

# 金铲铲之战项目文档

## 目录：

项目概述

项目背景

项目简介

项目主要功能

使用设计模式重构

使用创建型模式（Creational Patterns）重构

使用结构型模式（Structural Patterns）重构

使用行为模式（Behavioral Patterns）重构

使用课程未涉及的设计模式重构

项目展示

总结

使用设计模式重构的意义

本项目的重构方法

总结

参考资料

项目名称：金铲铲之战（2023年同济大学程序设计范式课程项目）

团队编号：16组

团队成员：

姓名	学号	联系方式	电子邮件
林继申 Jishen LIN	2250758	15143305542	2250758@tongji.edu.cn
刘淑仪 Shuyi LIU	2251730	15256633616	2251730@tongji.edu.cn
杨宇琨 Yukun YANG	2252843	18186206130	2242843@tongji.edu.cn
中谷天音 AMANE NAKATANI	2256225	18202190215	2256225@tongji.edu.cn

**备注：**本项目基于 Cocos2d-x 3.17.2 引擎开发，源代码整体体积较大。为了方便传输和存储，压缩包中已移除 Cocos2d 库函数代码及资源文件（包括音频、视频等设计素材），仅保留了项目的核心逻辑代码和重构部分的代码。重构的代码已通过注释明确标记，以便与原始代码区分。关于如何运行或构建本项目的说明，请参考原始 [GitHub 仓库](#) 中的相关文档。

## 项目概述

### 项目背景

随着自走棋类游戏的兴起，如《Dota 自走棋》、《云顶之弈》和《金铲铲之战》等，这类游戏凭借其独特的策略性和随机性吸引了大量玩家。自走棋游戏的核心玩法是通过收集、升级和搭配不同的英雄卡牌，形成强大的阵容与其他玩家或 AI 进行对战。这类游戏不仅考验玩家的策略思维，还提供了丰富的游戏体验和社交互动。

基于这一背景，本项目旨在开发一款类似的自走棋游戏，采用 Cocos2d-x 3.17.2 引擎进行开发。通过借鉴《Dota 自走棋》、《云顶之弈》和《金铲铲之战》等经典游戏的玩法，本项目不仅实现了基础的自走棋功能，如卡牌收集、升级、阵容搭配等，还扩展了多种羁绊加强功能和强化符文系统，增加了游戏的策略深度和可玩性。此外，游戏支持单人练习模式和多人联机对战，玩家可以与AI或其他玩家进行对战，享受丰富的游戏体验。

项目还注重游戏的视听体验，设计了引人入胜的初始界面和设置界面，加入了背景音效和战斗音效，提升了游戏的沉浸感。通过这一项目，玩家可以在一个充满策略和挑战的自走棋世界中，体验到收集、升级、搭配卡牌的乐趣，并与朋友或其他玩家一较高下。

### 项目简介

本项目是一款基于 Cocos2d-x 3.17.2 引擎开发的自走棋类游戏，灵感来源于《Dota 自走棋》《云顶之弈》和《金铲铲之战》等经典自走棋游戏。游戏以策略为核心，玩家通过收集、升级和搭配不同的英雄卡牌，组建强大的阵容，与其他玩家或AI进行对战。游戏不仅还原了经典自走棋的核心玩法，还加入了独特的创新元素，为玩家提供丰富的策略选择和沉浸式的游戏体验。

游戏的主要特色包括：

- 多样化的卡牌系统：**支持多种类型的卡牌，每种卡牌拥有独特的技能和属性，玩家可以通过升级卡牌提升其战斗力。
- 羁绊与符文系统：**扩展了多种羁绊加强功能和强化符文系统，玩家可以通过合理的阵容搭配和符文选择，发挥出更强的战斗力。
- 单人练习与多人联机：**支持练习模式，玩家可以与 AI 对战，提升自己的策略水平；同时支持联机模式，玩家可以创建房间或加入其他玩家的房间，与好友或其他玩家一较高下。
- 视听体验优化：**游戏设计了精美的初始界面和设置界面，加入了背景音效和战斗音效，增强了游戏的沉浸感和趣味性。

