
C++ 宠物小精灵 说明文档

姓名：

学号：

目录

一、 宠物小精灵的加入	3
1.1 满足的题目需求	3
1.2 类的设计	4
1.2.1 类的种类	4
1.2.2 类的继承关系	5
1.2.3 类的具体介绍	6
1.2.3.1 Description	6
1.2.3.2 Property	7
1.2.3.3 Level	8
1.2.3.4 Skill 和 SkillManager	9
1.2.3.5 Spirit	12
1.2.3.6 SpiritUtils	16
1.2.3.7 Utils	16
二、 用户注册与平台登录	17
2.1 满足的题目需求	17
2.2 类的设计	17
2.2.1 服务端	17
2.2.2.1 架构图	17
2.2.2.2 Driver 层	18
2.2.2.2.1 Database 类（单例）	19
2.2.2.2.2 SpiritDirver 类（单例）	19
2.2.2.2.3 UserDriver 类（单例）	20
2.2.2.3 Service 层	21
2.2.2.3.1 UserManager 类	21
2.2.2.3.2 Platform 类	21
2.2.2.4 Controller 层	22
2.2.2.4.1 多线程与多路 IO 复用	23
2.2.2.4.2 接口	23
2.2.2.4.2 Server 类	31
2.2.2 客户端	32
2.2.2.1 架构图	32
2.2.2.2 Client 类	32
三、 游戏对战的设计	33
3.1 满足的题目需求	33
3.2 类的介绍	34
3.2.1 Combat 类	34
3.2.2 Controller 层的新接口	35
3.2.3 Driver 层的新表	42
3.2.4 PVP 设计	42

一、宠物小精灵的加入

1.1 满足的题目需求

满足了所有题目需求，经过总结，大致以下几点：

1、宠物小精灵分为力量型、肉盾型、防御型、敏捷型四种类型。

2、每种类型还可细分为种族，例如力量型的火暴猴、防御性的壶壶等。

3、每个精灵包含描述、属性、技能和等级。描述包含种族、描述和昵称，用户可以自行设置宠物昵称。属性包含生命值、防御力、攻击力和速度。技能包含普通技能、特殊技能和终极技能，以及这些技能的效果和描述。等级包含当前等级，距离下一级还需升级的经验和总经验，下面是详细解释。

4、等级：初始等级为 1 级，最高等级为 15 级，升级所需经验根据当前等级确定，如果当前等级为 x ，则升级所需经验为 x^2 。

5、属性：等级升级后，属性也会相应升级，每种类型的升级策略略有差异。力量型的攻击力增加较多，肉盾型的生命值增加较多，以此类推。该升级是在一定范围内是随机的。

6、技能：一个精灵拥有三种技能，普通技能、特殊技能和终极技能，每种技能拥有一系列的 SkillEffect，这些 SkillEffect 由一个初始数值，并且该数值会根据等级的改变而改变，且根据等级唯一确定。若当前等级为 x ，则我们计算出当前技能系数 SkillCoef 为 $1 + (1 / 14) * (x - 1)$ ，当前 SkillEffect 数值由初始数值乘以技能系数确定，可以看出是随着技能线性增长的，不具有随机性。

7、描述：每个精灵都拥有一个描述，描述中包含该精灵的种族、描述和昵称，昵称用户可修改。

1.2 类的设计

1.2.1 类的种类

我们可以先观察文件目录结构

```
├── include
│   ├── Spirit
│   │   ├── Description.hpp    // 描述类头文件
│   │   ├── Level.hpp        // 等级类头文件
│   │   ├── Property.hpp     // 属性类头文件
│   │   ├── Skill.hpp        // 技能类头文件
│   │   └── Spirit.hpp       // 精灵类头文件
│   └── Utils
│       ├── SpiritUtils.hpp   // 精灵工具类 HEAD_ONLY
│       └── Utils.hpp         // 通用工具类头文件
└── src
    ├── Spirit
    │   ├── Description.cpp    // 描述类
    │   ├── Level.cpp         // 等级类
    │   ├── Property.cpp     // 属性类
    │   ├── Skill.cpp         // 技能类
    │   └── Spirit.cpp        // 精灵类
    └── Utils
        └── Utils.cpp         // 通用工具类
```

可以看到，我们共有描述类、等级类、属性类、技能类、精灵类、精灵工具类和通用工具类这几个类文件，实际上的类不止着一些，在精灵类文件中，还有防御性精灵类、生命型精灵类、火暴猴类等等，他们的具体继承关系见下图。

1.2.2 类的继承关系

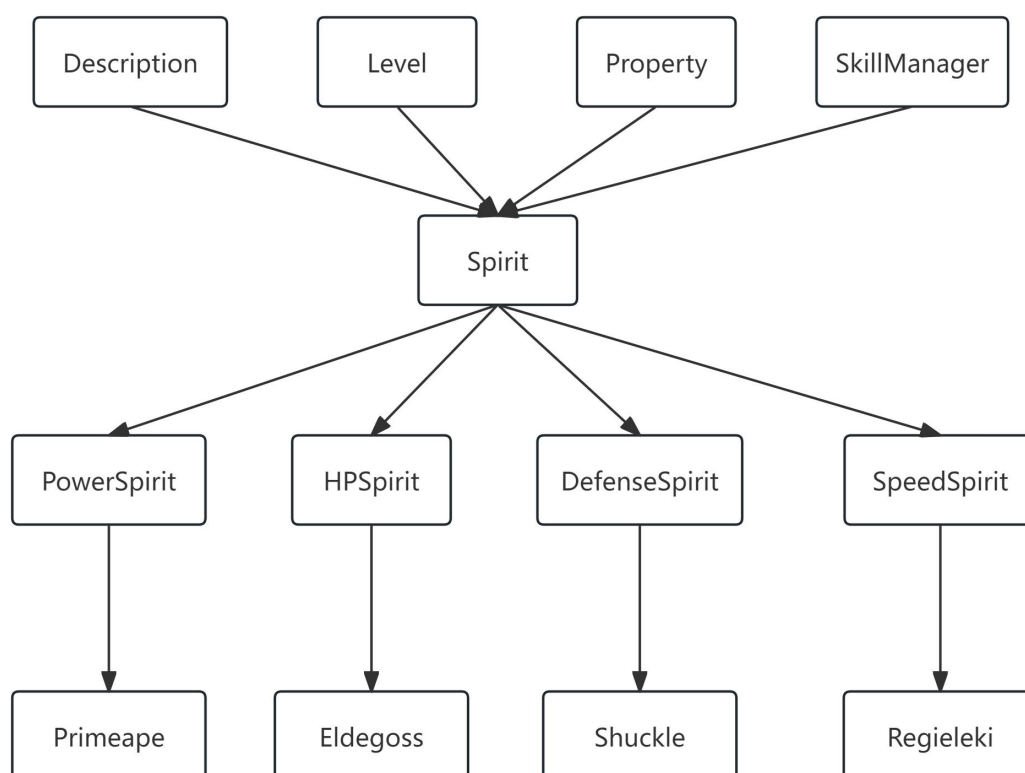
这里主要介绍一下精灵类的继承关系，具体可见下图。

由于每个精灵都必须有描述、等级、属性、技能这四个内容，所以精灵类 `Spirit` 继承了这四个类，即精灵类有四个父类，`Description`、`Level`、`Property`、`SkillManager`。其中前三个类在前面都有所说明，由于一个精灵有三个技能，所以继承一个 `Skill` 类是不够的，所以 `SkillManager` 类中有三个 `Skill` 的 `unique_ptr`，用于管理技能。

同时，精灵需要分为力量型、肉盾型、防御型、敏捷型四种类型，所以这四种类型的精灵类 `PowerSpirit`、`HPSpirit`、`DefenseSpirit`、`SpeedSpirit` 都继承于 `Spirit` 类。

每种类型的精灵又有不同种族，例如火暴猴、壶壶等，这些不同种族的精灵类也继承了他们对应的四种类型的精灵类。例如火暴猴类 `Primeape` 继承了 `PowerSpirit`。

