2024宠物小精灵

安装

本项目使用vcpkg作为包管理器，并采用CMake和Ninja进行编译。为了帮助用户快速设置和运行项目，以下是详细的安装和编译步骤。

下载项目

首先，克隆项目到本地：

```
git clone https://github.com/greed106/SpriteServer
cd SpriteServer
```

安装vcpkg

如果尚未安装vcpkg，请按照以下步骤安装：

1. 克隆vcpkg仓库：

```
git clone https://github.com/microsoft/vcpkg.git
cd vcpkg
```

2. 运行vcpkg安装脚本：
   - 对于Windows：

```
.\bootstrap-vcpkg.bat
```

   - 对于Linux和macOS：

```
./bootstrap-vcpkg.sh
```

3. 将vcpkg添加到环境变量中：
   - 对于Windows：

```
set PATH=%PATH%;<vcpkg安装目录>
```

   - 对于Linux和macOS：

```
export PATH=$PATH:<vcpkg安装目录>
```

配置vcpkg清单模式

在项目的根目录中，已经包含了`vcpkg.json`文件，用于指定项目依赖的开源库。内容如下：

```
{
  "dependencies": [
    "libhv",
    "libmysql",
    "spdlog",
    {
      "name": "poco",
      "features": ["mysql"]
    },
    "nlohmann-json",
    "fmt"
  ]
}
```

这些库的作用如下：

- **libhv**：一个高性能的网络库，用于实现HTTP和WebSocket通信。
- **libmysql**：MySQL数据库客户端库，用于与MySQL数据库交互。
- **spdlog**：一个快速的日志库，用于记录日志信息。
- **poco**：一个C++类库，用于简化网络编程、数据库访问和其他系统级编程任务。启用了`mysql`特性。
- **nlohmann-json**：一个现代的C++ JSON库，用于JSON序列化和反序列化。
- **fmt**：一个格式化库，用于替代`printf`的C++库。

特别感谢这些开源库的作者和贡献者们，是他们的工作使得这个项目的开发变得更加容易和高效。

配置CMake

项目中包含一个CMakeUserPresets.json模板文件。用户需要复制一份这个模板并进行相应修改：

1. 复制模板文件并重命名：

```
cp CMakeUserPresets.json.template CMakeUserPresets.json
```

2. 编辑`CMakeUserPresets.json`文件，修改其中的路径和配置：

```
{
    "version": 2,
    "configurePresets": [
      {
        "name": "vcpkg",
        "inherits": "default",
        "environment": {
          "VCPKG_ROOT": "/path/to/vcpkg"
        },
        "cacheVariables": {
          "CMAKE_BUILD_TYPE": "Release"
        }
      }
    ]
  }
```

将`/path/to/vcpkg`修改为实际vcpkg安装目录，`CMAKE_BUILD_TYPE`根据需要修改为`Release`或`Debug`。

编译项目

配置完成后，可以使用以下命令编译项目：

1. 创建并进入构建目录：

```
mkdir build
cd build
```

2. 运行CMake配置命令：

```
cmake --preset=vcpkg
```

3. 编译项目：

```
cmake --build build
```

至此，项目的编译已经完成，可以运行生成的可执行文件进行测试和使用。

通过以上步骤，您已经成功地安装并编译了项目。特别感谢使用到的开源库的作者和贡献者们，他们的工作大大简化了项目的开发过程。希望这些详细的介绍对您的项目有所帮助。

宠物小精灵的加入

题目需求

精灵属性

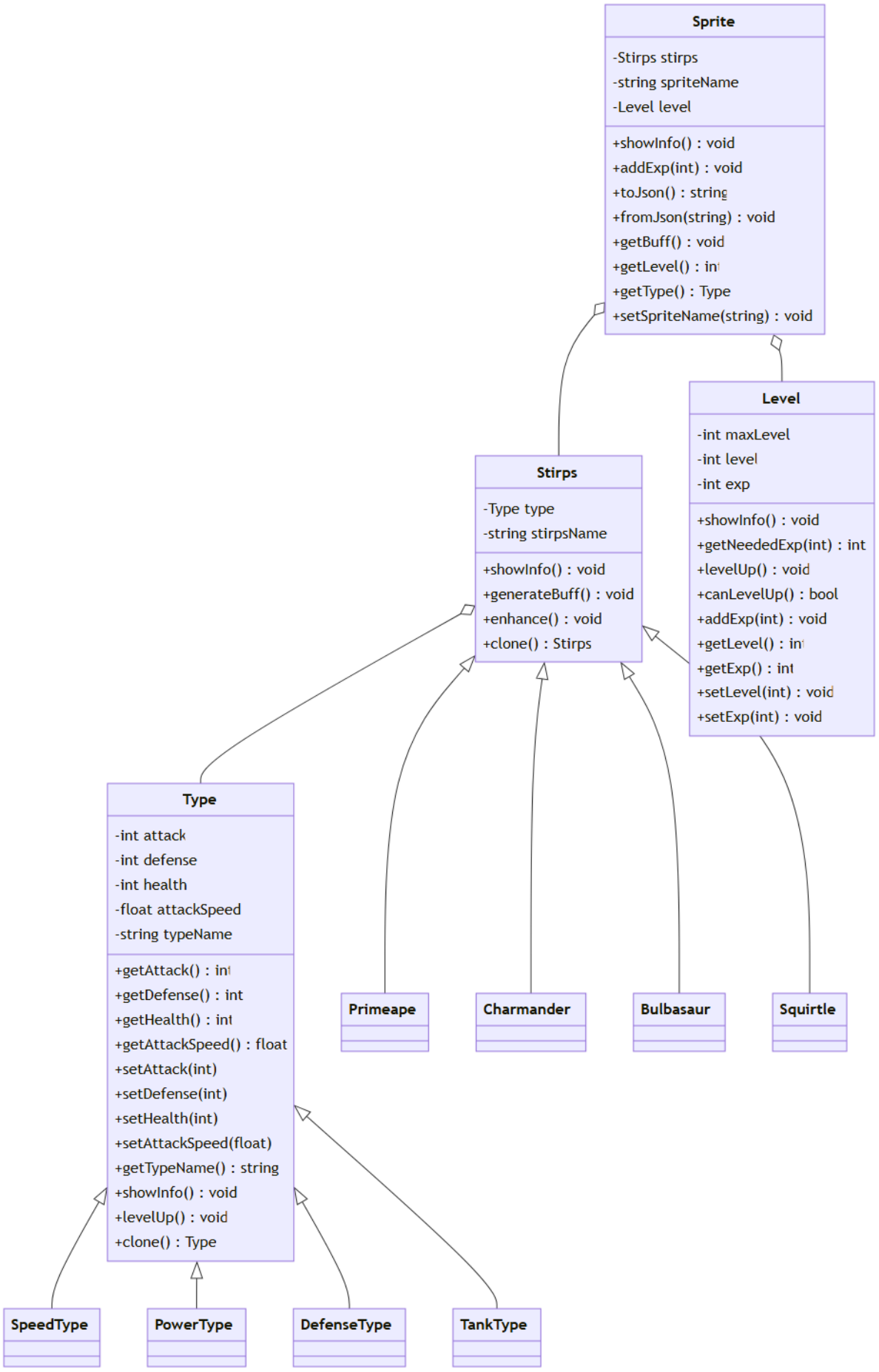
- 类型：力量型、肉盾型、防御型、敏捷型。每种不同的**类型对应不同的属性增长方式**
- 种族：对类型的特化，有自己的**准确初始属性值**
- 名字：可以由用户为自己的爱宠命名，同一个用户不能有重名的精灵
- 等级：升级到每个等级所需要的经验值递增，符合实际需求
- 经验值：使用一个非线性的递增函数确定经验值，这里采用 $y = x^2$
- 攻击力：
  $attack = type.attack \times (1 + \sum Buff.attackRate) + \sum Buff.attackIncrease$
- 防御力：
  $defence = type.defence \times (1 + \sum Buff.defenseRate) + \sum Buff.defenseIncrease$
- 生命值：生命值在对战开始时确定，每轮根据伤害扣除对应生命值，
  $hurt = attack > defence ? attack - defence : 0$
- 攻击速度：当前速度值确定公式：
  $\Delta speed = type.speed \times (1 + \sum Buff.speedRate) + \sum Buff.speedIncrease$

