

YAPOKEMON:宠物小精灵对战游戏

面向对象程序设计(C++)课程设计(2020年春)

徐逸辰^{1,2}

¹ 北京邮电大学,计算机学院

²2018213555

linyxus@bupt.edu.cn

目录

第一部分 引言	2
第二部分 功能展示	3
第三部分 设计与实现	8
第四部分 开发与测试	21
第五部分 总结与展望	22
第六部分 附录	23

第一部分 引言



图 1 YaPokemon 欢迎界面

面向对象编程 (*Object-Oriented Programming*) 是一种成熟而使用广泛的编程范式。OOP 通过将程序功能与状态的集合抽象为类和对象，提升了程序设计的效率与有序性，能够提高开发效率与代码质量。OOP 成熟而广泛地应用于多种语言与很多现代软件中，C++、Java 乃至 Python 等在当代编程业界举足轻重的编程语言，都以 OOP 为主要设计范式，或是兼容、包含了 OOP 的设计方法。

在本次课程设计中，我将在实践中使用 OOP 的设计思想，设计、实现、调试一个宠物小精灵对战游戏。并且实现如下基本功能：

1. 实现用来描述宠物小精灵各类状态的类结构体系，能够模拟小精灵的不同种类的升级、出招对战等等。
2. 实现一个客户端/服务器 (C/S) 模型，师兄用户的登录、注册，保存用户拥有的小精灵信息。
3. 实现用户小精灵与服务器小精灵的对战，并且实现小精灵的失去与获得等逻辑。增加用户勋章系统。增加用户胜率系统等。

在课程设计中，我实现了一个命名为 YaPokemon¹的宠物小精灵对战游戏。YaPokemon 除了实现了题目要求的所有基本功能之外，还实现了额外的这些功能，或者具有如下值得一提的亮点：

1. 由 Qt 和 Qml 实现的图形化用户界面。图 1 中给出了 YaPokemon 在进入游戏时的欢迎界面。YaPokemon 的 GUI 除了使用了在互联网上寻找到的图片资源²之外，大多数控件都通过撰写 Qml 进行自定义绘制。因而 YaPokemon 具有统一而美观的界面。

¹YaPokemon: Yet another Pokemon game.

²YaPokemon 使用的图片资源仅仅用于课程学习交流，不会用作任何商业用途。

2. 使用 UDP 实现了服务器-客户端之间的通信. 使用预先约定的 Json 字符串作为通信的载体,提升了开发效率,降低了调试难度.
3. 实现了基于 JWT (*Json Web Token*) 的用户鉴权方式. 在用户注册与登录模块中,使用 Hash 存储用户密码,并且加入了加盐机制,保证了用户安全性.
4. 自实现了基于 Json 的文件数据库. 适用于低并发、低负载场景,并且实现了高效、简明、美观的 API 接口. 下面的代码样例实现了对用户名为 **test** 的用户的令牌的修改.

```

1 db.table("users")
2 .where(_x_["username"] == "test")
3 .update({{"token", "NEW_TOKEN"}});

```

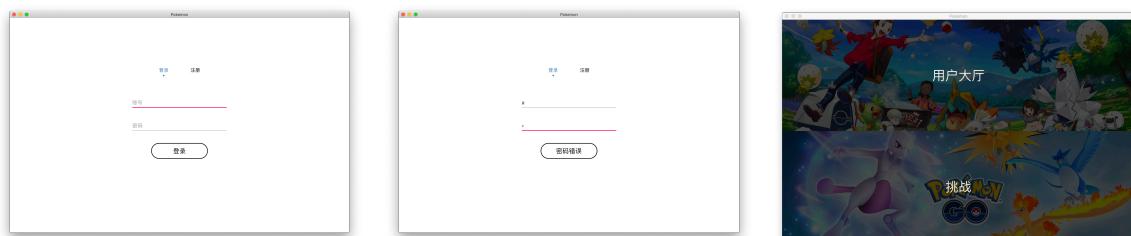
5. 在类设计方面,采用了层次化的类设计方法,并且将技能抽象为一个独立的类,使得游戏设计更为灵活,也提高了代码的可复用性. 除此以外,在游戏机制中增加了增益机制,也同时设计了与其相关的类结构.

在本报告中,我将首先对 YaPokemon 实现的各种功能进行展示. 包括用户界面、用户登录、查看用户信息、小精灵信息、小精灵对战等等. 随后,对各个功能模块的设计与实现进行展示与阐述. 接着,对 YaPokemon 开发使用的开发环境与工具进行描述,以及对 YaPokemon 的单元测试进行描述. 最后,对本次课程设计进行总结、思考与展望.

第二部分 功能展示

1 基本功能

在本节中,对 YaPokemon 的基本功能进行解释与展示.



(a) 登陆界面

(b) 登录失败提示

(c) 用户大厅

图 2 用户登录相关界面展示

用户登录 图 2 中给出了用户登录相关的交互界面. 注意输入框上方的按钮控制了进行登录还是进行注册. 当登录失败时,会在按钮上显示出错误信息,当登录成功时,会进入如图 2c 中所示的游戏大厅.



图 3 用户注册相关界面展示

用户注册 图 3 中给出了用户注册相关的交互界面. 可见,当用户注册密码为空或者用户名重复时,都会注册失败,在按钮上给出错误提示. 当注册成功时,将会跳转到登录界面,再点击登录即可进入用户大厅.

游戏大厅 图 2c 中的游戏大厅能够进行游戏中功能的导航. 点击上方的用户大厅按钮,则会进入用户大厅,显示用户列表. 点击下方的挑战按钮,则进入挑战界面,可以选择服务器上的宝可梦进行练习赛或者挑战赛.