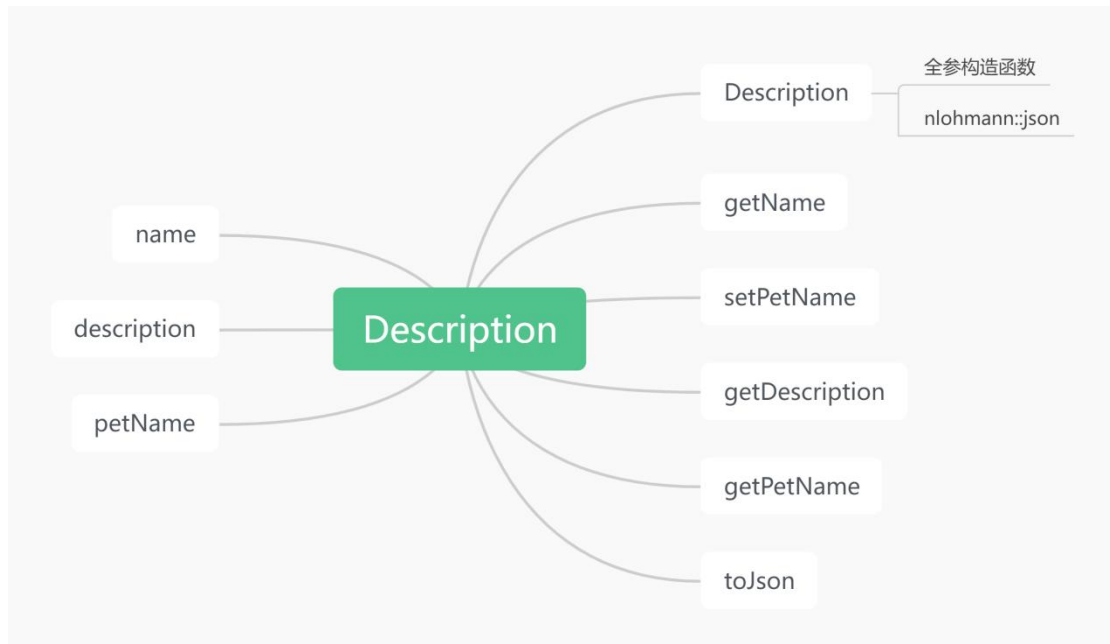
具体的继承关系见下图：



最上层为精灵共有的四种属性，第二层为精灵基类，第三层为四种类型的精灵基类，最下层为具体种族的精灵类，例如力量型的火暴猴，肉盾型的白蓬蓬、防御型的壶壶和速度型的雷吉艾基。

1.2.3 类的具体介绍

1.2.3.1 Description



Description 有三个数据成员，name、description 和 petName。

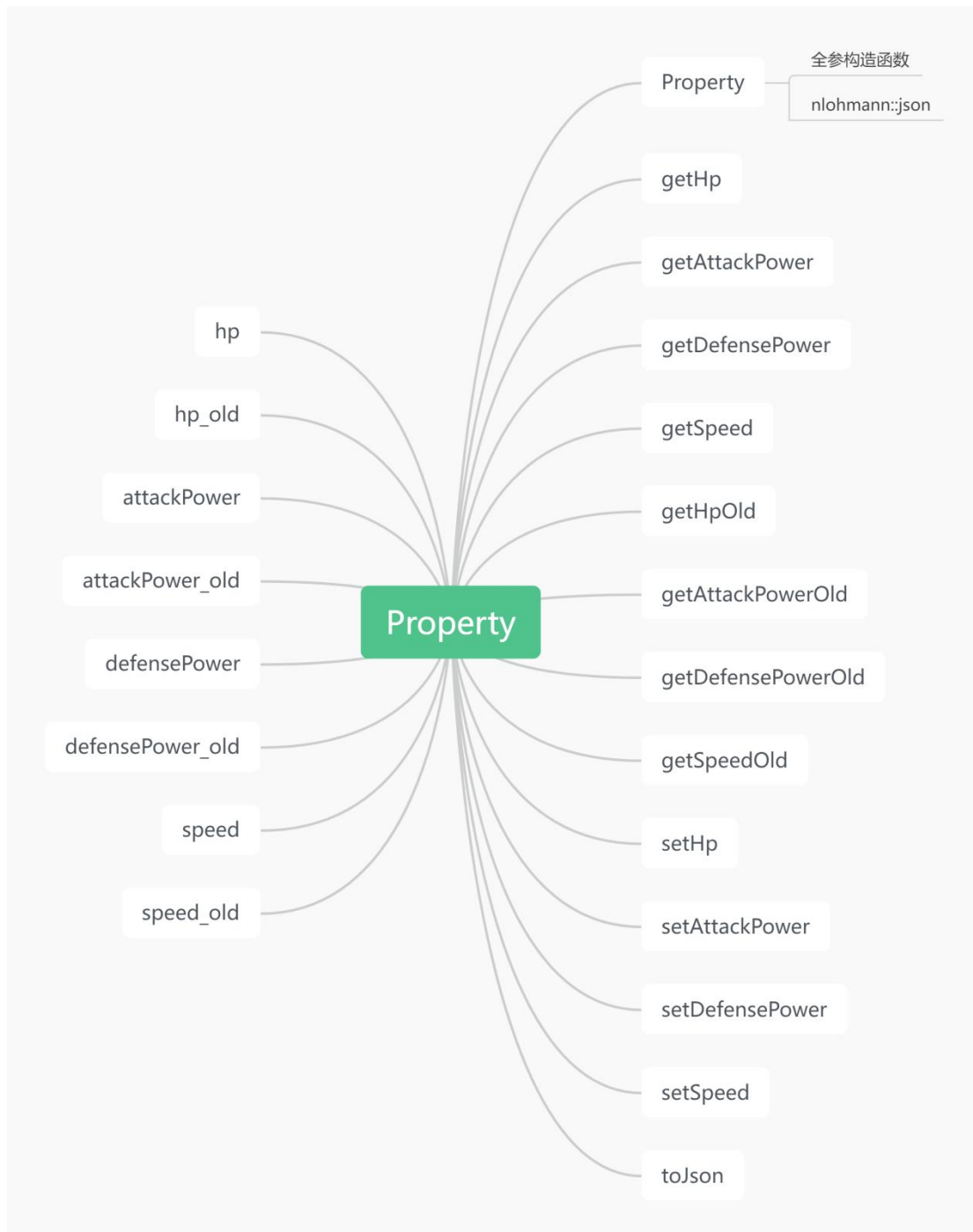
有三个 GET 函数和一个 SET 函数，因为只有 PetName 是可以修改的。

构造函数有两个重载，一个全参构造函数和另外一个通过 json 的反序列化构造函数。

toJson 是序列化函数。

这个类比较简单，就不具体介绍代码了。

1.2.3.2 Property



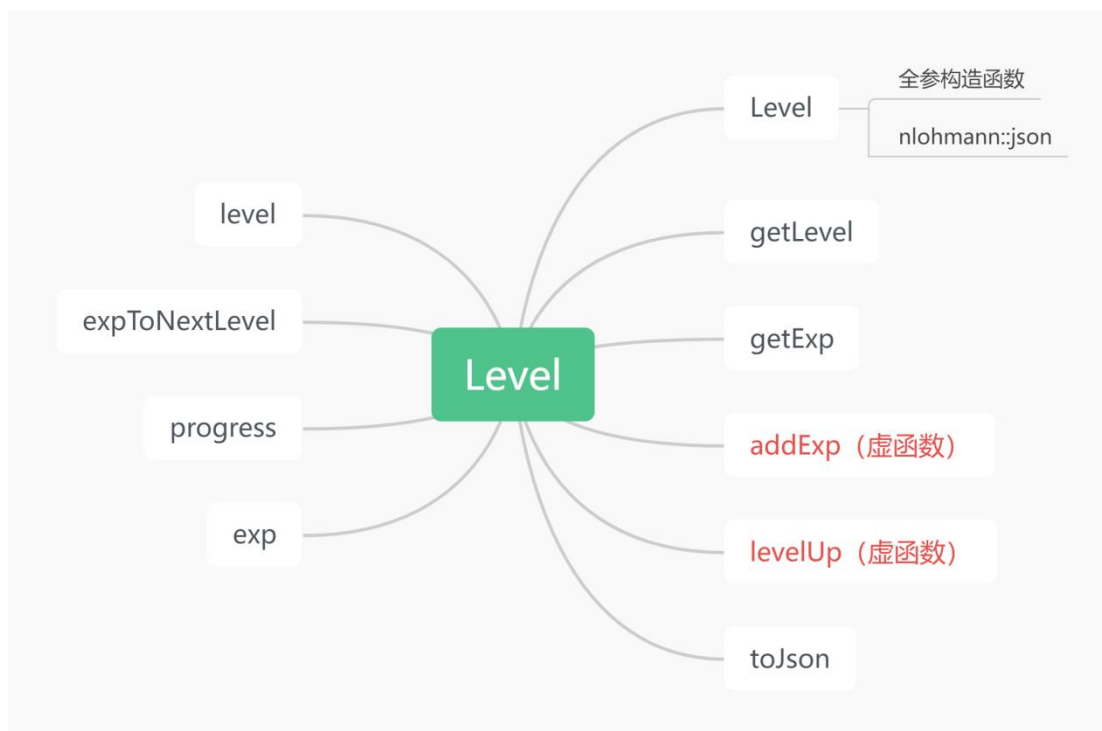
Property 有六个数据成员，分别是 hp、attackPower、defensePower 和 speed 以及他们的旧值，这些旧值在战斗中会用到。