独特的攻击方式

每个精灵实例的技能来源于重写自基类的技能方法，每个技能返回一个`Buff`，在具体双方对战中，根据`Buff组`实时结算每轮造成的伤害

类的设计

类之间的关系图


Sprite
-Stirps stirps
-string spriteName
-Level level
+showInfo() : void
+addExp(int) : void
+toJson() : string
+fromJson(string) : void
+getBuff() : void
+getLevel() : int
+getType() : Type
+setSpriteName(string) : void
Level
-int maxLevel
-int level
-int exp
+showInfo() : void
+getNeededExp(int) : int
+levelUp() : void
+canLevelUp() : bool
+addExp(int) : void
+getLevel() : int
+getExp() : int
+setLevel(int) : void
+setExp(int) : void
Stirps
-Type type
-string stirpsName
+showInfo() : void
+generateBuff() : void
+enhance() : void
+clone() : Stirps
Type
-int attack
-int defense
-int health
-float attackSpeed
-string typeName
+getAttack() : int
+getDefense() : int
+getHealth() : int
+getAttackSpeed() : float
+setAttack(int)
+setDefense(int)
+setHealth(int)
+setAttackSpeed(float)
+getTypeName() : string
+showInfo() : void
+levelUp() : void
+clone() : Type
Primeape
Charmander
Bulbasaur
Squirtle
SpeedType
PowerType
DefenseType
TankType

类的内容

Type 类及其派生类

`Type` 类是一个抽象基类，定义了所有派生类型（`SpeedType`、`PowerType`、`DefenseType`、`TankType`）的共同属性和方法。它包括以下成员变量和方法：

- **成员变量：**
  - `int attack`：攻击力。
  - `int defense`：防御力。
  - `int health`：生命值。
  - `int attackSpeed`：攻击速度。
  - `std::string typeName`：类型名称。
- **成员方法：**
  - 构造函数：初始化成员变量。
  - 访问器方法：`getAttack`、`getDefense`、`getHealth`、`getAttackSpeed`、`getTypeName`。
  - 修改器方法：`setAttack`、`setDefense`、`setHealth`、`setAttackSpeed`。
  - `showInfo`：显示类型信息。
  - 纯虚函数：`levelUp` 和 `clone`，需要在派生类中实现。

每个派生类（`SpeedType`、`PowerType`、`DefenseType`、`TankType`）继承自 `Type` 类，并实现了 `levelUp` 和 `clone` 方法。这些派生类表示不同类型的具体实现，定义了它们在升级时的行为。

Stirps 类及其派生类

`Stirps` 类是一个抽象基类，表示各种种族。它包含一个 `Type` 类型的智能指针和种族名称。成员变量和方法如下：

- **成员变量：**
  - `std::unique_ptr<Type> type`：指向具体类型的智能指针。
  - `std::string stirpsName`：种族名称。
- **成员方法：**
  - 构造函数：初始化 `type` 和 `stirpsName`。
  - `showInfo`：显示种族信息。
  - 纯虚函数：`generateBuff`、`enhance` 和 `clone`，需要在派生类中实现。

派生类（`Primeape`、`Charmander`、`Bulbasaur`、`Squirtle`）继承自 `Stirps` 类，并实现了 `generateBuff`、`enhance` 和 `clone` 方法。每个派生类表示不同的具体种族，并定义了它们的具体行为。

Sprite 类

`Sprite` 类是对游戏中精灵的表示。它包含一个 `Stirps` 类型的智能指针、精灵名称和 `Level` 类的实例。成员变量和方法如下：

- **成员变量：**
  - `std::unique_ptr<Stirps> stirps`：指向具体种族的智能指针。

    - `std::string spriteName`：精灵名称。
    - `Level level`：精灵的等级信息。
- **成员方法：**
    - 构造函数：初始化精灵的种族、名称和等级。
    - `showInfo`：显示精灵信息。
    - `addExp`：增加经验值，并可能触发升级。
    - 拷贝构造函数和赋值运算符：用于深拷贝和深赋值。
    - 移动构造函数和赋值运算符：用于移动语义。
    - `toJson` 和 `fromJson`：序列化和反序列化。
    - `getBuff`：获取种族的 Buff。
    - `getLevel` 和 `getType`：获取等级和类型信息。
    - `setSpriteName`：设置精灵名称。

Level 类

`Level` 类表示精灵的等级信息，包括当前等级和经验值。成员变量和方法如下：

- **成员变量：**
    - `static const int maxLevel`：最大等级。
    - `int level`：当前等级。
    - `int exp`：当前经验值。
- **成员方法：**
    - 构造函数：初始化等级和经验值，并根据经验值自动升级。
    - `showInfo`：显示等级信息。
    - `getNeededExp`：计算升级所需的经验值。
    - `levelUp`：处理经验值增加并自动升级。
    - `canLevelUp`：判断是否可以升级。
    - `addExp`：增加经验值并处理升级。
    - 访问器方法：`getLevel` 和 `getExp`。
    - 修改器方法：`setLevel` 和 `setExp`。

组合与继承关系

1. **继承关系：**
    - `SpeedType`、`PowerType`、`DefenseType`、`TankType` 继承自 `Type`。
    - `Primeape`、`Charmander`、`Bulbasaur`、`Squirtle` 继承自 `Stirps`。
2. **组合关系：**
    - `Stirps` 类包含一个 `Type` 类型的智能指针，表示种族包含具体类型。
    - `Sprite` 类包含一个 `Stirps` 类型的智能指针，表示精灵包含具体种族。
    - `Sprite` 类包含一个 `Level` 类的实例，表示精灵具有等级信息。

创新点

背景和问题

在面向对象编程中，C++由于缺乏反射机制，在对象的持久化存储和动态创建时会遇到一定的困难。尤其是在需要根据对象类型动态创建对象实例时，传统的方法通常依赖于大量的`switch-case`语句，不仅代码臃肿，而且难以维护和扩展。为了解决这些问题，我们引入了工厂模式和`nlohmann::json`库来实现对象的持久化和动态创建。此外，通过一个辅助的注册类来简化新种族类型的添加，避免了传统方法的复杂性。

解决方案

工厂模式

工厂模式提供了一种创建对象的接口，使得实例化的过程与客户端代码解耦。在本设计中，`Factory`类负责管理各种类型的创建函数，并通过静态成员变量存储这些函数。

```
class Factory {
public:
    using Creator = std::function<Stirps*(int)>;

    /**
     * @brief 注册类型的创建函数
     * @tparam T 要注册的类型
     */
    template <typename T>
    static void registerCreator();

    /**
     * @brief 创建指定类型的精灵对象
     * @param typeName 类型名称
     * @param spriteName 精灵名称
     * @param level 初始等级
     * @param exp 初始经验值
     * @return 创建的精灵对象的共享指针
     */
    static std::shared_ptr<Sprite> create(const std::string& typeName, const std::string& spriteName = "", int level = 1, int exp = 0);

    /**
     * @brief 创建随机类型的精灵对象
     * @param spriteName 精灵名称
     * @param level 初始等级
     * @param exp 初始经验值
     * @return 创建的随机精灵对象的共享指针
     */
    static std::shared_ptr<Sprite> createRandomSprite(const std::string& spriteName = "", int level = 1, int exp = 0);

private:
    /**
     * @brief 获取类型创建函数的映射
     * @return 类型创建函数的映射
     */
    static std::map<std::string, Creator>& getCreators();
```

```
};
```

注册机制

为了避免使用`switch-case`来注册新的种族类型，我们引入了一个辅助的注册类`Register`。这个类在全局匿名命名空间中实例化，确保在程序启动时自动注册所有种族类型。

```cpp
template<typename T>
class Register {
public:
    /**
     * @brief 构造函数，注册类型的创建函数
     */
    Register();
};

// 添加新的种族时只需要在匿名命名空间中添加对应的注册信息即可
namespace {
    static Register<Primeape> registerPrimeape;
    static Register<Charmander> registerCharmander;
    static Register<Bulbasaur> registerBulbasaur;
    static Register<Squirtle> registerSquirtle;
}
```

对象持久化与恢复

使用`nlohmann::json`库实现对象的持久化存储和恢复。通过将对象序列化为JSON格式，可以方便地保存和读取对象状态。

```cpp
class Sprite {
public:
    // 其他成员函数...

    /**
     * @brief 将精灵对象序列化为JSON格式
     * @param sprite 要序列化的精灵对象
     * @return JSON对象
     */
    static nlohmann::json toJson(const Sprite& sprite);

    /**
     * @brief 从JSON对象恢复精灵对象
     * @param j JSON对象
     * @return 恢复的精灵对象的共享指针
     */
    static std::shared_ptr<Sprite> fromJson(const nlohmann::json& j);

    // 其他成员函数...
};
```

创新点总结

- 动态对象创建：通过工厂模式和注册机制，实现了基于类型名的动态对象创建，避免了大量的`switch-case`语句，代码简洁且易于扩展。
- 对象持久化：利用`nlohmann::json`库，方便地实现了对象的序列化和反序列化，使对象的持久化存储变得简单且高效。
- 自动注册：通过辅助的注册类，所有种族类型在程序启动时自动注册，无需手动调用注册函数，进一步简化了代码。