图 4 用户信息相关界面展示

用户大厅 图 4a 中给出了展示用户列表的交互界面. 可见,用户列表中列出了所有用户,包括其用户名、胜率、拥有精灵个数等等. 用户名旁边的圆点标示了用户当前在线状态. 点击用户列表中的某一项,可以进入该用户的详情页面. 点击上方我的信息,则可以直接进入自己的详情页面.

用户详情 图 4b 中给出了用户详情页面. 用户详情页面中给出了用户名、用户徽章等信息. 并且给出了用户拥有的小精灵列表. 列表中的每一项给出了小精灵的名称、等级、各项属性信息. 点击某一项,则可以进入小精灵的描述界面.

小精灵描述界面 图 4c 中给出了小精灵描述界面. 界面中给出了小精灵名称与小精灵的技能列表. 对于每个技能, 给出了技能名称、技能类型与技能描述.



图 5 挑战与对战相关界面展示

挑战界面 图 5a 给出了 YaPokemon 的挑战界面. 挑战界面中给出了服务器小精灵列表. 用户可以选择与他们进行练习对战或进行挑战赛. 进行练习对战可以获取经验, 失败没有惩罚. 进行挑战赛可以获得经验且获胜可以直接获得一只被挑战的小精灵, 若失败则会送出参战小精灵.

对战界面 图 5b 中给出了 YaPokemon 的对战界面. 界面中包含了对战动画, 小精灵的生命值、名称、等级信息, 以及对战实时播报. 下方的实时播报中会显示小精灵的出招情况, 上方的显示了当前回合, 实时显示小精灵信息. 在战斗完成后, 会显示如图 5c 中所示的战斗结果界面. 点击返回大厅, 则会返回到挑战界面.

2 宠物小精灵与对战机制

在本节中, 将介绍 YaPokemon 宠物小精灵的种类与信息. 同时简单介绍 YaPokemon 的对战机制, 技能设计与增益机制.

2.1 宠物小精灵

YaPokemon 设计的小精灵机制中, 小精灵的能力共有 6 个属性维度, 7 种小精灵种类. 每种小精灵种类有 2 种小精灵, 共有 14 种小精灵. 具体的小精灵列表可见附录 A.

小精灵能力 小精灵的能力属性分为 6 个维度: 体力、攻击、防御、特殊攻击、特殊防御、速度. 各个维度的作用与含义如下所示:

- **体力.** 体力也就是小精灵的生命值. 各类技能可以伤害小精灵的生命值, 对战中当某一方生命值降为 0 时, 那一方战斗失败.
- **攻击.** 攻击折算为造成攻击伤害的技能伤害. 折算的比例由技能威力决定.

- **防御.** 防御可以抵消宝可梦收到的攻击伤害, 防御越高, 抵消的比例越大.
- **特殊攻击.** 特殊攻击可以折算为小精灵造成特殊攻击伤害的技能伤害. 折算的比例同样由特殊攻击的技能威力决定.
- **特殊防御.** 特殊防御同样可以按比例抵抗小精灵收到的特殊攻击伤害.
- **速度.** 双方宝可梦的速度决定了两方宝可梦的出招顺序. 除此以外, 速度也可以提高闪避率.

小精灵的能力会随着等级的提升而增长.

等级与属性增长 YaPokemon 固定规定小精灵每获取 50 经验值就提升一级等级. 最低等级为 0 级, 最高等级为 15 级. 小精灵的属性由两方面决定: 基础值与成长值. 假设某一只小精灵具有基础值 $b \in \mathcal{H} = \mathbb{N}^6$, 其中 \mathbb{N}^6 为一个 6 维向量, 对应了小精灵的六种属性; 且具有成长值 $g \in \mathcal{H}$, 则当其等级为 l 时, 其属性 $c = b + l \cdot g$.

小精灵种类 YaPokemon 中共设计了 7 中小精灵种类. 分别为: 神圣型, 普通型, 攻击型, 特殊攻击型, 生命型, 防御型, 速度型. 小精灵的种类决定了其成长值. 不同种类的小精灵, 成长值各有不同的侧重. 不同种类的背景分别如下:

1. **神圣型.** 神圣型宝可梦为 YaPokemon 中的神兽. 具有很高的属性增长, 并且拥有非常强力的技能.
2. **普通型.** 普通型宝可梦为发展均衡的宝可梦. 各个属性的增长时一致的.
3. **攻击型.** 攻击型宝可梦具有更高的攻击.
4. **特殊攻击型.** 特殊攻击型宝可梦具有更高的特殊攻击.
5. **生命型.** 生命型宝可梦具有更高的生命值.
6. **防御型.** 防御型宝可梦具有更高的双防.
7. **速度型.** 速度型宝可梦具有更高的速度和较高的攻击.

各类小精灵具体的基础值, 可见附录 A 中的表 2.



图 6 YaPokemon 中的小精灵

小精灵 YaPokemon 的 7 种小精灵种类每种分别由 2 种小精灵. 小精灵的种族决定了小精灵属性的基础值, 同时也决定了小精灵能够使出的技能. YaPokemon 中设计的小精灵种族共有如下这 14 种: 苍响、撼然玛特、伊布、百变怪、铝钢龙、卡比兽、钢铠鸦、双斧战龙、喷火龙、皮卡丘、拉普拉斯、霸王花、风速狗、多龙. 小精灵在游戏中的形象³如图 6 中所示. 各个小精灵的具体属性基础值在附录 A 中的表 3 详细列出.

2.2 小精灵招式与增益系统

增益系统 YaPokemon 在对战中加入了增益 (*Buffer*) 系统. 增益系统能够在战斗中增加、削弱己方、对方小精灵的各项属性, 也能够增强、削弱招式的效果. 增益具有有效回合数, 并且可以由小精灵招式施加在自己或对手身上. 增益的具体例子, 在下文对招式的介绍中给出.