所有的成员都有 GET 和 SET 函数，在 SET 函数中，先把当前值赋给 old，在给当前值赋新值，达到保存旧值的效果。

构造函数有两个重载，一个全参构造函数和另外一个通过 json 的反序列化构造函数。

toJson 是序列化函数。

1.2.3.3 Level



Level 类有四个数据成员，level、expToNextLevel、progress 和 exp。

expToNextLevel 是当前等级升级所需经验，exp 为达到当前等级后获得的经验，progress 为当前升级进程百分比，level 为当前等级。

level 和 exp 设置了 GET 函数。

并没有设置 SET 函数，因为在升级时，我们不仅需要处理等级，还需要处理升级后的属性加成、技能加成，所以在这一层（最顶层）并未实现 addExp 以及 levelUp 函数，这些函数是在下层实现的。addExp 是增加经验的接口，levelUp 处理了升一级后的属性加成以及技能加成问题。

构造函数有两个重载，一个全参构造函数和另外一个通过 json 的反序列化构造函数。

toJson 是序列化函数。

1.2.3.4 Skill 和 SkillManager

在介绍 Skill 类之前，先介绍我实现的 SkillEffect。

SkillEffect 是一个结构体，具体实现如下图所示

```

1  enum Type
2  {
3      HP,
4      ATTACKPOWER,
5      DEFENSEPOWER,
6      SPEED,
7      FIX_VALUE,
8  };
9
10 enum Goal
11 {
12     SELF,
13     ENEMY,
14 };
15
16 struct Target
17 {
18     Goal goal;
19     Type type;
20 };
21
22 struct Source
23 {
24     Goal goal;
25     Type type;
26     int value;
27 };
28
29 struct SkillEffect
30 {
31     std::string description;
32     Target target;
33     Source source;
34     int activationTime;
35     int duration;
36
37     // 重载 < 运算符用于优先队列排序
38     bool operator<(const SkillEffect &effect) const noexcept
39     {
40         return activationTime < effect.activationTime;
41     }
42 };

```

我们可以将 SkillEffect 看作一个 BUFF，任何一个 BUFF 经过抽象，都有以下几个部分：

description: 该 BUFF 的文字介绍

target: 该 BUFF 的作用对象，自己 or 敌人；该 BUFF 的效果，HP or 攻击力 or 防御

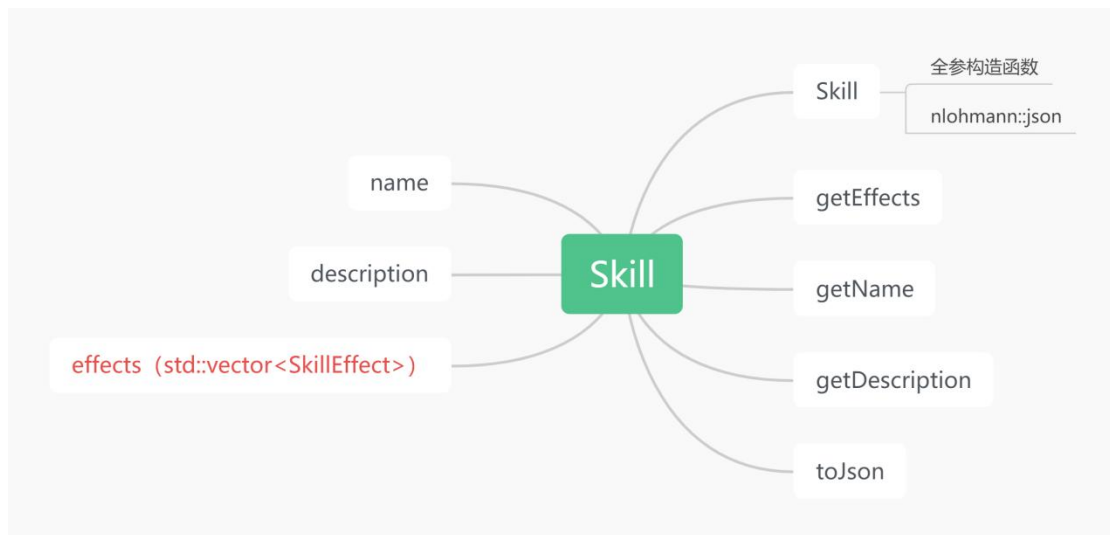
力 or 速度

source: 该 BUFF 数值的来源，默认是百分比模式，那么可能来源于敌人或者自己的攻击力、防御力、生命值、速度的百分比，或是一个固定值。

activationTime: BUFF 在多久后生效

duration: BUFF 的持续时间

一个技能，可以看作是 BUFF 的结合，即 `std::vector<SkillEffect>`，所以我们可以实现一个技能类 `Skill`，将 `SkillEffect` 做一个组合，再加上一些技能的信息，例如描述、名字等等。具体实现如下：



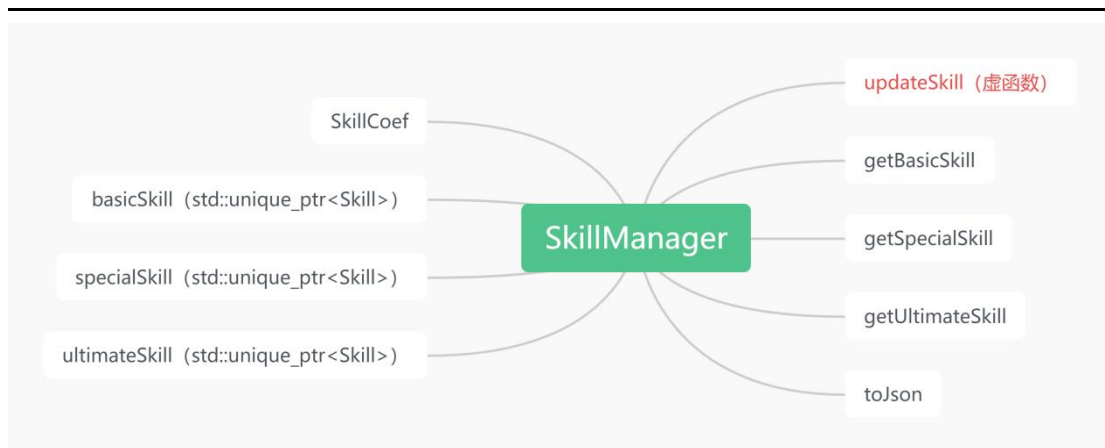
`effects` 即为 BUFF 的组合。

构造函数有两个重载，一个全参构造函数和另外一个通过 `json` 的反序列化构造函数。

`toJson` 是序列化函数。

由于一个精灵有三个技能，所以我们还需要创建一个 `SkillManager` 来管理这三个技能，同时精灵继承这个 `SkillManager` 类，这样精灵类就能够控制三个技能了。同时，由于技能升级后，技能也需要升级，所以 `SkillManager` 类需要提供一个升级技能的虚函数供下层精灵类重写。