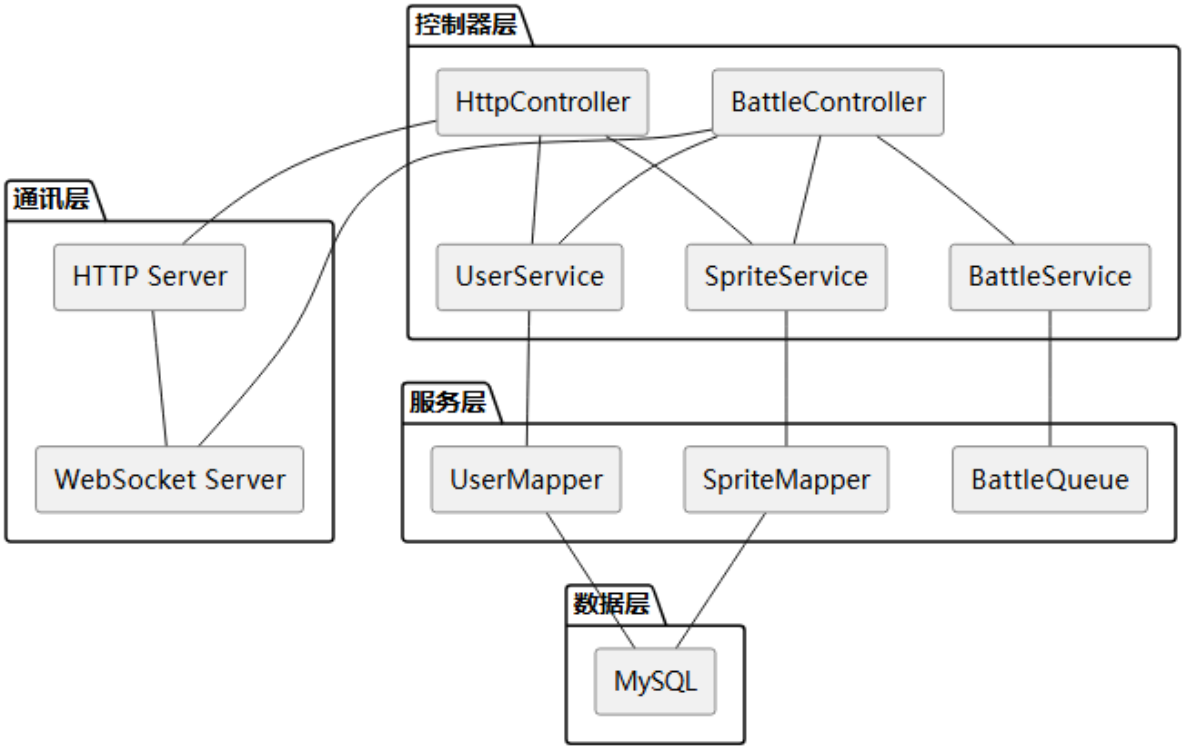
通过以上设计，实现了C++中缺乏反射机制情况下的动态对象创建和持久化存储，极大地提高了代码的可维护性和扩展性。这些创新点使得系统在添加新种族类型时更加灵活和高效。

用户注册与平台登陆

题目需求

- **网络库与通信协议**：使用libhv作为网络库，服务器和客户端之间通过HTTP协议和WebSocket协议进行通信。
- **用户信息储存**：使用MySQL数据库，使用`nlohmann::json`库作为JSON序列化和反序列化工具。
- **游戏系统初始化**：使用Command命令模式进行用户交互，在客户端构建有限状态自动机，使用命令行进行命令输入。
- **数据唯一性**：用户名全局唯一，不能有任何两个用户名相同。

后端系统架构图



后端系统设计

本设计基于`libhv`网络库，实现了一个支持HTTP和WebSocket协议的服务器与客户端通信模块。使用MySQL数据库进行数据存储，并采用`nlohmann::json`库进行JSON序列化与反序列化。

本设计采用了类似于Spring框架的三层架构，即Controller层、Service层和Mapper层。通过这种架构，代码结构清晰，职责分明，有助于提高系统的可维护性和扩展性。

Controller 层

Controller层负责处理客户端请求并将请求转发到相应的Service进行处理。Controller层主要关注与客户端的交互，不包含业务逻辑。

HttpController 接口

```
/**
 * @brief 处理HTTP请求的控制器
 */
class HttpController {
public:
    /**
     * @brief 获取HttpController的唯一实例
     * @return HttpController实例
     */
    static HttpController& getInstance();

    /**
     * @brief 注册HTTP路由
     * @param router HTTP服务对象
     */
    static void registerRoutes(hv::HttpService& router);

    // 删除拷贝构造函数和赋值运算符
    HttpController(const HttpController&) = delete;
    HttpController& operator=(const HttpController&) = delete;

private:
    UserService* userService;
    SpriteService* spriteService;
    std::shared_ptr<spdlog::logger> logger;

    /**
     * @brief 构造函数
     */
    HttpController();

    /**
     * @brief 析构函数
     */
    ~HttpController();

    /**
     * @brief 处理ping请求
     * @param ctx HTTP上下文指针
     * @return HTTP状态码
     */
    int handlePing(const HttpContextPtr& ctx);

    /**
     * @brief 处理获取用户请求
     * @param ctx HTTP上下文指针
```

```
     * @return HTTP状态码
     */
    int handleGetUser(const HttpContextPtr& ctx);

    /**
     * @brief 处理登录请求
     * @param ctx HTTP上下文指针
     * @return HTTP状态码
     */
    int handleLogin(const HttpContextPtr& ctx);

    /**
     * @brief 处理注册请求
     * @param ctx HTTP上下文指针
     * @return HTTP状态码
     */
    int handleRegister(const HttpContextPtr& ctx);

    /**
     * @brief 处理获取所有用户名请求
     * @param ctx HTTP上下文指针
     * @return HTTP状态码
     */
    int handleGetAllUserNames(const HttpContextPtr& ctx);

    /**
     * @brief 处理获取在线用户名称请求
     * @param ctx HTTP上下文指针
     * @return HTTP状态码
     */
    int handleGetOnlineUserNames(const HttpContextPtr& ctx);

    /**
     * @brief 处理退出请求
     * @param ctx HTTP上下文指针
     * @return HTTP状态码
     */
    int handleExit(const HttpContextPtr& ctx);
};
```

BattleController 接口

```
/**
 * @brief 处理WebSocket请求的控制器
 */
class BattleController {
private:
    std::map<std::string, WebSocketChannelPtr> clients;
    BattleService* battleService;
    UserService* userService;
    SpriteService* spriteService;
    std::atomic<bool> isRunning = true;
    BattleQueue::RequestQueue& reqQueue;
    BattleQueue::ResultQueue& resultQueue;
    std::thread resultThread;
```

```
14
15      /**
16       * @brief 处理WebSocket连接打开事件
17       * @param channel WebSocket通道指针
18       * @param req HTTP请求指针
19       */
20      static void onopen(const WebSocketChannelPtr& channel, const
    HttpRequestPtr& req);
21
22      /**
23       * @brief 处理WebSocket消息事件
24       * @param channel WebSocket通道指针
25       * @param msg 接收到的消息
26       */
27      static void onmessage(const WebSocketChannelPtr& channel, const
    std::string& msg);
28
29      /**
30       * @brief 处理WebSocket连接关闭事件
31       * @param channel WebSocket通道指针
32       */
33      static void onclose(const WebSocketChannelPtr& channel);
34
35      BattleController();
36      ~BattleController();
37
38      /**
39       * @brief 处理战斗请求
40       * @param msg 接收到的消息
41       * @param channel WebSocket通道指针
42       */
43      void handleBattleRequest(const std::string &msg, const
    WebSocketChannelPtr& channel);
44
45      /**
46       * @brief 处理战斗结果
47       */
48      void handleBattleResult();
49
50  public:
51      std::shared_ptr<spdlog::logger> logger;
52
53      // 删除拷贝构造函数和赋值运算符
54      BattleController(const BattleController&) = delete;
55      BattleController& operator=(const BattleController&) = delete;
56
57      /**
58       * @brief 获取BattleController的唯一实例
59       * @return BattleController实例
60       */
61      static BattleController& getInstance();
62
63      /**
64       * @brief 注册WebSocket服务
65       * @param ws WebSocket服务对象
66       */
```

```
    static void registerWebSocketService(hv::WebSocketService& ws);

    /**
     * @brief 注册HTTP路由
     * @param router HTTP服务对象
     */
    static void registerRoutes(hv::HttpService& router);
};
```