小精灵招式 YaPokemon 将小精灵的招式抽象为一个单独的类, 而不是作为一个虚成员函数来实现. 每个招式都具有一种类型: 如草、电、风、龙等等. 而每个招式在执行时, 都会产生一定的行为. 招式产生的行为也被抽象为了一个类, 其目的是避免不必要的冗余. 具体的设计细节会在下文给出. 下面, 介绍一些典型的招式.

典型招式 下面给出一些 YaPokemon 中实现的典型的小精灵招式.

1. **巨兽斩.** 攻击类招式. 能够给对方宝可梦造成攻击伤害.
2. **冷冻光线.** 特殊攻击类招式. 给对方宝可梦造成伤害的同时, 还会为对方增加一个降低速度的增益.
3. **终极吸取.** 特殊攻击类招式. 给对方造成特殊攻击伤害, 且恢复自身造成伤害一半的生命值.
4. **剑之舞.** 增益型招式. 大幅提升自己的攻击.
5. **祈雨.** 增益型招式. 对于全场小精灵, 大幅提升他们的水类招式伤害, 降低火类技能伤害.
6. **绝对零度.** 特殊招式. 一定概率秒杀对手.
7. **灭亡之歌.** 特殊增益招式. 为对手增加一个灭亡增益, 在四回合后若没有击败我方宝可梦, 则直接失败.

可见, 通过将技能抽象为一个独立的类 (*Move*), 并增加增益机制, 让 YaPokemon 的招式机制与对战机制变得非常灵活多变, 能够增加许多有趣的设计, 具有很丰富的表达能力.

³ 此处的形象资源来自于 Nintendo 公司开发的宝可梦: 剑盾. YaPokemon 不会以此获取任何商业利益.

3 服务器功能

除了 YaPokemon 的小精灵与游戏机制之外, YaPokemon 的服务器端也具有非常丰富功能.

完善的用户认证系统 YaPokemon 的服务器端设计了完善的用户认证系统. 首先, YaPokemon 在数据库中存储用户密码经过 **SHA256** 算法生成的哈希值, 保证了用户密码的安全. 同时, 服务器也会为用户密码加盐, 以防止弱密码带来的可预测的哈希值. 由于使用 UDP 进行通信, 服务端借助 **JWT** 生成用户令牌实现了无状态的通信. 如图 7 中是 JWT 令牌的基本结构. JWT 中会保存认证的用户账号, 认证的时间戳等等, 并且添加一个服务器签名来验证 JWT 令牌的真伪.

自实现的 Json 文件数据库 YaPokemon 具有一个自实现的 Json 文件数据库, 命名为 YaDB⁴. YaDB 具有优雅美观的 API 调用, 适用于低负载、低并发场景, 并且能够很大大地提升开发效率.

基于 UDP 与 Json 进行通信. YaPokemon 的客户端与服务端直接通过 UDP 进行通信. 相较于 TCP, UDP 具有更少的前置损耗, 因而具有更低的响应时间, 在用户体验上, 进行登录、查看信息、对战等操作的延迟时间会更加短暂. 除此以外, YaPokemon 通过 Json 来编码报文、约定报文格式. 这提升了开发效率, 降低了调试成本.

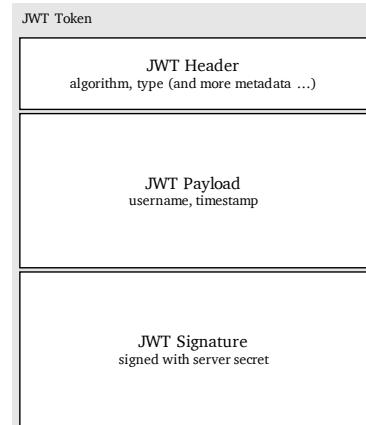


图 7 JWT 令牌结构

第三部分 设计与实现

在本节中, 将详细阐述 YaPokemon 的设计与实现. 分为三大部分来阐述: 宠物小精灵核心库, 包含了小精灵类的设计, 招式类的设计, 增益类的设计, 小精灵对战类的设计; 服务器设计, 包含了 YaDB 的设计方式, 用户类的设计, 服务器类的设计; 用户界面, 包含了 Qml 设计的简单描述, 以及客户端的设计与 Qml 数据模型类的设计.

4 宠物小精灵核心库

4.1 小精灵类的设计

首先, 对小精灵类的设计进行阐述.

⁴YaDB: Yet Another DataBase.

小精灵数据的层次 每只小精灵拥有的状态数据,至少拥有三个层次,他们分别是:

1. 种族相关的数据. 包括小精灵名称、各项数据的基础值与成长值、能使用的招式.
2. 小精灵个体相关数据. 包括小精灵当前经验值.
3. 小精灵在状态中的实时状态设计. 包括小精灵当前拥有的增益,小精灵当前属性.

若将这些状态都糅合入一个类中,虽然能够实现所有功能,却在设计上不够美观协调,在使用中也会产生浪费——有时我们只希望操作小精灵的种族,却不得不保存包括个体、战斗状态在内的所有数据. 这是不合理的.

类	名称	基础值	成长值	招式	经验值	增益	当前属性
Pokemon _{all_in_one}	✓	✓	✓	✓	✓	✓	✓
PokemonTemplate	✓	✓	✓	✓			
Pokemon	✓	✓	✓	✓	✓		
PokemonInstance	✓	✓	✓	✓	✓	✓	✓

表 1 不同类包含的状态信息

层次化小精灵类的设计 为了避免上面提到的不合理之处,我将小精灵类分为三个层次:

1. **PokemonTemplate** 小精灵种族类,存储了小精灵种族决定的信息:名称、基础值、成长值、招式.
2. **Pokmeon** 小精灵个体类,存储了某一小精灵个体的信息,包括该小精灵的种族和该小精灵的经验值.
3. **PokemonInstance** 小精灵实例类,为某个小精灵个体在某场战斗中的实例,除了对应的小精灵信息外,还存储了小精灵在当前战斗中拥有的增益和属性信息.