`SkillManager` 类设计如下：



SkillCoef 是之前提到的技能系数，用于技能升级。

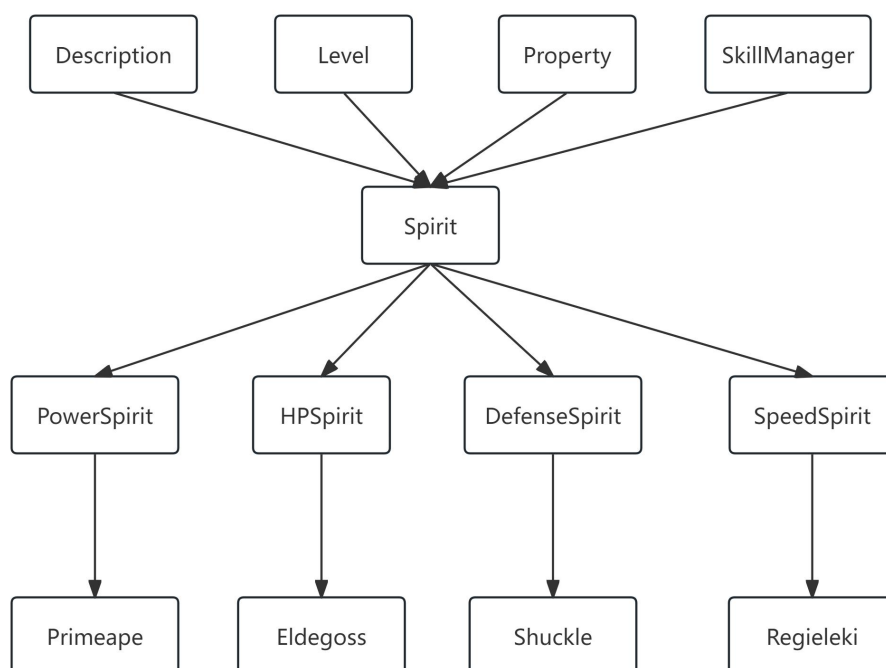
basicSkill、**specialSkill**、**ultimateSkill** 是三个技能，使用 **unique_ptr** 存储

updateSkill 是供下层精灵类重写的技能升级函数

toJson 是序列化函数

1.2.3.5 Spirit

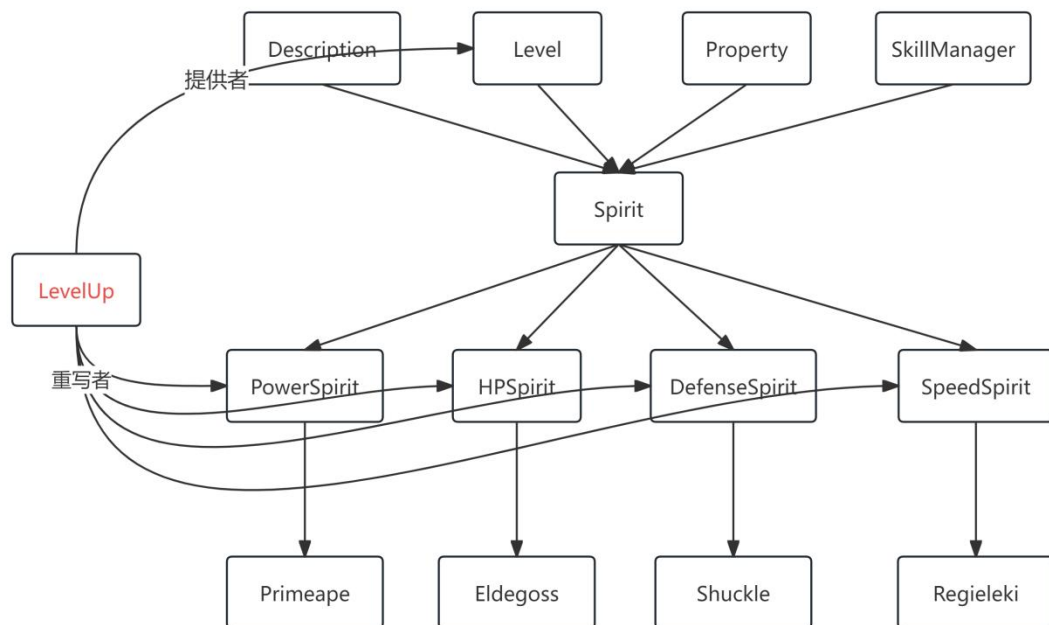
所有与精灵相关的类都在 **Spirit.hpp** 和 **Spirit.cpp** 中，他们的继承关系如下图所示