Service 层

Service层负责业务逻辑的处理。在本设计中，Service层的实现类似于Spring框架中的方式，先定义接口，再提供具体实现。Service层的主要作用是处理业务逻辑，如用户注册、登录等，确保业务逻辑与数据访问逻辑的分离。

UserService 接口

```
/**
 * @brief 用户服务接口，定义了用户相关的业务逻辑
 */
class UserService {
protected:
    UserService() = default;
    virtual ~UserService() = default;

public:
    UserService(const UserService&) = delete;
    UserService& operator=(const UserService&) = delete;

    /**
     * @brief 添加用户
     * @param name 用户名
     * @param password 用户密码
     */
    virtual void addUser(const std::string& name, const std::string&
password) = 0;

    /**
     * @brief 用户登录
     * @param name 用户名
     * @param password 用户密码
     * @return 是否登录成功
     */
    virtual bool login(const std::string& name, const std::string& password)
= 0;

    /**
     * @brief 获取用户信息
     * @param name 用户名
     * @return 用户信息的共享指针
     */
    virtual std::shared_ptr<User> getUser(const std::string& name) = 0;

    /**
     * @brief 更新用户信息
```

```
     * @param user 用户对象
     */
    virtual void updateUser(const User& user) = 0;

    /**
     * @brief 添加奖牌
     * @param name 用户名
     * @param medal 奖牌名称
     */
    virtual void addMedal(const std::string& name, const std::string& medal)
= 0;

    /**
     * @brief 增加用户胜利次数
     * @param name 用户名
     */
    virtual void addWinner(const std::string& name) = 0;

    /**
     * @brief 增加用户战斗次数
     * @param name 用户名
     */
    virtual void addBattleTimes(const std::string& name) = 0;

    /**
     * @brief 设置用户在线状态
     * @param name 用户名
     * @param online 在线状态
     */
    virtual void setOnline(const std::string& name, bool online) = 0;

    /**
     * @brief 获取在线用户名列表
     * @return 在线用户名列表
     */
    virtual std::vector<std::string> getOnlineUserNames() = 0;

    /**
     * @brief 检查用户名是否存在
     * @param username 用户名
     * @return 用户名是否存在
     */
    virtual bool isUserNameExist(const std::string& username) = 0;

    /**
     * @brief 获取所有用户名列表
     * @return 所有用户名列表
     */
    virtual std::vector<std::string> getAllUserNames() = 0;
};
```

Mapper 层

Mapper层负责数据访问，与数据库的交互。Mapper层的主要作用是与数据库交互，执行CRUD操作。通过Mapper层的实现，业务逻辑与数据访问逻辑分离，使得代码更加清晰，易于维护。

Mapper 接口

```
/**
 * @brief 数据映射基类，负责数据库连接管理
 */
class Mapper {
protected:
    static std::shared_ptr<Poco::Data::SessionPool> pool;

    Mapper();
    virtual ~Mapper() = default;

public:
    /**
     * @brief 初始化数据库连接池
     * @param connectionString 数据库连接字符串
     * @param minPoolSize 最小连接池大小
     * @param maxPoolSize 最大连接池大小
     * @param idlePoolSize 空闲连接池大小
     */
    static void initializePool(const std::string& connectionString,
                               size_t minPoolSize = 1, size_t maxPoolSize =
10,
                               size_t idlePoolSize = 5);

    /**
     * @brief 获取数据库连接池
     * @return 数据库连接池的共享指针
     */
    static std::shared_ptr<Poco::Data::SessionPool> getPool();

    // 禁止复制构造和赋值操作
    Mapper(const Mapper&) = delete;
    Mapper& operator=(const Mapper&) = delete;
};
```

UserMapper 实现

```
/**
 * @brief 用户数据映射类，继承自Mapper，负责用户相关的数据库操作
 */
class UserMapper : public Mapper {
private:
    UserMapper();

public:
    static UserMapper& getInstance();

    /**
     * @brief 根据用户名获取用户信息
```

```
     * @param name 用户名
     * @return 用户信息的共享指针
     */
    std::shared_ptr<User> getUserByName(const std::string& name);

    /**
     * @brief 添加用户
     * @param user 用户对象
     */
    void addUser(const User& user);

    /**
     * @brief 更新用户信息
     * @param user 用户对象
     */
    void updateUser(const User& user);

    /**
     * @brief 根据用户名获取用户奖牌
     * @param name 用户名
     * @return 奖牌列表
     */
    std::vector<std::string> getMedalsByName(const std::string& name);

    /**
     * @brief 添加奖牌
     * @param name 用户名
     * @param medal 奖牌名称
     */
    void addMedal(const std::string& name, const std::string& medal);

    /**
     * @brief 设置用户在线状态
     * @param name 用户名
     * @param online 在线状态
     */
    void setOnline(const std::string& name, bool online);

    /**
     * @brief 获取在线用户列表
     * @return 在线用户列表
     */
    std::vector<std::shared_ptr<User>> getOnlineUsers();

    /**
     * @brief 获取所有用户列表
     * @return 所有用户列表
     */
    std::vector<std::shared_ptr<User>> getAllUsers();
};
```

三层架构的优势

- **分层设计**：将Controller层、Service层和Mapper层分开，职责明确，使代码结构清晰，易于理解和维护。
- **解耦**：业务逻辑与数据访问逻辑分离，通过接口和实现分离，使代码更具可扩展性。
- **可测试性**：通过接口定义业务逻辑，可以方便地对每一层进行单元测试，提高系统的可靠性。
- **重用性**：Service层和Mapper层的设计使得逻辑可以被多个Controller复用，减少重复代码。
- **扩展性**：新增功能时，只需在对应层添加或修改相应的实现，其他层次无需改动，提高了系统的可扩展性。

消息队列实现的组件间通信

在本系统中，为了实现组件间的高效通信，特别是在处理 `BattleController` 和 `BattleService` 之间的匹配对战请求和结果传递时，使用了自定义实现的消息队列。消息队列确保了系统各组件之间的松耦合，并且能够有效地处理异步任务。

消息队列的定义

消息队列是一种用于异步通信的机制，其中消息可以在不同组件或线程之间传递。消息队列在本系统中的具体实现包括以下两部分：

1. **请求队列（RequestQueue）**：用于存储和传递战斗请求。
2. **结果队列（ResultQueue）**：用于存储和传递战斗结果。

每个队列都是一个线程安全的队列，支持多生产者和多消费者，确保在高并发场景下的可靠性和效率。

```
/**
 * @brief 模板类，实现了一个线程安全的消息队列，用于组件间的异步通信。
 *
 * @tparam T 队列中存储的消息类型。
 */
template <typename T>
class MessageQueue {
private:
    moodycamel::ConcurrentQueue<T> queue; ///< 底层并发队列，用于存储消息。
    std::mutex mtx; ///< 互斥锁，用于保护队列和条件变量。
    std::condition_variable conditionVariable; ///< 条件变量，用于通知队列状态的变化。
    bool shutdown = false; ///< 标志位，表示队列是否已关闭。

public:
    /**
     * @brief 将消息加入队列。
     *
     * @param item 要加入队列的消息。
     */
    void enqueue(const T& item);

    /**
     * @brief 从队列中取出消息。如果队列为空，则等待消息的到来。
     *
     * @return std::optional<T> 如果队列已关闭且为空，返回std::nullopt；否则返回取出的消息。
```

```
     */
    std::optional<T> dequeue();

    /**
     * @brief 关闭队列，通知所有等待的消费者。队列关闭后，不再接受新的消息。
     */
    void shutdownQueue();
};

```

消息队列的优势

相比于传统的阻塞调用方式，使用消息队列具有以下优势：

- **异步处理**：消息队列允许任务在后台异步处理，避免了主线程的阻塞，提高了系统的响应速度。
- **解耦**：通过消息队列，组件之间的依赖性降低，各个组件可以独立开发、测试和维护。
- **弹性伸缩**：消息队列可以很容易地适应负载的变化，通过增加或减少消费者的数量，能够平滑地扩展或缩减系统的处理能力。
- **可靠性**：消息队列提供了消息的持久化和重试机制，确保消息不会丢失，即使在系统出现故障时也能保证数据的一致性。