- **小小英雄与技能释放：**玩家可以操控小小英雄在棋盘上移动，场上卡牌拥有红蓝血条，蓝条满时可释放技能，增加了游戏的策略性和操作性。

本项目的目标是为玩家提供一个兼具策略深度和娱乐性的自走棋游戏，同时通过丰富的功能和创新的玩法，打造一个独特的自走棋世界。无论是单人练习还是多人对战，玩家都能在游戏中体验到自走棋的乐趣与挑战。

## 项目主要功能

### 1. 初始界面与设置界面

- 提供直观且吸引人的初始界面，玩家可以快速开始游戏或进入设置界面。
- 设置界面支持调整音效、画质、按键配置等个性化选项，提升玩家体验。

### 2. 背景音效与音乐

- 支持动态背景音乐，根据游戏场景切换不同的音效，增强沉浸感。
- 提供音效开关选项，玩家可根据喜好自定义音效设置。

### 3. 卡牌系统

- 支持多种类型的卡牌，每种卡牌拥有独特的属性、技能和羁绊效果。
- 卡牌分为不同等级和职业，玩家需要通过策略搭配来组建最强阵容。

### 4. 卡牌升级功能

- 玩家可以通过消耗相同卡牌或资源对卡牌进行升级，提升其属性和技能效果。
- 升级后的卡牌外观和特效会发生变化，增强视觉表现力。

### 5. 小小英雄系统

- 玩家可以操控小小英雄在棋盘上移动，小小英雄拥有独特的皮肤和动作。
- 小小英雄不仅是玩家的代表，还能通过互动增加游戏的趣味性。

### 6. 战斗系统

- 场上卡牌拥有红蓝血条，蓝条满时可释放技能，技能效果根据卡牌类型和等级有所不同。
- 战斗过程实时计算伤害和技能效果，提供流畅的战斗体验。

### 7. 房间功能

- 支持创建房间和加入房间，玩家可以邀请好友或其他玩家进行对战。
- 房间内支持聊天功能，方便玩家交流策略或互动。

### 8. 练习模式

- 支持单人练习模式，玩家可以与N个AI玩家对弈（ $N \geq 2$ ），用于熟悉游戏规则和测试阵容。
- AI难度可调节，适合不同水平的玩家。

### 9. 联机模式

- 支持多人联机对战，玩家可以与N个人类玩家同台竞技（N ≥ 2）。
- 联机模式采用实时同步技术，确保战斗过程的公平性和流畅性。

## 10. 羁绊系统

- 支持多种羁绊效果，玩家通过搭配特定类型的卡牌激活羁绊，获得属性加成或特殊效果。
- 羁绊系统增加了游戏的策略深度，玩家需要合理规划阵容以最大化羁绊收益。

## 11. 强化符文系统

- 提供多种强化符文，玩家可以在游戏过程中选择符文来增强卡牌或阵容的战斗力。
- 符文效果多样，包括属性提升、技能强化、羁绊加成等，增加了游戏的随机性和可玩性。

## 12. 战斗音效与特效

- 战斗中支持技能释放音效、攻击音效和胜利/失败音效，提升战斗的紧张感和代入感。
- 技能特效经过精心设计，视觉效果炫酷且符合卡牌主题。

# 使用设计模式重构

## 使用创建型模式（Creational Patterns）重构

## 使用生成器模式（Builder Pattern）重构

### 原有问题

在重构前的代码中，`Champion` 类的初始化设计存在以下问题：

### 1. 维护性问题

- **常量定义分散：** `CHAMPION_ATTR_MAP` 中的常量定义分散在多个地方，导致维护困难。每次新增一个英雄时，都需要在多个地方修改代码，容易出错。

```
1 enum ChampionCategory {  
2     NoChampion,           // 无战斗英雄  
3     Champion1,          // 劫（一星）  
4     Champion2,          // 劫（二星）  
5     Champion3,          // 永恩（一星）  
6     Champion4,          // 永恩（二星）  
7     Champion5,          // 奥拉夫（一星）  
8     ...  
9 };
```

```
1 typedef struct {
```

```

2     ChampionCategory championCategory; // 战斗英雄种类
3     std::string championName;           // 战斗英雄名称
4     std::string championImagePath;      // 战斗英雄图片路径
5     int price;                      // 价格
6     int level;                      // 等级
7     int healthPoints;                // 生命值
8     int magicPoints;                 // 魔法值
9     int attackDamage;                // 攻击伤害
10    int skillTriggerThreshold;       // 技能触发阈值
11    int attackRange;                 // 攻击范围
12    float attackSpeed;              // 攻击速度
13    float movementSpeed;            // 移动速度