这些类都没有新增成员函数，只是在构造函数中初始化了父类的成员函数。实际上，父类的成员函数包含了一个精灵的所有属性。

但是，这些类重写了父类的一些虚函数，这些虚函数在上面已经提到，现在进行总结。

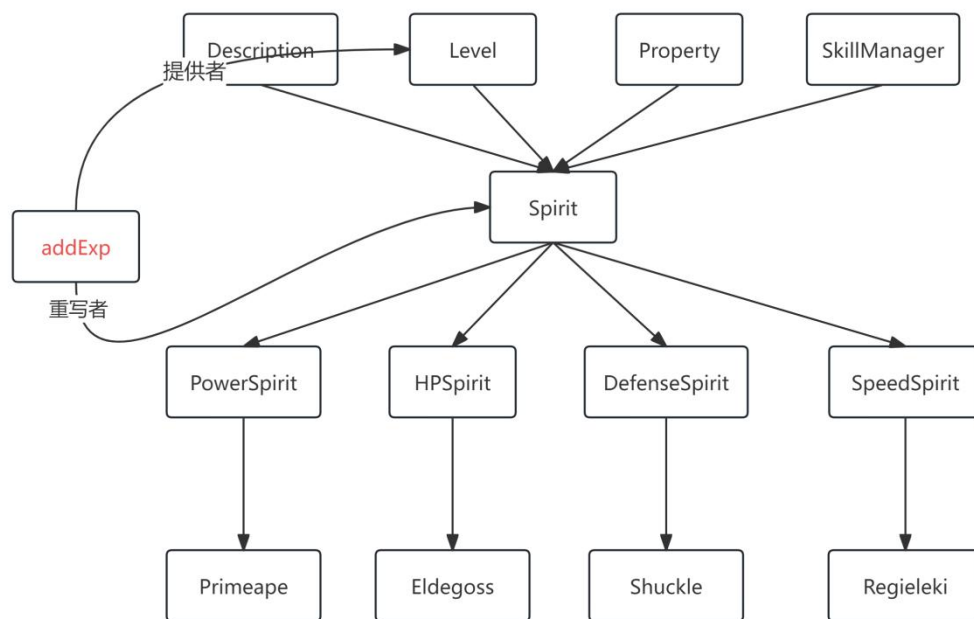
首先是 LevelUp



LevelUp 由第一层 Level 提供，由第三层具体类型的精灵重写，这是由于，不同类型的精灵在升级时的点数加成是不同的，此处采取的策略是：

对于主属性，我们随机加 5 - 8 点，对于其他属性，随机加 3 - 6 点。

接下来是 addExp

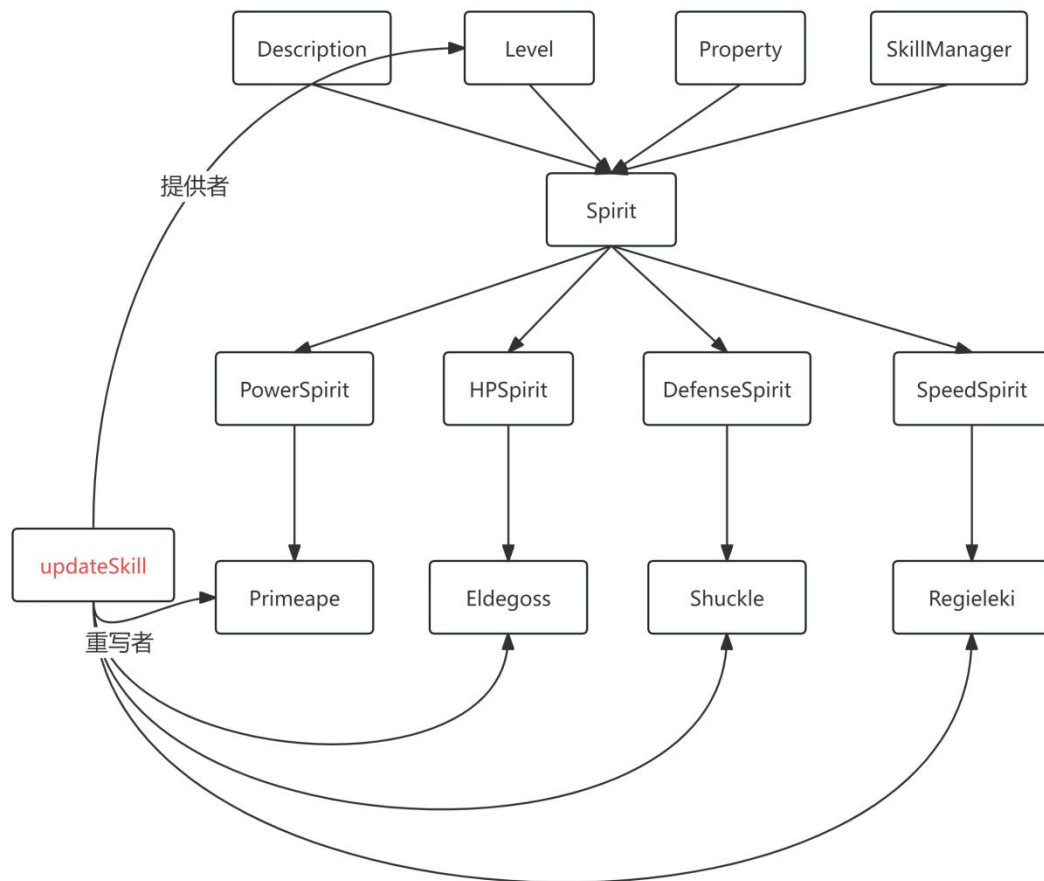


addExp 由第一层 Level 提供，由第二层重写，这是由于增加经验后，我们可以调用子类重写的 levelUp 去升级对应的点数，而升级的逻辑，即多少经验增加一级等等，所有精灵都是一样的，所以我们选择在精灵基类去重写这个函数。

具体策略是：

如果当前经验超过 15 级，那么只增加 exp，不升级 level，否则加上经验后，不断循环当前经验，每升一级就调用一次 LevelUp 来升级属性，直到当前经验不够再升一级，最后计算 progress。

接下来是 updateSkill，updateSkill 不仅是升级技能，也是技能的初始化，这是因为技能不是精灵类的父类，而是通过 SkillManager 类中的 unique_ptr 存储的，所以需要手动初始化，并且技能的升级是与等级一一确定的，所以在精灵构造时就可以确定技能的属性，见下图：



由上述原因，技能类是通过最底层的具体种族的精灵重写的，这是因为每个精灵的技能都不同，所以我们需要在最下层重写。

1.2.3.6 SpiritUtils

当我们把一个 JSON 反序列化为一个对象的时候，我们首先要确认这个对象属于哪个类型，这边我们通过判断 JSON 的种类字符串来实例化 JSON，这就是 SpiritUtils 类的作用。

同时，在后期我们还需要随机生成一个精灵类的功能，这也说 SpiritUtils 类的作用之一。

SpiritUtils 类实现了两个静态函数：

`static std::unique_ptr<Spirit> getSpirit(nlohmann::json j)`，将 JSON 实例化为 Spirit 对象，其中 `unique_ptr` 是根据 `make_unique` 一个具体的种族来实现的，也就是多态。

`static nlohmann::json getRandomSpirits(int maxLevel)`：生成一个 1-maxLevel 之间的随机精灵，共后续任务使用。

1.2.3.7 Utils

Utils 类定义了一些通用的静态工具函数：

`static std::string getSHA256(const std::string &text)`：获取 text 的 SHA256 值，用于密码的 SHA256 校验。

`static int get_random_int(int min, int max)`：获取 min - max 之间的随机数。

二、用户注册与平台登录

2.1 满足的题目需求

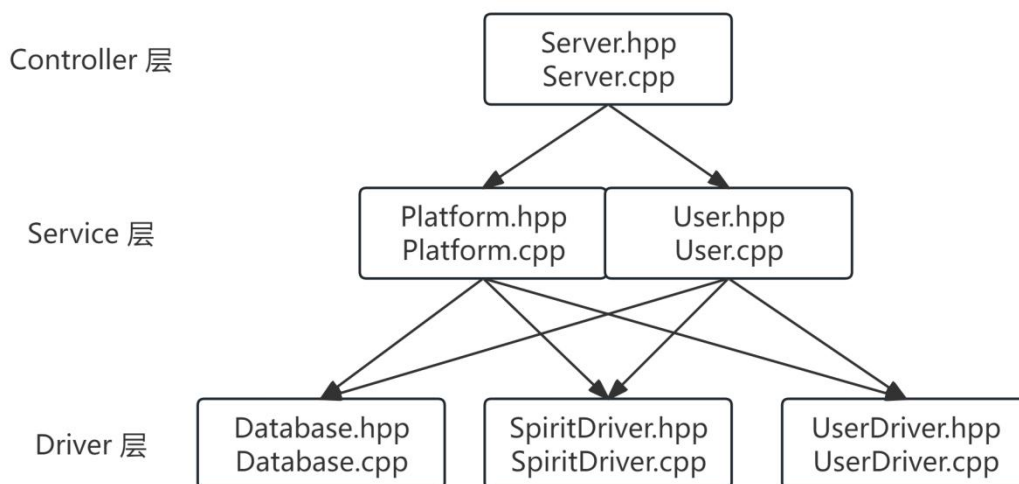
- 1、实现用户的注册、登录与登出。用户名全局唯一。
- 2、不允许同一个用户重复登录。
- 3、客户端与服务端利用 libhv 库，通过 HTTP 通讯。
- 4、用户注册后会随机分配给用户 3 个 1 级精灵。
- 5、可以查看在线用户以及他们拥有的精灵信息。
- 6、用户可以查看自己的精灵详细信息并修改昵称。
- 7、服务端使用 SQLite 持久化用户和精灵信息

2.2 类的设计

2.2.1 服务端

2.2.2.1 架构图

服务端使用了 Controller - Service - Driver 三层架构，层次清晰，下面是架构图，展现了每个架构对应的文件。



下面自底向上介绍整个架构，同时介绍实现细节。

2.2.2.2 Driver 层

本项目使用 SQLite 持久化数据，为了实现第二个需求，我们需要两个表：

表 `users`，用于保存用户信息，建表语句如下：

// 建表语句

```
// CREATE TABLE users(
```

```
//    id integer primary key autoincrement,    序号，自增，主键
```

```
//    username text not null,                用户名
```

```
//    password_hash text not null)           密码的 SHA256 值
```

表 `spirits`，用于保存精灵，建表语句如下：

```
// CREATE TABLE spirits(
```

```
//    id integer primary key,                序号，自增，主键
```

```
//    user_id integer not null,              该精灵主人的用户 ID
```

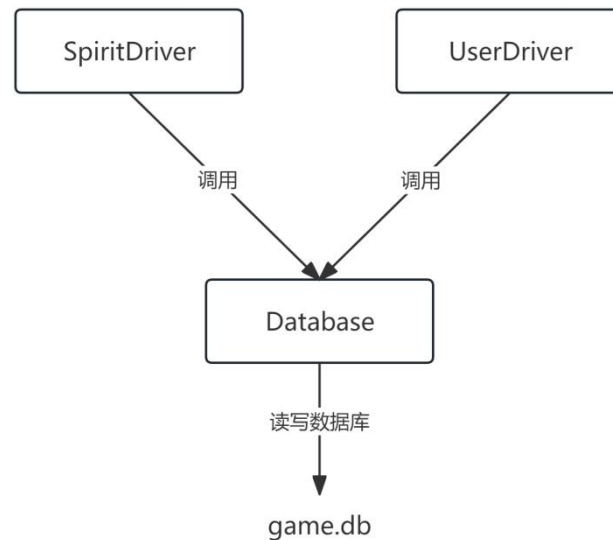
```
//    spirit_json text not null)             精灵的序列化 JSON 字符串
```

由于 SQLite 的官方 C/C++ Interface 都是 C 风格的，直接操作数据库，且包含大量裸指

针，所以我写了一个 `Database` 类，该类是一个单例，用于直接和底层数据库交互。

在 `Database` 类之上，对于用户信息，实现了单例 `SpiritDirver` 类和 `UserDriver` 类，并在这些类中定义了对应于数据表项的 `SpiritInterface` 和 `UserInterface`。