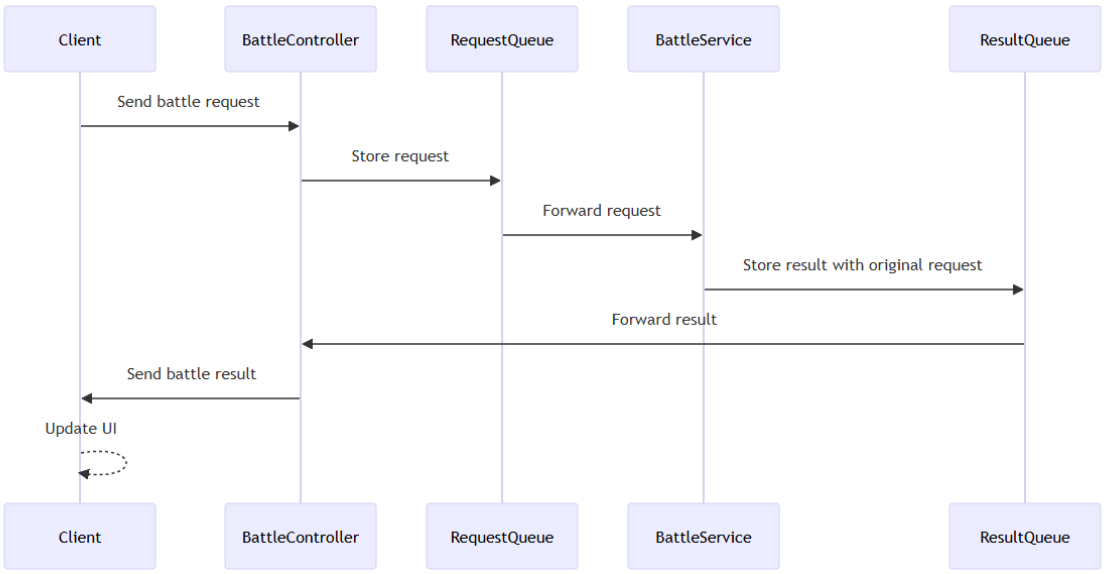
消息队列的作用

在本系统中，消息队列的主要作用包括：

1. **缓冲作用**：在`BattleController`接收到战斗请求后，将请求放入请求队列中，避免直接阻塞在处理逻辑上。
2. **异步处理**：`BattleService`从请求队列中取出请求进行处理，处理完成后将结果放入结果队列中。
3. **结果传递**：`BattleController`从结果队列中获取处理结果，并将结果发送回客户端。

组件间的通信逻辑流程

1. **接收请求**：客户端发送战斗请求到`BattleController`。`BattleController`接收到请求后，将其存储在请求队列（RequestQueue）中。
2. **处理请求**：`BattleService`作为请求的消费者，从请求队列中取出战斗请求，并进行处理。处理过程中，`BattleService`可以执行匹配对手、计算战斗结果等逻辑。
3. **存储结果**：当`BattleService`完成战斗处理后，将战斗结果连同原始请求一起放入结果队列（ResultQueue）中。
4. **返回结果**：`BattleController`作为结果的消费者，从结果队列中取出处理完成的战斗结果，并将结果发送回对应的客户端。
5. **更新客户端**：客户端接收到战斗结果后，更新UI或进行其他相应的处理。



解决的问题

使用消息队列解决了以下几个问题：

- **异步任务处理**：避免了同步调用造成的阻塞，提高了系统的并发处理能力。
- **资源竞争**：通过队列的线程安全机制，解决了多线程环境下的资源竞争问题。
- **系统解耦**：将`BattleController`和`BattleService`解耦，使得两者可以独立地进行扩展和维护。
- **提高系统性能**：异步处理避免了长时间的同步阻塞，提高了系统的整体性能和响应速度。
- **增强系统可靠性**：消息队列可以提供持久化和重试机制，确保消息不会丢失，增强了系统的可靠性。
- **提升可维护性**：通过解耦组件，系统的各个部分可以独立开发和测试，提升了系统的可维护性和扩展性。
- **灵活的任务调度**：消息队列可以灵活地调整任务的优先级和处理顺序，适应不同的业务需求。

这种设计确保了系统的高效性和灵活性，能够适应高并发和复杂业务场景。通过消息队列实现的组件间通信，不仅提高了系统的性能和响应速度，也增强了系统的可靠性和可维护性。

数据库设计

我使用了MySQL数据库来存储用户和精灵的数据，以确保数据的一致性和可靠性。定义了三个主要的数据库表：`user`、`user_medal`和`user_sprite`。每个表都具有特定的功能和数据结构，以满足系统的需求。

表结构

1. **user 表**

   `user`表用于存储用户的基本信息，包括用户名、密码、胜利次数、战斗次数和在线状态。

   - **name**: 用户名，作为主键，确保全局唯一。长度为100个字符。
   - **password**: 用户密码，用于用户登录验证。长度为100个字符。
   - **winner**: 用户的胜利次数，用于统计用户在游戏中的胜利记录。整数类型。
   - **battle_times**: 用户的战斗次数，用于统计用户参与的战斗总数。整数类型。
   - **online**: 用户的在线状态，使用`tinyint(1)`表示布尔值，0表示离线，1表示在线。默认值为0。

2. **user_medal 表**

   `user_medal` 表用于存储用户获得的奖牌信息。每条记录表示某个用户获得的一枚奖牌。

   - **name**: 用户名，作为外键，关联到 `user` 表的 `name` 字段。长度为100个字符。
   - **medal**: 奖牌名称，表示用户获得的具体奖牌。长度为100个字符。

3. **user_sprite 表**

   `user_sprite` 表用于存储用户拥有的精灵信息。每条记录表示某个用户拥有的一个精灵。

   - **username**: 用户名，作为外键，关联到 `user` 表的 `name` 字段。长度为100个字符。
   - **sprite_name**: 精灵名称，表示用户为其精灵取的名字。长度为100个字符。
   - **sprite**: 精灵的具体信息，以JSON格式存储，包含精灵的各项属性和状态。

数据库设计详细介绍

1. **用户表（user）**

   `user` 表是整个系统的核心表之一，存储了用户的基本信息和状态。该表的主键是 `name` 字段，确保了每个用户名在系统中是唯一的。这对于用户登录和身份验证至关重要。此外，`password` 字段用于存储用户的密码，系统在处理密码时应采用适当的加密措施来确保安全性。

   `winner` 和 `battle_times` 字段分别记录了用户的胜利次数和参与战斗的总次数。这些数据可以用于生成用户的排名和统计信息。`online` 字段则用于表示用户的在线状态，有助于实现实时的在线用户列表功能。

2. **用户奖牌表（user_medal）**

   `user_medal` 表用于记录用户获得的奖牌。每个用户可以获得多枚奖牌，每条记录对应用户的一个奖牌。`name` 字段作为外键，关联到 `user` 表的 `name` 字段，以确保数据的一致性和完整性。

   奖牌信息存储在 `medal` 字段中，该字段记录了具体的奖牌名称或类型。通过这种方式，系统可以轻松地扩展和管理不同种类的奖牌，并为用户展示他们的成就。

3. **用户精灵表（user_sprite）**

   `user_sprite` 表存储了用户拥有的精灵信息。每个用户可以拥有多个精灵，每条记录对应用户的一个精灵。`username` 字段作为外键，关联到 `user` 表的 `name` 字段，以确保数据的一致性。

   `sprite_name` 字段用于存储用户为其精灵取的名字，而 `sprite` 字段则以JSON格式存储精灵的详细信息。这种设计提供了极大的灵活性，允许精灵的属性和状态在不改变数据库表结构的情况下进行调整和扩展。JSON格式的使用使得数据的序列化和反序列化变得简单且高效。

数据库设计优势

- **数据一致性**：通过使用外键约束，确保 `user_medal` 和 `user_sprite` 表中的用户名与 `user` 表中的用户名一致，保持数据的一致性和完整性。
- **扩展性**：数据库表设计结构清晰，易于扩展。例如，可以很方便地在 `user_sprite` 表中添加更多精灵属性或在 `user` 表中增加新的用户信息字段。
- **高效查询**：各表的设计使得查询操作简洁高效。例如，可以通过用户名快速获取用户的所有奖牌和精灵信息。
- **数据安全**：用户密码存储在数据库中，通过适当的加密措施，可以确保用户数据的安全性。
- **灵活性**：使用JSON格式存储精灵信息，使得精灵的属性和状态可以灵活地调整和扩展，而不需要改变数据库表结构。

本设计通过精细化的数据库表结构，确保了系统的数据安全性、可靠性和扩展性，为系统的稳定运行提供了坚实的基础。

网络通信

在本系统中，网络通信模块是实现客户端与服务器之间交互的核心部分。我们通过HTTP和WebSocket两种协议实现不同类型的通信需求。在介绍这两种协议之前，首先介绍多路IO复用的定义及其优势。

多路IO复用

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。

定义： 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：

1. **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。
2. **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。
3. **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。

HTTP协议

通过`libhv`库实现HTTP服务器，处理客户端的各种HTTP请求。HTTP协议主要用于处理非实时的、一次性的请求操作。这包括用户注册、登录、获取用户信息、获取所有用户名称和在线用户名称等功能。