在有了上面的设计想法之后,可以实现如下的虚基类.

```

1 class BasePokemonTemplate {
2 public:
3     BasePokemonTemplate() = default;
4     ~BasePokemonTemplate() = default;
5
6     virtual PokemonId id() const = 0;
7
8     virtual Hexagon<llint> growth() const = 0;
9     virtual Hexagon<llint> base() const = 0;
10
11    virtual std::string name() const = 0;
12    virtual PokemonType type() const = 0;
13

```

```

14     virtual std::vector<shared_ptr<Move>> moves() const = 0;
15 };
16
17 class Pokemon {
18 public:
19     Pokemon() = default;
20     ~Pokemon() = default;
21
22     virtual std::string name() const = 0;
23     virtual llint level() const = 0;
24     virtual Hexagon<llint> max() const = 0;
25
26     virtual void learn(llint exp) = 0;
27     virtual const BasePokemonTemplate *temp() const = 0;
28 };
29
30 class PokemonInstance {
31 public:
32     PokemonInstance() = default;
33     ~PokemonInstance() = default;
34
35     virtual const Pokemon *pokemon() const = 0;
36     virtual Hexagon<llint> current() const = 0;
37
38     virtual void accept_action(shared_ptr<Action> action, bool
39         critical = false) = 0;
40     virtual void accept_actions(const Actions &actions) {
41         for (const auto &act : actions) {
42             accept_action(act);
43         }
44     }
45     virtual vector<shared_ptr<Buff>> buff() const = 0;
46     virtual void update_buff() = 0;
47 };

```

可见,三个虚基类分别定义了属于各个层级的小精灵类需要给出的接口. 而各个层级只保管他们负责的状态,并能给出指向上一个层级的引用. 图 1 给出了层次化小精灵类与朴素小精灵类管理的数据的区别.

进一步地,对于每个小精灵种族类,可以通过定义模板类来生成属于他们的小精灵个体类和实例类. 具体地,考虑下面的代码.

```
1 template <typename T>
2 class PokemonOf : public Pokemon {
3 public:
4     PokemonOf();
5     ~PokemonOf() = default;
6
7     std::string name() const override;
8     llint level() const override;
9     Hexagon<llint> max() const override;
10
11    void learn(llint exp) override;
12    const BasePokemonTemplate * temp() const override;
13 private:
14     T _template;
15     llint _exp;
16 };
17
18 template<typename T>
19 class PokemonInstanceOf : public PokemonInstance {
20 public:
21     explicit PokemonInstanceOf(const PokemonOf<T> &pokemon);
22     ~PokemonInstanceOf() = default;
23
24     const Pokemon *pokemon() const override;
25
26     Hexagon<llint> current() const override;
27
28     void accept_action(shared_ptr<Action> action, bool
29                         critical) override;
30
31     vector<shared_ptr<Buff>> buff() const override;
32     void update_buff() override;
33
34     shared_ptr<PokemonInstance> clone() const override;
35
36 private:
37     PokemonOf<T> _pokemon;
38     Hexagon<llint> _current;
39     vector<shared_ptr<Buff>> _buff;
40 }
```

上面定义的模板类，可以生成小精灵种族类所对应的个体类与实例类。举个例子，若

描述了伊布的小精灵种族类**Eevee**为**BasePokemonTemplate**的派生类, 则通过**PokemonOf<Eevee>**和**PokemonInstanceOf<Eevee>**可以产生伊布的小精灵个体类和实例类.

小精灵种族类的层次 如之前所说的, 小精灵的种族可以分为 7 个种类, 而不同的种类决定了小精灵各项属性种类的成长值. 因而在种族类的设计中, 从**BasePokemonTemplate**基类派生出各个种类的小精灵种族类. 如**SuperPokemonTemplate**, **NormalPokemonTemplate** 等等. 各个种类的小精灵种族类定义了小精灵的成长值, 但是其仍然是一个纯虚基类, 无法被实例化, 因为没有定义小精灵的名称和基础值.

在获得各个种类的小精灵种族类后, 进一步进行继承派生, 可以得到具体的小精灵类, 如继承自**NormalPokemonTemplate**的伊布类**Eevee**.

4.2 小精灵招式类的设计

YaPokemon 的类设计中, 将小精灵招式也抽象为一个独立的类. 首先给出招式纯虚基类的接口定义:

```

1 class Move {
2 public:
3     Move() = default;
4     ~Move() = default;
5
6     virtual Actions move(shared_ptr<PokemonInstance> self,
7                           shared_ptr<PokemonInstance> target) = 0;
8     virtual string name() const = 0;
9     virtual string desc() const = 0;
10    virtual MoveCat move_cat() const = 0;
11 };

```

可见, 招式类需要实现的接口包含: 行为函数, 返回一个行为, 定义了招式施展时的行为; 招式名称; 招式描述与招式类型.

行为类的引入 可以发现, 招式类的接口中, **move**函数返回的是一个**Action**类的值. 下面, 首先解释为什么需要设计一个额外的**Action**类.