三个类的关系如下图所示：



2.2.2.2.1 Database 类（单例）

`Database` 层直接调用 `sqlite3` 官方的 API 并且将 CRUD 操作封装，供上层使用。

`SQLite3` 官方 API 会直接返回一个二维指针，这里将其转化为 C++ 风格的 `std::vector<std::string>` 并返回。

2.2.2.2.2 SpiritDirver 类（单例）

`SpiritDirver` 类中定义了精灵的数据格式如下

```
1 struct SpiritInterface
2 {
3     int id;
4     int user_id;
5     std::string spirit_json;
6 };
```

同时还包含了表的一些信息，例如列数、字段名，它将 Database 返回的 `std::vector<std::string>` 进行序列化，并且直接返回 `std::vector<SpiritInterface>`。

为上层提供精灵的增删改查

2.2.2.2.3 UserDriver 类（单例）

UserDriver 类中定义了用户数据格式如下

```
1 struct UserInterface
2 {
3     int id;
4     std::string username;
5     std::string password_hash;
6 };
```

同时还包含了表的一些信息，例如列数、字段名，它将 Database 返回的 `std::vector<std::string>` 进行序列化，并且直接返回 `std::vector<UserInterface>`。

为上层提供用户的增删改查

2.2.2.3 Service 层

Service 层包含两个类 Platform 和 UserManager，Platform 类继承 UserManager 类。

2.2.2.3.1 UserManager 类

UserManager 类是一个管理用户登录和注册的类。它提供了用户登录、注册、获取用户信息、注销以及检查登录状态的功能，他调用 UserDriver 来实现功能，并且提供登录、注册、注销的高级接口。此外，它还提供了一个虚函数 registerUser，Platform 会重写这个函数，已达到注册后添加三个精灵的效果。

2.2.2.3.2 Platform 类

Platform 类是一个继承自 UserManager 的类，它扩展了 UserManager 的功能，专注于管理与精灵相关的信息和操作。

在开发过程中，遇到了数据一致性的问题，最后采用的策略是，增删改精灵时立即修改数据库。

Platform 用于为整个用户连接提供接口，它继承 UserManager，每有一个用户连接，就会为这个连接创建一个 Platform 实例。

init(): 立即从数据库中获取该用户所有的精灵信息，并序列化保存到成员变量中。

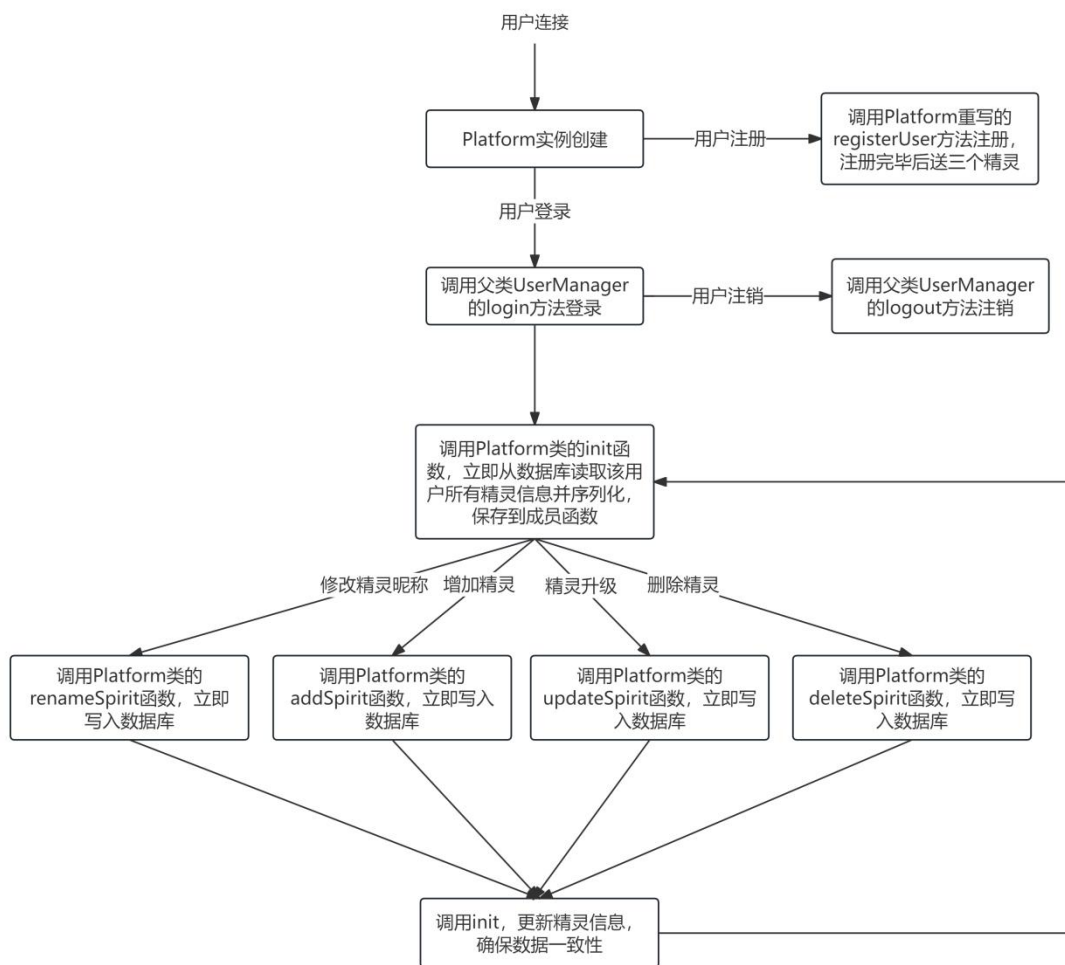
addSpirit(): 增加精灵，立即写入数据库

renameSpirit(): 修改精灵昵称，立即写入数据库

deleteSpirit(): 删除精灵，立即写入数据库

updateSpirit(): 升级精灵，立即写入数据库

以下是工作流程：



2.2.2.4 Controller 层

Controller 层只有一个 Server 类，

1、在 Server 类最重要的数据成员是 `std::map<int, std::unique_ptr<Platform>>> platforms`，`int` 即 `session_id`，每一个客户端在连接服务端之后，服务端都会为其创建一个 Platform 类的实例，自此之后，客户端的任何操作都会根据其提交的 `session_id` 所确定的 Platform 类的实例来确定。

2、此外，为了防止 `platforms` 的读写冲突，我们定义了一个锁 `std::mutex mtx`，每次读写 `platforms` 前，都先 `lock`，读取完毕后立即 `unlock`。

3、在 Platform 类的构造函数中，我们利用 `libhv` 库定义了一系列 HTTP 接口，供客户端使用，第二个任务需要用到的接口内容见 2.2.2.4.1。

4、构造函数需要提供一个参数 `port`，即 HTTP 服务的端口，在运行 `run` 函数之后，服

务端会启动并进入主循环。

2.2.2.4.1 多线程与多路 IO 复用

我们利用了 libhv 库提供的接口实现了多线程与多路 IO 复用，首先在 Server 类的 run 函数中，我们通过 `server.worker_threads = 4` 设置了工作线程数为 4。

同时 libhv 库底层通过多路 IO 复用来提高效率。

多路 IO 复用（IO Multiplexing）是一种处理多重 IO 操作的技术，通过单个线程可以同时监视多个文件描述符，以提高系统的并发处理能力。多路 IO 复用技术在网络编程中广泛应用，可以在同一时刻处理多个网络连接而不需要为每个连接创建一个独立的线程。

多路 IO 复用允许程序通过一个系统调用（如 `select`、`poll` 或 `epoll`）同时等待多个文件描述符变为就绪（即有数据可读、可写或有错误发生）。一旦这些文件描述符中的任何一个变为就绪，系统调用返回，程序可以对这些就绪的文件描述符进行相应的 IO 操作。

优势： 相比于传统的同步通信和异步通信，多路 IO 复用具有以下优势：

高效性：可以同时监视多个文件描述符，提高了系统的并发处理能力，避免了为每个连接创建一个线程的资源消耗。

节省资源：减少了线程的数量，从而降低了线程上下文切换的开销，节省了系统资源。

灵活性：支持多种 IO 事件（如可读、可写、异常等），适用于各种网络通信场景。

在高并发服务器设计中，多路 IO 复用是实现高效网络通信的重要技术之一。通过合理使用多路 IO 复用，可以显著提高系统的性能和吞吐量。

2.2.2.4.2 接口

GET 获取会话

GET /create_session

创建一个会话

返回示例

200 Response

```
{
  "session_id": 0
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功

POST 注册

POST /register

Body 请求参数