HTTP协议的具体应用场景：

- **用户注册**：客户端发送用户注册请求，服务器接收请求后创建新用户并返回注册结果。
- **用户登录**：客户端发送登录请求，服务器验证用户凭证并返回登录结果。
- **获取用户信息**：客户端请求获取特定用户的信息，服务器查询数据库并返回用户数据。
- **获取所有用户名称**：客户端请求获取系统中的所有用户名，服务器查询数据库并返回结果。
- **获取在线用户名称**：客户端请求获取当前在线的用户名称，服务器查询数据库并返回结果。

这些操作通过HTTP协议进行处理，适合处理短期连接和不需要保持连接的请求。

WebSocket协议

WebSocket协议是一种基于TCP的全双工通信协议，支持客户端与服务器之间的双向通信。与传统的HTTP轮询方法相比，WebSocket具有显著的优势：

1. **低延迟**：HTTP轮询需要客户端定期向服务器发送请求以获取最新数据，这种方式带来了高延迟和不必要的网络开销。WebSocket连接一旦建立，服务器可以在有新数据时立即推送给客户端，显著降低了延迟。
2. **减少网络开销**：HTTP轮询会频繁建立和关闭连接，造成大量的网络开销。WebSocket在连接建立后保持长连接，减少了连接建立和关闭的开销。
3. **实时通信**：WebSocket支持全双工通信，客户端和服务器可以随时相互发送消息，适合需要实时更新的应用场景，如实时聊天、在线游戏等。

4. **资源节约**：WebSocket协议减少了服务器资源的消耗，不需要为每个请求都进行处理和响应，可以更高效地利用服务器资源。

WebSocket协议在系统中的实际应用：

- **实时对战匹配**：客户端发送战斗匹配请求到服务器，服务器处理后通过WebSocket实时推送匹配结果。
- **实时游戏状态更新**：在对战过程中，服务器可以通过WebSocket实时推送战斗状态、结果等信息给客户端，确保玩家实时同步游戏状态。
- **在线状态管理**：服务器通过WebSocket实时更新和管理用户的在线状态，当用户上线或下线时立即通知客户端。

通过选择WebSocket而不使用传统的HTTP轮询方法，我们在系统中实现了高效、低延迟的实时通信，提升了用户体验和系统性能。

总结

在本系统中，HTTP协议和WebSocket协议各自承担了不同的通信任务。HTTP协议适用于处理一次性、短期的请求操作，如用户注册和登录。WebSocket协议则用于需要实时性和低延迟的场景，如实时对战匹配和游戏状态更新。通过合理选择和使用这两种协议，并结合多路IO复用技术，我们构建了一个高效、灵活的网络通信模块，确保了系统的高性能和良好的用户体验。

JSON序列化与反序列化

使用`nlohmann::json`库进行JSON序列化与反序列化，便于数据的存储和传输。所有涉及数据交换的地方，均使用JSON格式进行编码和解码，确保数据格式的一致性和兼容性。

设计总结

本设计通过引入`libhv`库、`nlohmann::json`库和MySQL数据库，实现了一个功能完善、结构清晰的服务器与客户端通信模块。通过三层架构设计，代码职责明确，易于维护和扩展。数据唯一性和实时通信功能的实现，确保了系统的可靠性和用户体验。

博士，您工作辛苦了。以下是修改后的游戏对战系统模块的详细报告，包含居中显示的公式和战斗经验结算的介绍。

游戏对战系统

题目需求

- 与服务器端电脑对战：
  - 升级赛：精灵增长经验，无惩罚
  - 决斗赛：获胜时可以获得被打败的精灵，失败时送出一个精灵
- 无精灵时由系统送出随机初始精灵
- 由电脑自动进行双方的攻击：有概率闪避攻击和造成暴击伤害
- 实现用户系统：用户可以有自己的徽章

对战系统设计

精灵对战系统通过维护一个Buff组来结算每次的伤害，并通过优先队列来确定攻击顺序。系统自动执行双方的攻击，根据攻击和防御属性及Buff的影响计算伤害，直至一方获胜。

Buff 数据结构

Buff表示战斗中对精灵属性的临时增益或减益效果。具体数据结构如下：

- **duration**：Buff持续生效的回合数。
- **attackRate**：Buff对攻击属性的加成比例。
- **defenseRate**：Buff对防御属性的加成比例。
- **speedRate**：Buff对速度属性的加成比例。
- **attackIncrease**：Buff对攻击属性的固定加成值。
- **defenseIncrease**：Buff对防御属性的固定加成值。
- **speedIncrease**：Buff对速度属性的固定加成值。
- **forSelf**：Buff是否作用于自身。

伤害计算公式

根据精灵的基础属性和Buff的影响，伤害计算公式如下：

攻击力：

$$attack = type.attack \times (1 + \sum Buff.attackRate) + \sum Buff.attackIncrease$$

防御力：

$$defense = type.defense \times (1 + \sum Buff.defenseRate) + \sum Buff.defenseIncrease$$

伤害值：

$$hurt = attack > defense ? attack - defense : 0$$

攻击者确定方式

为了确定攻击者的顺序，系统使用优先队列维护对战双方的精灵，优先级基于精灵的“总路程值”。“总路程值”较小的精灵优先攻击。

1. **初始化**：将对战的两只精灵放入优先队列中，按总路程值排序。
2. **选择攻击者**：从优先队列中取出头部精灵作为攻击者，并更新其总路程值：

$$\text{总路程值}_{new} = \text{总路程值}_{old} + \Delta speed$$

其中，速度值计算公式为：

$$\Delta speed = type.speed \times (1 + \sum Buff.speedRate) + \sum Buff.speedIncrease$$

3. **更新队列**：将更新后的精灵重新放入优先队列，确定下一次攻击顺序。

产生Buff方式

根据精灵不同的类型，产生不同类型的Buff。Buff的类型和数值由随机数生成，包括根据权重的随机数选择和正态分布确定的数值。

1. **权重随机数**：使用随机数生成器根据设定的权重随机选择Buff的类型。例如，力量型精灵的攻击Buff权重较高，防御Buff权重次之，速度Buff权重最低。
2. **正态分布随机数**：使用正态分布生成Buff的具体数值，正态分布的均值和标准差根据精灵的类型进行调整。例如，力量型精灵的攻击Buff均值较高，而防御型精灵的防御Buff均值较高。

```
// 攻击/防御/速度Buff的权重
std::vector<double> weights = {0.3, 0.3, 0.4};
int index = RandomUtil::weightedRandomIndex(weights);

double rate = RandomUtil::normalDistributedRandom(0.1, 0, 2);
int increase = std::round(RandomUtil::normalDistributedRandom(4, 0, 100));
int duration = std::round(RandomUtil::normalDistributedRandom(1, 0, 3));
```

这种设计的好处在于：

- **随机性**：每次战斗生成的Buff具有随机性，增加了游戏的不可预测性和趣味性。
- **可控性**：通过调整正态分布的均值和标准差，可以控制不同类型精灵的Buff特点，确保游戏的平衡性。

暴击伤害和闪避机制

在精灵对战系统中，为了增加战斗的随机性和趣味性，系统引入了暴击伤害和闪避机制。

闪避机制

闪避机制使得攻击有一定概率完全失效，即伤害值为0。在计算伤害时，如果随机数小于某个阈值（例如0.05），则认为本次攻击被闪避。

实现方式如下：

```
int damage = attacker.getAttack() - defender.getDefense();
if (damage < 0 || RandomUtil::random() < 0.05) {
    damage = 0;
    message = "Miss!";
    logs.push_back(message);
}
```

在这段代码中，`RandomUtil::random()`生成一个0到1之间的随机数，如果该随机数小于0.05，则本次攻击被闪避，伤害值设为0，并记录闪避日志。

暴击机制

暴击机制使得攻击有一定概率造成额外的伤害，增加战斗的不确定性。暴击伤害通过伤害值增益的正态分布来实现，具体数值根据正态分布生成。

实现方式如下：

```
double rate = RandomUtil::normalDistributedRandom(0.1, 0, 2);
int increase = std::round(RandomUtil::normalDistributedRandom(4, 0, 100));
int duration = std::round(RandomUtil::normalDistributedRandom(1, 0, 3));
```