首先考虑没有**Action**类的情形. 假设**move**函数的类型签名变更为:

```

1 virtual void move(shared_ptr<PokemonInstance> self,
2                   shared_ptr<PokemonInstance> target) = 0;

```

可见, 返回值变为空. 于是这个函数需要直接作用于自身或对手, 修改小精灵实例的属性与状态. 然而, 小精灵的状态如当前生命值, 当前拥有的增益效果等等, 往往是私有的, 为了能

够修改这些私有状态，则要求**Move**类为所有小精灵实例类的友元。这样实现是可行的，但显然不够优雅，增加了友元。而事实上，**YaPokemon** 所有面向对象的实现中不包含任何一个友元。**Action**类最主要的目的，一方面是增加代码模块化程度，提高可复用性，另一方面就是避免此类友元的引入。

事实上，我们发现招式带来的效果其实可以简单归纳为几种类型：造成攻击伤害、造成特殊攻击伤害、直接修改血量（恢复生命值或秒杀）、增加增益效果。因而，**YaPokemon** 将招式带来的效果抽象为**Action**类，招式在施展时返回多个**Action**，再为**PokemonInstance**类定义**accept_action**函数，将**Action**描述的行为作用在自己身上。借此，便实现了不包含友元的招式机制。

工具招式类 可以发现，招式事实上存在着许多主要的类别，如攻击招式、增益招式等等。为了提升开发效率，**YaPokemon** 中定义了一些特定类别的工具招式类，在定义招式时，直接从这些类派生并且在初始化时给出必要的信息，就可以直接定义对应类型的招式，提升了代码的可复用性。下面给出攻击招式基类的定义作为例子：

```

1 class AttackMove : public Move {
2 public:
3     AttackMove(llint power, MoveCat cat);
4     ~AttackMove() = default;
5
6     Actions move(shared_ptr<PokemonInstance> self, shared_ptr<
7         PokemonInstance> target) override;
8     Actions attack_action(const shared_ptr<PokemonInstance>&
9         self, ActionTarget t = ActTarget);
10    virtual MoveCat move_cat() const override { return cat; }
11    llint power;
12    MoveCat cat;
13}
```

在定义了上述类之后，在需要定义新的简单攻击招式时，只需要继承**AttackMove**类，并在初始化时给出招式威力和类型即可。

4.3 增益类的设计

YaPokemon 的增益机制中，增益可以改变宝可梦的属性，也可以改变宝可梦招式的效果。具体地，增益类定义两个函数**map_current**与**map_action**，分别改变宝可梦当前属性与招式产生的效果。其具体机制类似于一些成熟服务器框架中的中间件 (*middleware*)，在计算对局中小精灵的当前属性时，会按照增益获得的顺序，用增益定义的接口函数修改当前属性；在小精灵施展招式之后，也会类似地变换招式返回的效果。

具体的接口定义如下所示：

```

1 class Buff {
2 public:
3     Buff() = default;
4     ~Buff() = default;
5
6     virtual void map_action(const std::shared_ptr<Action> &
7         action) = 0;
8     virtual Hexagon<llint> map_current(const Hexagon<llint> &
9         current) = 0;
10    virtual bool expire() const = 0;
11
12    virtual string name() const = 0;
13
14    virtual shared_ptr<Buff> clone() const = 0;
15 };

```

4.4 对战类的设计

YaPokemon 设计了一个对战类管理小精灵的对战状态。对战类的定义如下所示：

```

1 class Battle {
2 public:
3     Battle(const shared_ptr<Pokemon>& left, const shared_ptr<
4             Pokemon>& right);
5     BattleRound proceed();
6     BattleStatus check() const;
7     PokemonTurn next_turn() const;
8     PokemonTurn turn_at(llint i) const;
9     shared_ptr<PokemonInstance> left() const;
10    shared_ptr<PokemonInstance> right() const;
11    llint turn_count() const;
12
13    static int get_exp(int this_level, int that_level);
14    int exp_gain() const;
15 private:
16    shared_ptr<PokemonInstance> _left;
17    shared_ptr<PokemonInstance> _right;
18    llint _turn_count;
19 };

```

可见,对战类在实例化时需要给出两只小精灵,对战类会将他们实例化,得到两个小精灵实例,准备开始对战. 同时,对战类给出了接口,可以获得对战中的主要信息: 对战状态(是否仍在进行中,获胜者是哪一方)、下一个出招小精灵、当前回合数、当前两方小精灵的实例状态. 对战类也给出了proceed函数,让对战进展一个回合.

YaPokemon 中也定义了一个**BattleHistory**类, 来方便地管理对战的出招和状态历史,其定义如下:

```

1 class BattleHistory {
2 public:
3     explicit BattleHistory(Battle battle);
4     BattleRound proceed();
5     const Battle &battle() const;
6     void complete();
7     const QList<BattleStep> &history() const;
8     BattleStatus result() const;
9     static BattleResult run_battle(const shared_ptr<Pokemon> &
10                                left, const shared_ptr<Pokemon> &right);
11
12 private:
13     Battle _battle;
14     QList<BattleStep> _history;
15     BattleStatus _result;
16 };

```

可见,对战历史类为一个辅助类,可以接受一个对战类,并管理其每一步的历史. **BattleHistory**类也给出了一个静态函数,可以直接获得两只小精灵进行对战的所有历史信息. 这在服务器端非常有用.

5 服务器设计

5.1 YaDB 的类设计

YaPokemon 实现了一个简单的,基于Json文件的文件数据库,称为 YaDB. 这一节中,将介绍 YaDB 类的实现. 首先给出 YaDB 类的定义:

```

1 class Yadb {
2 public:
3     explicit Yadb(QString path);
4     ~Yadb();

```

```

5
6     QString path() const;
7     void sync() const;
8
9     Yadb &table(const QString& table_name);
10    int insert(json record);
11
12    Subset set();
13    QVector<json> all();
14    json get();
15    Subset where(const shared_ptr<Predicate> &pred);
16    void update(const json &j);
17    int count();
18    bool exists(const shared_ptr<Predicate> &pred);
19
20    inline json raw_json() const { return _data; }
21 private:
22     QString _path;
23     json _data;
24     QString _table_name;
25 };

```

从定义中可以看到：

1. YaDB 类在初始化时，接受一个文件路径，并读取这一路径的 Json 文件。注意，若路径不存在，则会创建一个包含 `_default` 表的初始文件。
2. `sync` 函数将内存中的 Json 数据写入文件。
3. `table` 函数切换当前的表。
4. YaDB 给出了 `all`, `where`, `update`, `count`, `exists` 等函数，可以方便地对数据表中的记录进行操作。

具体地，YaDB 操作的 Json 文件具有下面的格式：

```
{
  "_default": {
    "_count": 10
    "_data": [ ... ]
  },
  "other_table": { ... },
  ...
}
```

可见,Json 内部包含了多张数据表,且至少包含一张`_default`表,每张表包含了`_count`元信息与存储的数据记录的数组.