14    float defenseCoefficient;       // 防御系数
15    Bond bond;                     // 羁绊效果
16 } ChampionAttributes;

```

```

1 const std::map<ChampionCategory, ChampionAttributes> CHAMPION_ATTR_MAP = {
2     {Champion1, CHAMPION_1_ATTR},    // 劫 (一星)
3     {Champion2, CHAMPION_2_ATTR},    // 劫 (二星)
4     {Champion3, CHAMPION_3_ATTR},    // 永恩 (一星)
5     {Champion4, CHAMPION_4_ATTR},    // 永恩 (二星)
6     {Champion5, CHAMPION_5_ATTR},    // 奥拉夫 (一星)
7     ...
8 };

```

- **硬编码：**英雄的属性（如攻击力、防御系数等）是硬编码在 `CHAMPION_ATTR_MAP` 中的，这会导致代码的可读性和可维护性变差。

```

1 const ChampionAttributes CHAMPION_1_ATTR = { // 劫 (一星)
2     Champion1, u8"劫", "../Resources/Champions/Champion1.png", 1, 1, 500, 0,
3     50, 200, 1, 0.75f, 1.0f, 1.0f, NoBond
4 };
5 const ChampionAttributes CHAMPION_2_ATTR = { // 劫 (二星)
6     Champion2, u8"劫", "../Resources/Champions/Champion2.png", 3, 2, 900, 0,
7     75, 200, 1, 0.75f, 1.0f, 1.0f, NoBond
8 };
9 const ChampionAttributes CHAMPION_3_ATTR = { // 永恩 (一星)
10    Champion3, u8"永恩", "../Resources/Champions/Champion3.png", 1, 1, 550, 0,
11    45, 200, 1, 1.0f, 1.0f, 1.2f, Brotherhood
12 };
13 const ChampionAttributes CHAMPION_4_ATTR = { // 永恩 (二星)
14    Champion4, u8"永恩", "../Resources/Champions/Champion4.png", 3, 2, 990, 0,
15    68, 200, 1, 1.0f, 1.0f, 1.2f, Brotherhood

```

```
12 };
13 const ChampionAttributes CHAMPION_5_ATTR = { // 奥拉夫 (一星)
14     Champion5, u8"奥拉夫", "../Resources/Champions/Champion5.png", 1, 1, 700,
15     0, 55, 200, 1, 0.6f, 1.0f, 1.4f, Lout
16 };
```

- 当前代码中，`Champion` 类的构造函数直接通过 `CHAMPION_ATTR_MAP` 来初始化属性，这种方式使得代码的可维护性和可扩展性较差。

```
1 Champion::Champion(const ChampionCategory championCategory) :
2     currentBattle(nullptr),
3     attributes(CHAMPION_ATTR_MAP.at(championCategory)),
4     currentLocation({ 0, 0 }),
5     currentDestination({ 0, 0 }),
6     currentEnemy(nullptr),
7     sword(nullptr),
8     healthBar(nullptr),
9     manaBar(nullptr),
10    currentMove(nullptr),
11    isMyFlag(false),
12    isMoving(false),
13    isAttaking(false),
14    attackIntervalTimer(0.0f),
15    moveIntervalTimer(0.0f)
16 {
17     sprite = Sprite::create(attributes.championImagePath);
18     maxHealthPoints = attributes.healthPoints;
19     maxMagicPoints = attributes.skillTriggerThreshold;
20 }
```