通过这种方式，系统能够在每次攻击时考虑闪避和暴击的可能性，增加了战斗的不可预测性和趣味性，为玩家带来更丰富的游戏体验。

战斗经验结算

在战斗结束后，系统根据胜者和败者的等级差距来计算经验值，采用指数函数进行经验值计算：

1. **胜者经验计算**：

$$winnerExp = round(\frac{e^{loserLevel - winnerLevel}}{10}) + 10$$

这意味着如果败者的等级远高于胜者，胜者将获得较高的经验值，反之则较少。

2. **败者经验计算**：

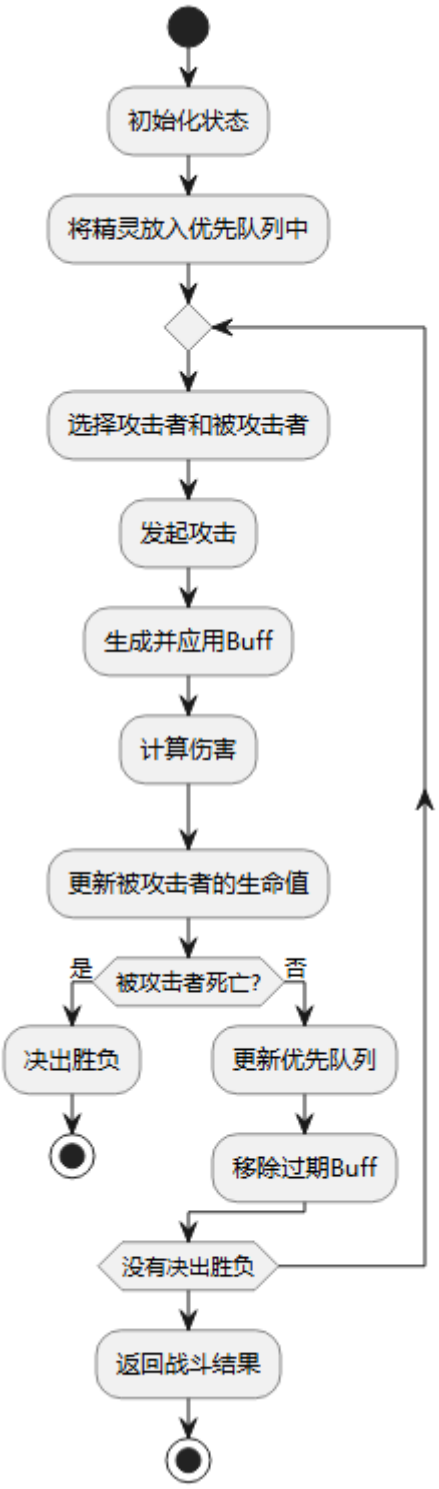
$$loserExp = round(\frac{e^{winnerLevel - loserLevel}}{10}) + 5$$

这意味着如果败者的等级远低于胜者，败者将获得较高的经验值，反之则较少。

这种经验结算方式确保了战斗的公平性，使得战斗更加有趣和平衡。

战斗过程

1. **初始化状态**：双方精灵初始化状态，清除之前的Buff效果，并更新优先级。将两只精灵放入优先队列中。
2. **开始战斗**：
    - **每一回合**：
        1. **选择攻击者和被攻击者**：从优先队列中取出两只精灵，头部精灵作为攻击者，尾部精灵作为被攻击者。
        2. **发起攻击**：攻击者发起攻击，生成一个Buff状态，并根据Buff的作用对象（自身或对方）应用Buff。
        3. **计算伤害**：根据攻击力、防御力和Buff的影响计算伤害值，并更新被攻击者的生命值。
        4. **记录日志**：记录当前回合的攻击和伤害情况，便于战斗结束后回顾。
        5. **检查胜负**：判断被攻击者是否死亡。如果被攻击者死亡，则决出胜负，生成战斗结果（包括胜者、败者、经验值和战斗日志），结束战斗。
        6. **更新状态**：如果没有决出胜负，更新双方精灵的优先级和Buff效果，移除过期的Buff，将更新后的精灵重新放入优先队列，进入下一回合。
3. **返回结果**：当战斗结束时，返回包含战斗详细信息的结果对象，包括胜者、败者、经验值和战斗日志。



总结

精灵对战系统通过维护一个Buff组来计算每次的伤害，使用优先队列来确定攻击顺序，实现了自动化、实时的战斗流程。通过精细化的状态管理和伤害计算公式，系统能够提供公平、合理的对战体验。同时，战斗日志的记录使得每次对战的细节透明可查，提升了用户的参与感和游戏体验。

对战结果结算

对战结果结算包括升级战和决斗战两种模式。升级战没有任何惩罚，而决斗战中，失败的一方需要送出一个自己的精灵。对战结果的结算通过调用`BattleService`、`UserService`和`SpriteService`等服务来实现。

对战流程

1. **接收对战请求**：`BattleController`类通过WebSocket接收对战请求，并将请求加入到任务队列中。
2. **匹配对战对象**：`BattleService`负责处理对战请求，进行匹配对战。
3. **处理对战结果**：对战结束后，`BattleService`将对战结果放入结果队列中。
4. **结算对战结果**：`BattleController`从结果队列中取出对战结果，进行结算。

升级战结算流程

升级战没有任何惩罚，仅为参与对战的精灵增加经验值。

1. **处理对战结果**：从结果队列中取出对战结果。
2. **增加经验值**：根据对战结果，为精灵增加经验值。胜者获得较多经验值，败者获得较少经验值。
3. **更新精灵信息**：调用`SpriteService`更新精灵的信息。

决斗战结算流程

决斗战中，失败的一方需要送出一个自己的精灵，胜者获得败者的一个精灵。

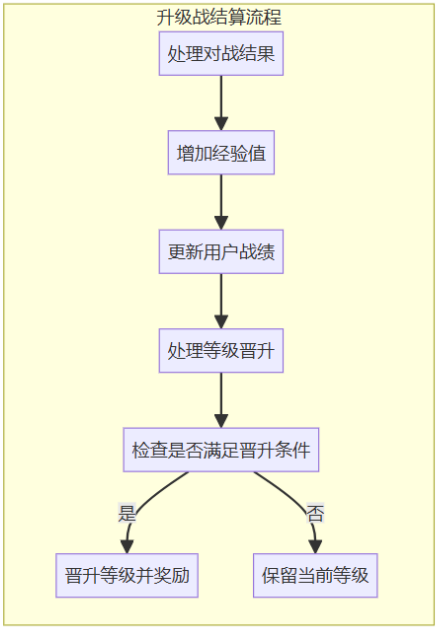
1. **处理对战结果**：从结果队列中取出对战结果。
2. **增加经验值**：根据对战结果，为精灵增加经验值。胜者获得较多经验值，败者获得较少经验值。
3. **更新用户战绩**：根据对战结果，调用`UserService`更新用户的胜率和战斗次数。胜者增加胜利次数，败者增加战斗次数。
4. **处理精灵转移**：如果失败者需要送出一个精灵，则从失败者的精灵列表中选择一个精灵转移给胜者。具体流程如下：
    - 生成新精灵名称，避免名称冲突。
    - 获取败者的精灵信息，更新精灵的拥有者信息。
    - 调用`SpriteService`移除失败者的精灵，并将更新后的精灵添加到胜者的精灵列表中。
5. **检查精灵数量**：确保失败者至少有一个精灵。如果失败者没有精灵，则系统会赠送一个初始精灵给失败者。
6. **更新勋章**：调用`UserService`根据用户的精灵数量和等级，更新用户的勋章信息。具体包括：
    - 根据用户拥有的精灵数量，更新数量勋章（如铜、银、金数量勋章）。
    - 根据用户拥有的高等级精灵数量，更新等级勋章（如铜、银、金等级勋章）。