数据的选择与操作 YaDB 设计了非常美观高效的 API,下面是一个操作用户数据的例子:

```
1 db.table("users").where(_x_[ "pokemon_count" ] > 10)
2     .update({{"badge", "silver"}});
```

为了实现这样美观的 API,YaDB 实现了三个辅助类:`Subset`,`Selector`,`Predicate`.

- **Subset** 这个类描述了数据记录的一个子集, 使用`where`能够从一个子集中选取一个更小的自己进行操作.
- **Selector** 上面的示例代码中的`_x_["pokemon_count"]`就是一个`Selector`. 这个类描述某条 Json 数据记录的特定字段. `_x_`是一个预先定义好的选择器, 它选择数据记录本身. 而重载的`[]`运算符会从一个选择器生成一个新的选择器, 这个选择器选择上一个选择器选择的 Json 对象的一个字段. 举个例子, 对于下面的 Json 数据对象:

```
{ "a": { "b": { "c": 1 } } }
```

选择器`_x_`会选择它本身, 而`_x_["a"] ["b"]`则会选择`{"c" : 1}`, 进一步地,`_x_["a"] ["b"] ["c"]`会选择`1`.

- **Predicate** `where`函数接受的就是这个类. 这个类描述一个谓词, 判断某条数据记录是否符合条件. 重载运算符`<`,`>`,`==`,`!=`等等可以从一个选择器和一个常量生成谓词.

5.2 服务器类设计

YaPokemon 设计了一个服务器类来方便管理服务器的状态, 实现服务器的功能. 服务器类的定义如下:

```
1 class PokemonServer : public QObject {
2     Q_OBJECT
3 public:
4     explicit PokemonServer(QObject *parent, quint16 port,
5                             QString db_path = "db.json");
6 private:
7     bool user_register(const QString& username, const QString
8                         &password);
```

```
8  bool verify_user(const QString &username, const QString &
9    password);
10 QByteArray jwt_for_user(const QString &username);
11 QString get_user(const QByteArray &token);
12 QString check_req_auth(json payload);
13 void pulse(const QString &username);
14 int inactive_duration(const QString &username);
15 pair<int, int> win_lose_count(const QString &username);
16 BattleResult run_battle(int pokemon_id, int boss_id, int
17   boss_level);
18 BattleResult exe_battle(int pokemon_id, int boss_id);
19 BattleResult real_battle(int pokemon_id, int boss_id);
20 bool remove_user_pokemon(const QString &username, int
21   pokemon_id);
22 void add_user_pokemon(const QString &username, int pid);
23 bool verify_user_pokemon(const QString &username, int pid)
24   ;
25 void pokemon_learn(int pid, int exp);
26
27 int create_pokemon(PokemonId pid);
28 shared_ptr<Pokemon> get_pokemon_by_id(int pid);
29 static shared_ptr<Pokemon> pokemon_from_info(PokemonId pid
30   , int exp);
31 static shared_ptr<Pokemon> pokemon_from_json(json obj);
32
33 static QByteArray compose_error_resp(QString error_msg);
34 static QByteArray compose_succ_resp(const json&
35   succ_payload = json::object());
36
37 json compose_user_json(const QString &username);
38 json compose_user_list();
39 json compose_pokemon_list();
40 json compose_boss_list();
41 static json compose_pokemon_instance(const shared_ptr<
42   PokemonInstance> &instance);
43 static json compose_battle_round(const BattleRound &round)
44   ;
45 static json compose_battle_step(const BattleStep &step);
46 static json compose_battle_result(const BattleResult &
47   result);
48
49 QByteArray user_register_handler(json payload);
50 QByteArray user_auth_handler(json payload);
```

```

42     QByteArray user_check_auth_handler(json payload);
43     QByteArray user_list_handler(const json& payload);
44     QByteArray pokemon_list_handler(const json& payload);
45     QByteArray pokemon_get_id(const json &payload);
46     QByteArray battle_exe_handler(const json& payload);
47     QByteArray battle_real_handler(const json& payload);
48     QByteArray battle_list_boss_handler(const json& payload);
49     void message_handler();

50
51     yadb::Yadb _db;
52     QUdpSocket _socket;
53     const QByteArray jwt_secret = "19je9dhxplsa9";
54     QVector<PokemonId> _boss = {
55         PokemonZacian,
56         PokemonZamazenta,
57         PokemonLapras,
58         PokemonDragapult,
59         PokemonDuraludon,
60         PokemonHaxorus,
61         PokemonArcanine,
62         PokemonCharizard,
63         PokemonCorviknight,
64         PokemonEevee,
65         PokemonDitto,
66         PokemonVileplume,
67         PokemonSnorlax,
68         PokemonPikachu,
69     };
70 }

```

可见,服务器类主要定义了这些功能与状态:

1. 定义了一个 Qt 的 UDP 句柄. 监听服务器端口. 且定义了一些信息处理器来处理对应类型的信息. 对于每个功能, 都定义了相应的处理器. 总的处理器**message_handler**为一个槽, 与 UDP 句柄的新报文信号连接, 会根据 Json 报文类型, 选择对应的处理器.
2. 定义了服务器进行 JWT 签名的密钥. 这个密钥应该绝对保密, 才能保证用户令牌不被伪造.
3. 定义了服务器上的挑战赛给出的小精灵类型.

