户端返回的消息。客户端进行分析数据并展示。客户端所有操作完成后，向服务器发送“continue”。控制器收到”continue“才能继续往下运行。

徽章系统

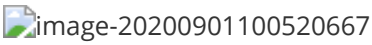
在查看宠物界面添加勋章的展示。勋章展示规则如下：

- 精灵爱好者徽章
  - 金质 - 精灵数量不少于20
  - 银质 - 精灵数量不少于10
  - 铜质 - 精灵数量不少于5
- 精灵大师徽章
  - 金质 - 15级精灵数量不少于5
  - 银质 - 15级精灵数量不少于3
  - 铜质 - 拥有15级精灵

新增变量showBadge控制徽章是否展示，如果是对战选择精灵则无法看到徽章，只有查看玩家自己或其他玩家的小精灵时才会展示徽章

图标来自www.iconfont.cn

展示界面：



胜率系统

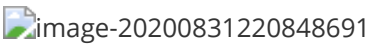
在数据库Login表中添加字段

```
<win_round> <total_round>
```

前者记录获胜场次，后者记录总场次。

登录时，服务器从数据库中获得该玩家的胜场和总场次，并通过Endpoint::setWinRound()和Endpoint::setTotalRound()在Endpoint中存储用户的胜场。每次战斗结束后，Endpoint对胜场和总场次进行更新。

用户查询精灵列表时，再次从数据库请求胜场和总场次，并计算胜率。展示情况与徽章相同，共用showbadge进行判定是否展示。



遇见的问题

1. **TCP数据流的长度限制**

   在程序的测试环节中，我发现当随着小精灵数量的上升到一定个数，在对战过程中会出现程序的崩溃。因此我对程序设置断点进行了调试，发现在进入对战界面后，客户端收到的数据仍保留着先前操作（查看小精灵列表）的数据，而先前查看小精灵列表的数据并没有展示完全。因此我发现，TCPSocket的发送长度限制出现了问题。因此我调整了长度限制的宏定义DEFAULT_BUFLEN，将其从1024调整为2048，虽然可以接收到所有数据了，但是会出现其他的问题。因此我的解决方案就是限制小精灵的个数，将获取精灵列表的字符串的长度限定在1024以下。