```
{
  "username": "conc",
  "password": "123456",
  "session_id": 0
}
```

返回示例

200 Response

```
{
  "msg": "string"
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功
404	Not Found	记录不存在

GET 获取精灵信息

GET /spirits/get

Body 请求参数

```
{
```

```
"session_id": 0
}
```

返回示例

成功

```
[
  {
    "id": 4,
    "spirit_json": {
      "description": {
        "description": "火暴猴球状的身体长满了淡黄色的毛。它四肢的被毛消失且戴上了黑色的圆环，手变成了拳套的样子，脚也像人类的脚掌，只是只有两个脚趾。它的尾巴消失了，眼珠变成一条线，额头上一直有发怒隆起的十字青筋。",
        "name": "火暴猴",
        "petName": "conc"
      },
      "level": {
        "exp": 1000,
        "expToNextLevel": 196,
        "level": 14,
        "progress": 92
      },
      "property": {
        "attackPower": 182,
        "defensePower": 122,
        "hp": 124,
        "speed": 150
      },
      "skills": {
        "basicSkill": {
          "description": "火暴猴用拳头猛击对手，造成攻击力 19 % 的伤害",
```

```
"effects": [  
  {  
    "activationTime": 0,  
    "description": "火爆猴普通攻击",  
    "duration": 0,  
    "source": {  
      "goal": "SELF",  
      "type": "ATTACKPOWER",  
      "value": -19  
    },  
    "target": {  
      "goal": "ENEMY",  
      "type": "HP"  
    }  
  }  
],  
"name": "基础攻击"  
},  
"specialSkill": {  
  "description": "火暴猴恢复自身防御力 38 % 的生命值",  
  "effects": [  
    {  
      "activationTime": 0,  
      "description": "恢复",  
      "duration": 0,  
      "source": {  
        "goal": "SELF",  
        "type": "DEFENSEPOWER",  
        "value": 38  
      },  
    }  
  ],  
}
```

```
"target": {  
    "goal": "SELF",  
    "type": "HP"  
}  
}  
],  
"name": "恢复"  
},  
"ultimateSkill": {  
    "description": "火暴猴用拳头猛击对手，造成攻击力 57 % 的伤害，并恢复自身防御  
力 19 % 的生命值",  
    "effects": [  
        {  
            "activationTime": 0,  
            "description": "地球上投",  
            "duration": 0,  
            "source": {  
                "goal": "SELF",  
                "type": "ATTACKPOWER",  
                "value": -57  
            },  
            "target": {  
                "goal": "ENEMY",  
                "type": "HP"  
            }  
        },  
        {  
            "activationTime": 0,  
            "description": "地球上投",  
            "duration": 0,
```

```
      "source": {
        "goal": "SELF",
        "type": "DEFENSEPOWER",
        "value": 19
      },
      "target": {
        "goal": "SELF",
        "type": "HP"
      }
    },
    "name": "地球上投"
  }
}
]
```

记录不存在

```
{
  "msg": "Invalid session id"
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功
404	Not Found	记录不存在

GET 获取随机精灵

GET /random_spirits

Body 请求参数

```
{  
  "session_id": 0  
}
```

返回示例

略

返回结果

状态码	状态码含义	说明
200	OK	成功

POST 更改精灵名字

POST /spirits/rename

Body 请求参数

```
{  
  "session_id": 0,  
  "spirit_id": 4,  
  "new_name": "cobe"  
}
```

返回示例

略

返回结果

状态码	状态码含义	说明
200	OK	成功

GET 登出

GET /logout

Body 请求参数

```
{
```

```
"session_id": 0
}
```

返回示例

略

返回结果

状态码	状态码含义	说明
200	OK	成功

POST 客户端退出

POST /exit

Body 请求参数

```
{
  "session_id": 0
}
```

返回示例

略

返回结果

状态码	状态码含义	说明
200	OK	成功

GET 获取所有用户

GET /all_users

Body 请求参数

```
{
  "session_id": 0
}
```

返回示例

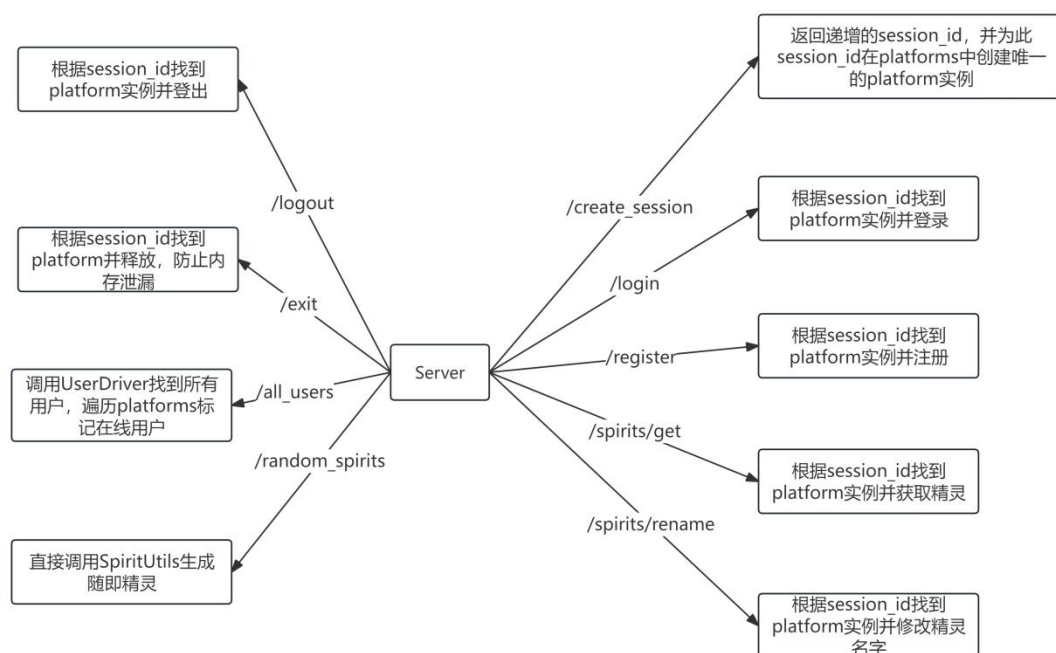
略

返回结果

状态码	状态码含义	说明
200	OK	成功

2.2.2.4.2 Server 类

Server 类的具体工作流程如下：



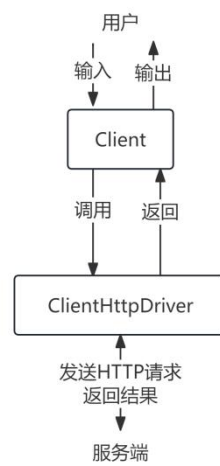
2.2.2 客户端

2.2.2.1 架构图

客户端实现了两个类

Client: 客户端逻辑类，用于读取用户输入、输出 UI 等。

ClientHttpDriver: HTTP 驱动类，将 HTTP 接口封装成函数。



2.2.2.2 Client 类

Client 类负责与用户的交互，以自动机的模式工作，拥有十种状态如下：

CREATE_SESSION,

START,

MAIN,

LOGIN,

REGISTER,

LOGOUT,

BAG,

MY_SPIRITS,

ONLINE_USERS,

EXIT,

状态转移图见下：



三、游戏对战的设计

3.1 满足的题目需求

满足了题目的所有需求，并进行了扩展：

- 1、PVE 部分，完成了决斗赛和升级赛两种，每打赢一场升级赛，自动增加 50 点经验。
决斗赛失败需要送出一只精灵，决斗赛胜利可以获得对手精灵。
- 2、若用户送出了最后一个精灵，系统会送给用户一个精灵。
- 3、对战部分，加入概率闪避攻击和暴击伤害机制，每个回合出招由速度随机决定，速度越大，出招的概率越大。
- 4、某用户可以查看自己的胜率。

5、实现了勋章功能。

6、实现了 PVP，可以实现用户在线对战。

3.2 类的介绍

3.2.1 Combat 类

Combat 类为战斗类，战斗逻辑在该类中实现。

该类的构造函数需要传入两个精灵。

对于每个精灵，都有一个 `std::vector<std::vector<SkillEffect>>` `spiritSkillEffects`，该数据成员存储了该精灵三个技能的 BUFF 序列。

下面主要讲解对战逻辑：

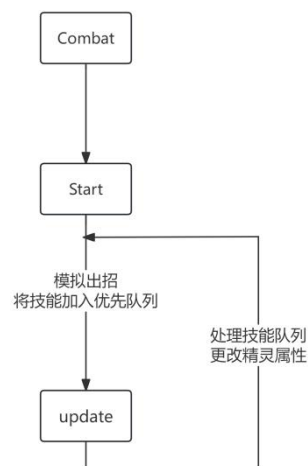
对于每个精灵，都有 `std::priority_queue<SkillEffect>` `skillQueue` 这个优先队列，该优先队列存储精灵出招的所有 BUFF，且根据激活时间优先进行排队。

在每个回合，精灵出招结束后，`update()`函数会遍历每个精灵的优先队列，从中取出激活时间为 0 的 BUFF，并立即生效，同时剩余效果的激活时间会减一。

执行 `start()`函数后，程序会自动进行自动模拟对战，每进行一轮就调用一次 `update()`，直到某个精灵生命值小于 0。

`start()`函数执行完毕后,说明对战结束,此时对战结果已经存储到了 `nlohmann::json result` 这个 json 对象中,可以通过 `getResult()`函数获得这个结果,交由前端显示。在这个 json 对象中,有三个字段,首先是 `fighter`,其中有两个对战精灵的昵称, `round` 是一个数组,数组中的每一项即代表一轮,包含了每个精灵的属性以及对战语句, `winner` 是一个数字,指示了胜利的精灵。

下面是执行过程图:



`getValue()`函数能够获得某个精灵的某个当前属性

`getOldValue()`函数能够获得某个精灵的某个属性的旧值

`setValue()`函数能够设置某个精灵的某个属性

`handleEffect()`用来在 `update` 函数中处理单个 `Effect`, 将其生效

`toFixValue()`可以将数值来源为百分比的 `SkillEffect` 转化为固定数值, 防止出现问题

3.2.2 Controller 层的新接口

为了实现功能, Controller 层的 `Server` 类中增加了新的接口, 如下所示:

GET 战斗

GET /combat

Body 请求参数

两个精灵 JSON

返回示例

成功