## 2. 扩展性问题

- **新增属性困难：**如果未来需要为英雄增加新的属性（如暴击率、闪避率等），需要在 `ChampionAttributes` 结构体中添加新的字段，并在 `CHAMPION_ATTR_MAP` 中为每个英雄设置该属性。这会导致代码量急剧增加，且容易遗漏某些英雄的属性设置。
- **属性冗余：**某些属性可能只对部分英雄有效，但在当前设计中，所有英雄都必须拥有这些属性，导致冗余。例如，远程英雄可能不需要近战英雄的某些属性（如攻击范围），反之亦然。

## 3. 初始化复杂度过高

- **构造函数参数过多：**`Champion` 类的构造函数需要初始化大量参数，导致代码冗长且难以阅读。如果未来需要增加更多属性，构造函数会变得更加复杂。

- **默认值问题**: 某些属性可能有默认值，但在当前设计中，所有属性都必须显式初始化，导致代码冗余。

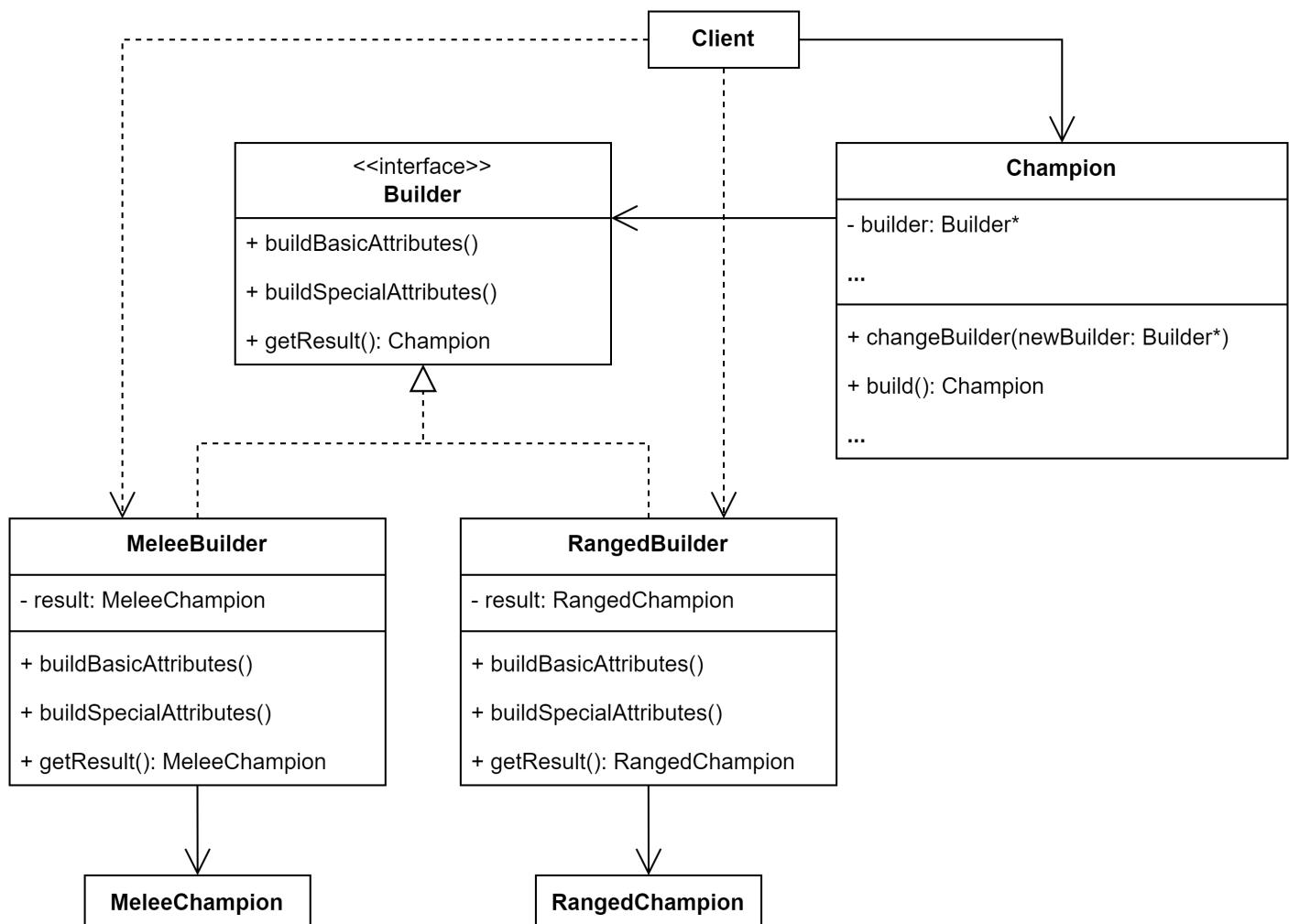
#### 4. 缺乏灵活性

- **固定的属性设置**: 当前设计中的英雄属性是固定的，无法在运行时动态调整。如果未来需要根据游戏进度或玩家选择动态调整英雄属性，当前设计将无法满足需求。

#### 重构必要性

在重构前的代码中，`Champion` 类的初始化设计存在多个问题，包括维护性差、扩展性不足、初始化复杂度过高以及缺乏灵活性。具体来说，常量定义分散在多个地方，导致维护困难；硬编码的英雄属性使得代码可读性和可维护性变差；新增属性需要在多个地方修改代码，容易出错；构造函数参数过多，导致代码冗长且难以阅读；属性冗余和默认值问题也增加了代码的复杂性。此外，当前设计无法在运行时动态调整英雄属性，缺乏灵活性。为了解决这些问题，使用生成器模式进行重构是非常必要的。生成器模式可以将对象的构建过程与其表示分离，使得代码更加模块化、可维护和可扩展。

#### UML 类图



生成器模式 UML 类图

#### 重构步骤

为了解决上述问题，我们使用生成器模式对 `Champion` 类进行重构。以下是重构的具体步骤：