同时, 服务器端也实现了几个工具函数, 用于 JWT 令牌的验证、生成与解码:

```

1 QByteArray encode_jwt(const json &payload, const QByteArray &
2   secret);
3
4 bool verify_jwt(const QByteArray &jwt, const QByteArray &
5   secret);
6
7 json decode_jwt(const QByteArray &jwt);

```

6 用户界面

6.1 客户端类的设计

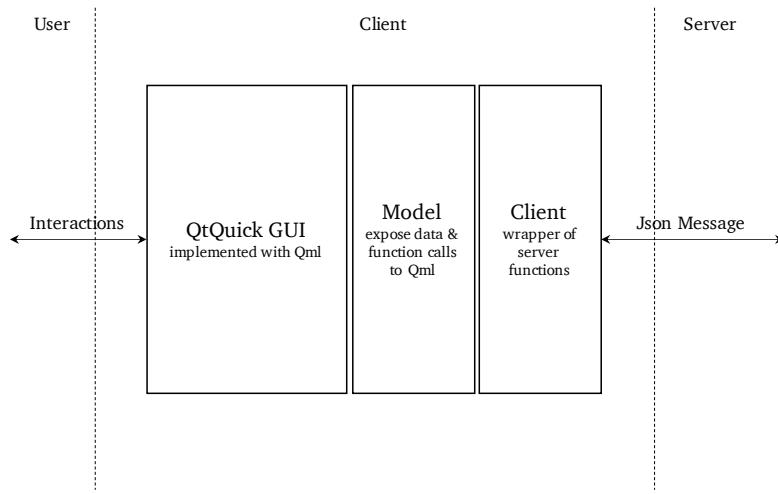


图 8 客户端模块结构

在客户端部分, YaPokemon 实现了两个客户端类, 分别负责不同的功能:

- **PokemonClient** 客户端连接类, 负责封装服务器给出的 API, 与服务器通信;
- **ClientModel** 客户端数据模型类, 负责调用连接类, 将各类数据与接口暴露给 Qml, 实现用户界面与底层逻辑的桥接.

客户端的模块结构图如图 8 中所示. 由 QtQuick Qml 实现的用户界面负责与用户交互, 展现图形化界面; 数据模型类负责桥接用户界面与逻辑, 将显示数据与函数调用暴露给 GUI; 而客户端连接类负责与服务器通信, 包装了服务端提供的各项功能调用.

其中, 数据模型需要成为一个单独的类, 是由于 Qml 对暴露给它的接口有着特定的要求. 数据模型类的主要功能便是将客户端数据封装为符合 Qml 要求的 Qt 类型, 将特定的数据与调用封装为 Qt 对象的属性, 封装可供 Qml 调用的函数接口等等.

6.2 用户界面的实现

用户界面是通过 Qt 提供的 QtQuick API 接口与 Qml 绘制实现的. Qml 是 QtQuick 提供的基于 OpenGL 的用户界面绘制接口, 具有很高的性能与很大的灵活性. YaPokemon 实现的用户界面全部是由 Qml 绘制完成的.

不仅如此, YaPokemon 封装、实现了风格化的, 一致的界面空间. 如带有动画效果的圆角游戏按钮, 列表项目等等. 这样的封装一方面保证了图形化界面一致美观, 另一方面提升了开发效率.

第四部分 开发与测试

7 开发与构建环境

YaPokemon 项目在开发过程中使用的开发、构建环境如下:

- macOS Catalina 10.15.6
- fish (friendly interactive shell), version 3.1.2
- iTerm 2, 3.3.12
- CMake 3.17.2
- Ninja 1.10.0
- Conan 1.28.1
- CLion 2020.2.1
- Visual Studio Code 1.48.2
- Qt Creator 4.12.4

在开发中, 使用的主要集成开发环境为 CLion, 通过配置 CMake, 引入 Qt 相关组件库, CLion 可以提供代码导航, 自动补全, 语法高亮, 自动代码提示等丰富的功能. 能够极大地提升开发效率. 配置 CMake 进行 Qt 开发的具体细节可见附录 B. 而除了 CLion 以外, 我也主要使用 Qt Creator 进行 Qml 文件的开发, 凭借其丰富的语法高亮与代码提示, 也能大大地提升开发效率.

而在构建方面, 如前所述, 我使用了 CMake、QMake 与 Ninja 相关工具链对项目进行编译. 项目具体的编译方法可见附录 C.

```

PokemonTest.LearnAndLevelWorks (0 ms)
[ RUN ] PokemonTest.GrowthHworks
[ OK ] PokemonTest.GrowthHworks (0 ms)
[ RUN ] PokemonTest.Polyworks
[ OK ] PokemonTest.Polyworks (0 ms)
[ RUN ] PokemonTest.MaxWorks
[ OK ] PokemonTest.MaxWorks (0 ms)
[----] 5 tests from PokemonTest (0 ms total)

[----] 1 test from PokemonInstanceTest
[ RUN ] PokemonInstanceTest.AttackWorks
[ OK ] PokemonInstanceTest.AttackWorks (1 ms)
[----] 1 test from PokemonInstanceTest (1 ms total)

[----] 1 test from TestBattle
[ RUN ] TestBattle.BattleTestCase1
[ OK ] TestBattle.BattleTestCase1 (0 ms)
[----] 1 test from TestBattle (0 ms total)

[----] 1 test from YadbTest
[ RUN ] YadbTest.OpenWorks
[ OK ] YadbTest.OpenWorks (1 ms)
[----] 1 test from YadbTest (1 ms total)

[----] Global test environment tear-down
[=====] 8 tests from 4 test suites ran. (2 ms total)
[ PASSED ] 8 tests.
d/pokemon-core/b/bin $

```

图 9 通过所有单元测试的运行截图

8 单元测试

YaPokemon 项目在开发过程中使用 GoogleTest 对项目中主要的算法逻辑进行了测试. 测试的主要模块包含: YaDB、小精灵对战类、小精灵类、小精灵实例类. 共 4 个测试集, 8 个单元测试.