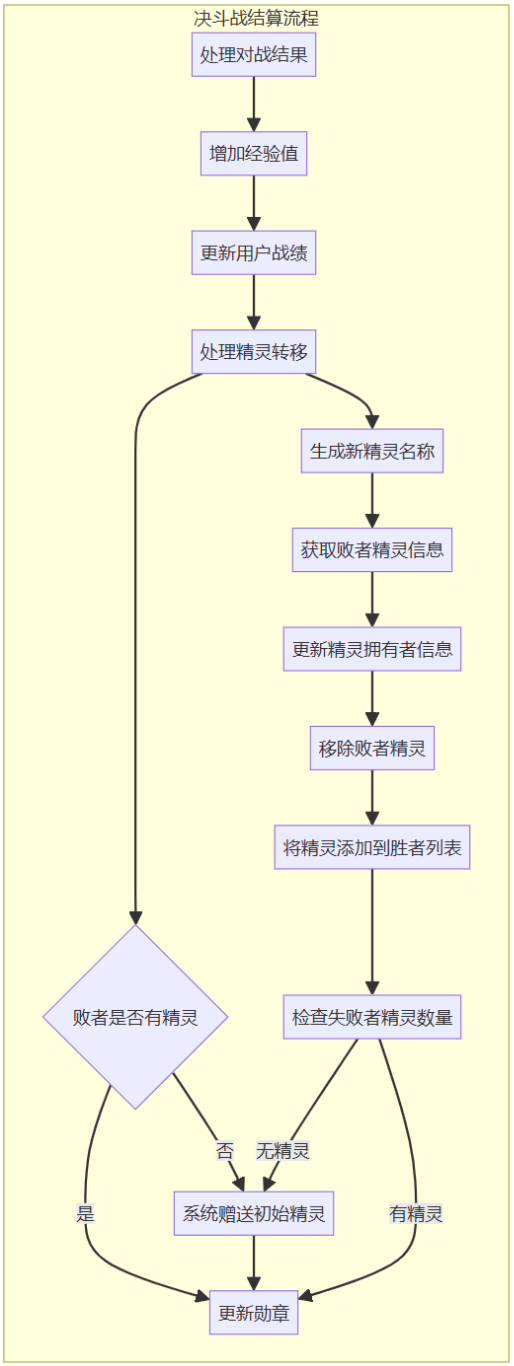
具体结算逻辑

1. **处理对战结果**：
    - 从结果队列中取出对战结果，包括对战的参与者、胜者、败者以及获得的经验值。
2. **增加经验值**：
    - 根据对战结果为双方精灵增加经验值。胜者的精灵获得更多经验值，败者的精灵获得较少经验值。
    - 调用`SpriteService`更新精灵的信息，包括经验值和等级的变化。
3. **更新用户战绩**：
    - 调用`UserService`更新用户的战绩信息。胜者增加胜利次数，败者增加战斗次数。
4. **处理精灵转移**：

- 在决斗战中，失败者需要送出一个精灵。
- 生成一个新的精灵名称，确保名称不重复。
- 获取失败者的精灵信息，更新精灵的拥有者信息。
- 调用 `SpriteService` 移除失败者的精灵，并将更新后的精灵添加到胜者的精灵列表中。
- 如果失败者的精灵数量为零，系统会赠送一个随机的初始精灵。

5. **更新勋章**：
   - 根据用户的精灵数量和等级，更新用户的勋章信息。
   - 如果用户拥有的精灵数量达到一定标准，授予对应的数量勋章（铜、银、金）。
   - 如果用户拥有的高等级精灵数量达到一定标准，授予对应的等级勋章（铜、银、金）。





通过调用各个服务实现对战结果的结算，系统能够在对战结束后进行经验值增加、精灵转移和勋章更新，确保了对战的公平性和游戏体验的丰富性。升级战和决斗战的不同结算方式增加了游戏的多样性和挑战性，为玩家提供了更丰富的游戏体验。

客户端

客户端使用了命令设计模式，通过构建有限状态自动机，输入不同的命令在各个状态间进行转换。

状态管理

客户端的状态管理由`Context`类和具体状态类（如`AuthenticatedState`和`UnauthenticatedState`）实现。所有状态类都继承自`State`类，通过状态映射表定义各自的命令和行为。相比于传统的`switch-case`方式，这种设计具有更高的可扩展性和维护性。

状态类

所有状态类都继承自`State`类，每个状态类定义了该状态下可执行的命令及其行为。通过命令映射表（`commands`）将命令字符串映射到具体的命令处理函数。

- `State`：基类，定义了基本的命令处理逻辑和命令映射表。
- `AuthenticatedState`：用户已认证状态，定义了已登录用户可执行的命令。
- `UnauthenticatedState`：用户未认证状态，定义了未登录用户可执行的命令。
- `ExitState`：退出状态，定义了退出客户端时的命令。

Context 类

`Context`类是状态管理的核心，通过单例模式确保全局唯一的实例，并提供状态切换和请求处理的功能。

- `setState`：设置当前状态。
- `request`：处理客户端请求，根据当前状态执行相应的命令。
- `initWebSocket`：初始化WebSocket连接，定义WebSocket事件处理逻辑。

状态转换方式

通过状态映射表实现状态转换，避免了传统`switch-case`方式的冗长和不易维护的问题。状态映射表将命令字符串与具体的命令处理函数关联，便于添加新命令和修改现有命令的行为。

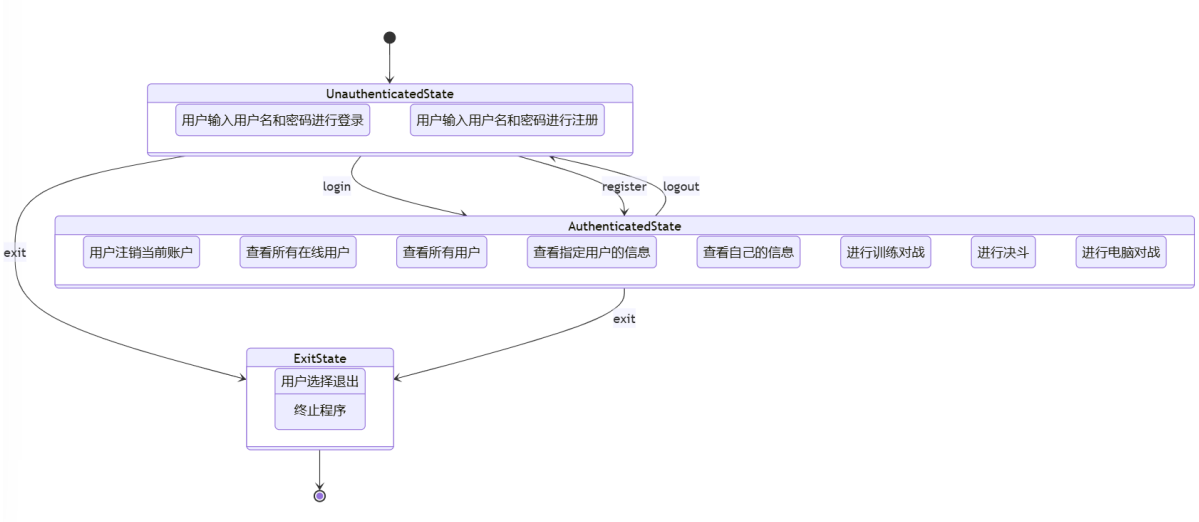
状态转换的优势

- **可扩展性**：可以方便地添加新的状态和命令，而无需修改现有代码。
- **维护性**：通过映射表定义命令，逻辑清晰，易于理解和维护。
- **解耦**：每个状态类独立定义自己的命令处理逻辑，避免了集中在一个`switch-case`语句中的代码膨胀。

客户端的状态转移

客户端在初始化时设置初始状态，并通过用户输入的命令在不同状态间进行转换。以下是状态转换的示例：

1. **登录**：用户在未认证状态下输入`login`命令，成功登录后状态切换为`AuthenticatedState`。
2. **注销**：用户在已认证状态下输入`logout`命令，成功注销后状态切换为`UnauthenticatedState`。
3. **注册**：用户在未认证状态下输入`register`命令，成功注册后状态切换为`AuthenticatedState`。
4. **退出**：用户在任何状态下输入`exit`命令，状态切换为`ExitState`，并退出客户端。



UnauthenticatedState

- **login**：用户输入用户名和密码进行登录，成功后状态切换为`AuthenticatedState`。
- **register**：用户输入用户名和密码进行注册，成功后状态切换为`AuthenticatedState`。
- **exit**：用户选择退出，状态切换为`ExitState`。

AuthenticatedState

- **logout**：用户注销当前账户，状态切换为`UnauthenticatedState`。
- **online_list**：查看所有在线用户。
- **all_list**：查看所有用户。
- **info**：查看指定用户的信息。
- **my_info**：查看自己的信息。
- **pve_train**：进行训练对战。
- **pvp_battle**：进行决斗。
- **pve_battle**：进行电脑对战。

ExitState

- **exit**：终止程序。

WebSocket 连接管理

为了实现实时通信，客户端使用了WebSocket协议。`Context`类的`initWebSocket`方法负责初始化WebSocket连接，并定义了以下事件处理逻辑：

- **onopen**：WebSocket连接打开时的处理逻辑，设置连接状态并通知等待线程。
- **onmessage**：接收到消息时的处理逻辑，解析消息并将其放入消息队列。
- **onclose**：WebSocket连接关闭时的处理逻辑，设置连接状态。

Command设计模式

通过命令设计模式，客户端可以灵活地添加新命令和修改现有命令的行为。每个状态类包含一个命令映射，将命令字符串映射到具体的命令处理函数。客户端通过使用命令设计模式和有限状态自动机，实现了灵活的状态管理和命令处理。通过WebSocket协议实现实时通信，提高了系统的响应速度和用户体验。整体设计简洁、可扩展，便于后续的功能扩展和维护。