```
{
  "fighter": [
    "conc",
    "火暴猴 252"
  ],
  "round": [
    {
      "attack": "conc 此轮攻击! , 发动技能 恢复",
      "property": [
        {
          "attackPower": 182,
          "defensePower": 122,
          "hp": 170,
          "speed": 150
        },
        {
          "attackPower": 105,
          "defensePower": 60,
          "hp": 65,
          "speed": 95
        }
      ],
      "result": [
        "conc 受到了 conc 的 恢复 技能的影响, HP 由 124 变为 170"
      ]
    },
    {
      "attack": "火暴猴 252 此轮攻击! , 发动技能 基础攻击",
      "property": [
        {
          "attackPower": 182,
          "defensePower": 122,
          "hp": 160,
          "speed": 150
        },
        {
          "attackPower": 105,
          "defensePower": 60,
          "hp": 65,
          "speed": 95
        }
      ],
      "result": [
```

"conc 受到了 火暴猴 252 的 火爆猴普通攻击 技能的影响, HP 由 170 变为 160"

```
    ]
  },
  {
    "attack": "conc 此轮攻击! , 发动技能 基础攻击",
    "property": [
      {
        "attackPower": 182,
        "defensePower": 122,
        "hp": 160,
        "speed": 150
      },
      {
        "attackPower": 105,
        "defensePower": 60,
        "hp": 31,
        "speed": 95
      }
    ],
    "result": [
      "火暴猴 252 受到了 conc 的 火爆猴普通攻击 技能的影响,HP 由 65 变为
31"
```

```
    ]
  },
  {
    "attack": "火暴猴 252 此轮攻击! , 发动技能 地球上投",
    "property": [
      {
        "attackPower": 182,
        "defensePower": 122,
        "hp": 113,
        "speed": 150
      },
      {
        "attackPower": 105,
        "defensePower": 60,
        "hp": 31,
        "speed": 95
      }
    ],
    "result": [
      "conc 受到了 火暴猴 252 的 地球上投 技能的影响, HP 由 160 变为 113,
暴击! ",
```

```

        "火暴猴 252 的 地球上投 技能未命中! "
    ]
},
{
    "attack": "火暴猴 252 此轮攻击! , 发动技能 恢复",
    "property": [
        {
            "attackPower": 182,
            "defensePower": 122,
            "hp": 113,
            "speed": 150
        },
        {
            "attackPower": 105,
            "defensePower": 60,
            "hp": 31,
            "speed": 95
        }
    ],
    "result": [
        "火暴猴 252 的 恢复 技能未命中! "
    ]
},
{
    "attack": "conc 此轮攻击! , 发动技能 基础攻击",
    "property": [
        {
            "attackPower": 182,
            "defensePower": 122,
            "hp": 113,
            "speed": 150
        },
        {
            "attackPower": 105,
            "defensePower": 60,
            "hp": -20,
            "speed": 95
        }
    ],
    "result": [
        "火暴猴 252 受到了 conc 的 火爆猴普通攻击 技能的影响,HP 由 31 变为
-20, 暴击! "
    ]
}

```

```
],  
  "winner": 0  
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功

DELETE 删除精灵

DELETE /spirits/delete

Body 请求参数

```
{  
  "session_id": 0,  
  "spirit_id": 7  
}
```

请求参数

名称	位置	类型
body	body	object
» session_id	body	integer
» spirit_id	body	integer

返回结果

状态码	状态码含义	说明
200	OK	成功

POST 添加精灵

POST /spirits/add

Body 请求参数

```
{  
  "session_id": 0,
```

```
"random": true
```

```
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功

返回数据结构

POST 升级精灵

POST /spirits/level_up

Body 请求参数

```
{  
  "session_id": 0,  
  "spirit_id": 4,  
  "exp": 1000  
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功

GET 获取胜率

GET /get_win

Body 请求参数

```
{  
  "session_id": 0  
}
```

返回结果

状态码	状态码含义	说明
-----	-------	----

状态码	状态码含义	说明
200	OK	成功

POST 更新胜率

POST /update_win

Body 请求参数

```
{
  "session_id": 0,
  "win": true
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功

POST 进入匹配

POST /join_match

Body 请求参数

```
{
  "session_id": 0,
  "spirit_id": 17
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功

GET 获取匹配结果

GET /get_match

Body 请求参数

```
{  
    "session_id": 0  
}
```

返回结果

状态码	状态码含义	说明
200	OK	成功

3.2.3 Driver 层的新表

为了保存胜率，在数据库中增加了新的表格 **Win**，用于存储胜率，该表格共有三个字段，**user_id**，**win**，**total**，分别表示用户 ID、胜的场数和总场数。

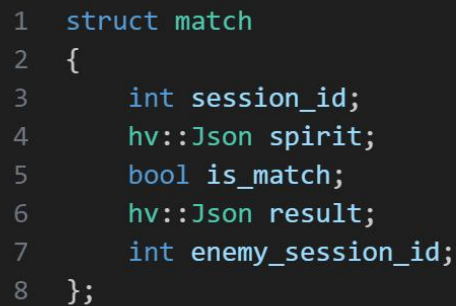
同时，增加了 **WinDriver** 单例类来实现胜率的操作。

3.2.4 PVP 设计

为了实现 PVP，在 **Server** 类中增加了两个接口，**join_match** 和 **get_match**。

当用户选择 PVP 之后，会方法 **join_match** 这个 API，通过这个 API，用户可以加入匹配队列。

在 **Server** 中，新增了一个匹配接口 `std::queue<match> matching_queue`，**match** 结构体的定义如下：



```
1 struct match
2 {
3     int session_id;
4     hv::Json spirit;
5     bool is_match;
6     hv::Json result;
7     int enemy_session_id;
8 };
```

客户端加入匹配后，服务端会将其加入匹配队列中。

客户端加入匹配队列后，每 1s 会轮询 `get_match` 来获取匹配结果，在 `get_match` 中，服务端会处理匹配队列，如果发现超过两个用户在队列中后，立即将前两个用户的 `match` 结构体的 `is_match` 值赋为 `true`，新建 `combat` 类进行模拟对战，并将对战结果存入 `match` 结构体，以实现匹配的结果。