YaPokemon 能够通过所有的 8 个单元测试, 通过单元测试的截图如图 9 中所示.

第五部分 总结与展望

在本次课程设计中吧, 我设计、实现了一个功能丰富的宠物小精灵对战游戏 YaPokemon. YaPokemon 具有丰富、全面、有趣的小精灵对战机制, 一个功能全面的基于 UDP 的服务器, 且具有设计良好、表达力丰富的类设计实现.

在本次课程设计中, 我受益匪浅, 感触颇多. 在进行设计、实现的过程中, 我也遭遇了许多问题. 例如, 在开发环境方面, 由于使用 CMake 编译构建 Qt 的方法在互联网上的资源非常稀少, 导致一开始在配置工程的构建环境时, 就花费了许多时间进行摸索; 在类的设计方面, 软件中涉及需要把一个纯虚基类的指针指向的类复制一份的需求, 一开始的我不知如何实现, 采用了动态判断其子类类型, 进行转换后复制的做法, 开发效率和运行效率都很低. 后来改为在基类中增加一个 `clone` 接口, 作用是将自身复制一份, 并在子类中实现. 很好地解决了这个问题, 减少了许多模板代码.

通过解决在开发中遇到的问题, 以及通过在开发软件中进行的实践, 我更好地掌握了面向对象的程序设计技巧, 对类的设计方法, 基类接口的设计策略, 对各个类的功能划分都有了更为切实的经验与认知. 除此以外, 通过对一个较为完善的软件的设计, 我更好地掌握了 Qt 等业界成熟开发框架的掌握与了解, 习得了更多实践开发的经验. 对于使用 C++ 进行面向对象软件设计开发也有了更为扎实的掌握与理解.

回顾过去, 展望未来, 此次课程设计让我获益良多, 积累了更多的经验. 在接下来的学习中, 我要近一步巩固面向对象程序设计这一成熟而广泛使用的设计范式, 提升自己的软件开发能力.

第六部分 附录

附录 A 小精灵图鉴

种类	生命值	攻击	防御	特殊攻击	特殊防御	速度
神圣型	3	3	3	3	3	3
普通型	2	2	2	2	2	2
生命型	7	1	1	1	1	1
攻击型	1	5	1	1	2	2
特殊攻击型	1	1	1	5	2	2
防御型	2	1	4	1	4	1
速度型	1	3	1	2	1	4

表 2 小精灵种类与成长值

名称	种类	生命值	攻击	防御	特殊攻击	特殊防御	速度
苍响	神圣型	10	30	10	10	10	30
臧然玛特	神圣型	20	10	20	10	20	10
伊布	普通型	10	10	10	10	10	10
百变怪	普通型	10	10	10	10	10	10
铝钢龙	生命型	15	5	12	12	12	8
卡比兽	生命型	15	8	12	8	12	8
钢铠鸦	攻击型	8	20	8	5	8	15
双斧战龙	攻击型	8	20	8	5	8	12
喷火龙	特殊攻击型	15	5	5	18	6	15
皮卡丘	特殊攻击型	8	8	8	20	8	8
拉普拉斯	防御型	12	10	15	10	15	10
霸王花	防御型	12	5	15	8	15	8
风速狗	速度型	15	10	10	10	10	15
多龙	速度型	5	20	5	5	5	20

表 3 小精灵与基础值

附录 B CMake 配置 Qt 工程的方法

使用 CMake 构建 Qt 工程, 需要设置 CMake 工具链前缀, 增加 Qt 编译选项等等. 一个简单的用于编译 Qt 项目的 CMakeLists.txt 样例如下所示:

```
cmake_minimum_required(VERSION 3.10)
project(ExampleProject)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_INCLUDE_CURRENT_DIR ON)
```

```

set(CMAKE_PREFIX_PATH "$HOME/Qt/5.15.0/clang_64")

# setup qt
find_package(Qt5 COMPONENTS Widgets Qml Quick Network REQUIRED)
include_directories(${Qt5Widgets_INCLUDE_DIRS} ${QtQml_INCLUDE_DIRS})
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOUIC ON)
set(CMAKE_CXX_FLAGS
    "${CMAKE_CXX_FLAGS} ${Qt5Widgets_EXECUTABLE_COMPILE_FLAGS}")

add_executable(main src/main.cc qml.qrc
target_link_libraries(main
    Qt5::Core Qt5::Widgets Qt5::Qml
    Qt5::Quick Qt5::Network)

```

YaPokemon 的项目构建文件可见根目录下的 CMakeLists.txt.

附录 C 构建 YaPokemon

C.1 构建依赖

YaPokemon 的构建依赖下面这些工具：

1. C++ Compiler (Clang / GCC / MSVC)
2. CMake
3. Qt
4. Ninja (optional)
5. Conan

C.2 构建步骤

首先, 创建构建生成文件夹:

```
mkdir -p build/default
```

随后, 生成构建文件:

```
cd build/default  
ninja install ../..  
cmake -GNinja ../..
```

最后,使用 Ninja 执行构建:

```
ninja
```

若不想使用 Ninja, 也可以直接使用 CMake 默认的 Unix Makefiles 搭配 make 进行构建:

```
cmake ../..  
make
```