## 1. 定义 Builder 接口

我们定义了一个 `Builder` 接口，其中包含三个纯虚函数：`buildBasicAttributes`、`buildSpecialAttributes` 和 `getResult`。`buildBasicAttributes` 用于构建英雄的基础属性，如生命值和攻击力；`buildSpecialAttributes` 用于构建英雄的特殊属性，如攻击范围和防御系数；`getResult` 则返回构建完成的 `Champion` 对象。这个接口为具体的生成器类提供了统一的构建流程，确保所有生成器都遵循相同的构建步骤。

```
1 class Builder {
2 public:
3     virtual ~Builder() = default;
4     virtual void buildBasicAttributes() = 0;
5     virtual void buildSpecialAttributes() = 0;
6     virtual Champion getResult() = 0;
7 };
```

## 2. 实现具体的生成器类

我们实现了两个具体的生成器类：`MeleeBuilder` 和 `RangedBuilder`。`MeleeBuilder` 用于构建近战英雄，`RangedBuilder` 用于构建远程英雄，他们具有不同的基础属性和特殊属性。

```
1 class MeleeBuilder : public Builder {
2 private:
3     ChampionAttributes attributes;
4
5 public:
6     void buildBasicAttributes() override {}
7     void buildSpecialAttributes() override {}
8     Champion getResult() override {}
9 };
```

```
1 class RangedBuilder : public Builder {
2 private:
3     ChampionAttributes attributes;
4
5 public:
6     void buildBasicAttributes() override {}
7     void buildSpecialAttributes() override {}
8     Champion getResult() override {}
9 };
```

### 3. 重构 Champion 类

在 Champion 类中，我们添加了一个 Builder 指针，并提供了 changeBuilder 和 build 方法。changeBuilder 方法用于动态切换生成器，允许我们在运行时选择使用 MeleeBuilder 或 RangedBuilder。build 方法则调用生成器的 buildBasicAttributes 和 buildSpecialAttributes 方法，完成英雄的构建，并返回构建结果。

```
1 void Champion::changeBuilder(Builder* newBuilder) {
2     if (builder) {
3         delete builder;
4     }
5     builder = newBuilder;
6 }
7
8 Champion Champion::build() {
9     if (builder) {
10         builder->buildBasicAttributes();
11         builder->buildSpecialAttributes();
12         return builder->getResult();
13     }
14     throw std::runtime_error("No builder set!");
15 }
```

### 4. 使用生成器模式创建近战英雄和远程英雄

下面我们通过生成器模式创建近战英雄和远程英雄。

```
1 MeleeBuilder meleeBuilder;
2 Champion champion1;
3 champion1.changeBuilder(&meleeBuilder);
4 Champion meleeChampion = champion1.build();
5
6 RangedBuilder rangedBuilder;
7 Champion champion2;
8 champion2.changeBuilder(&rangedBuilder);
9 Champion rangedChampion = champion2.build();
```

## 实现的改进

### 1. 清晰的近战和远程英雄区分

- 通过引入 MeleeBuilder 和 RangedBuilder，我们将近战和远程英雄的构建逻辑分离到不同的生成器类中。每个生成器类专注于构建特定类型的英雄，使得代码结构更加清晰。

## 2. 提高代码的可维护性

生成器模式将英雄的构建过程分解为多个步骤（如 `buildBasicAttributes` 和 `buildSpecialAttributes`），并将属性的设置集中到生成器类中。新增或修改属性时，只需在相应的生成器类中进行调整，而不需要修改全局的 `CHAMPION_ATTR_MAP`。

## 3. 扩展性增强

生成器模式将属性的构建过程抽象为接口方法（如 `buildBasicAttributes` 和 `buildSpecialAttributes`），未来如果需要新增属性，只需在生成器类中添加相应的构建步骤，而不需要修改 `Champion` 类的构造函数或其他相关代码。

## 4. 降低初始化复杂度

生成器模式将复杂的初始化过程分解为多个简单的步骤（如 `buildBasicAttributes` 和 `buildSpecialAttributes`），并通过 `getResult` 方法返回构建完成的 `Champion` 对象。这使得初始化逻辑更加模块化和易于理解。

## 5. 灵活性提高

- 生成器模式允许我们在运行时动态切换生成器（通过 `changeBuilder` 方法），从而灵活地构建不同类型的英雄。例如，可以根据游戏进度或玩家选择动态调整英雄的属性。

## 6. 减少属性冗余

- 生成器模式允许我们根据英雄类型（近战或远程）动态设置属性，避免了不必要的属性冗余。例如，`MeleeBuilder` 只设置近战英雄所需的属性，而 `RangedBuilder` 只设置远程英雄所需的属性。

# 使用单例模式（Singleton Pattern）重构

## 原有问题

在重构前的代码中，`LocationMap` 类的设计存在以下问题：

### 1. 多实例问题

`LocationMap` 类可以被多次实例化，这会导致系统中存在多个 `LocationMap` 对象，每个对象都包含一份相同的位置属性与屏幕坐标的键值对。这不仅浪费内存资源，还可能导致数据不一致的问题。

```
1 LocationMap::LocationMap() {
2     initializeLocationMap();
3 }
```

### 2. 全局访问问题

由于 `LocationMap` 类可能被多次实例化，其他模块在使用 `LocationMap` 时，需要明确知道使用哪个实例，这增加了代码的复杂性和耦合度。

```
1 const std::map<Location, cocos2d::Vec2>& LocationMap::getLocationMap() const {  
2     return locationMap;  
3 }
```

### 3. 初始化问题

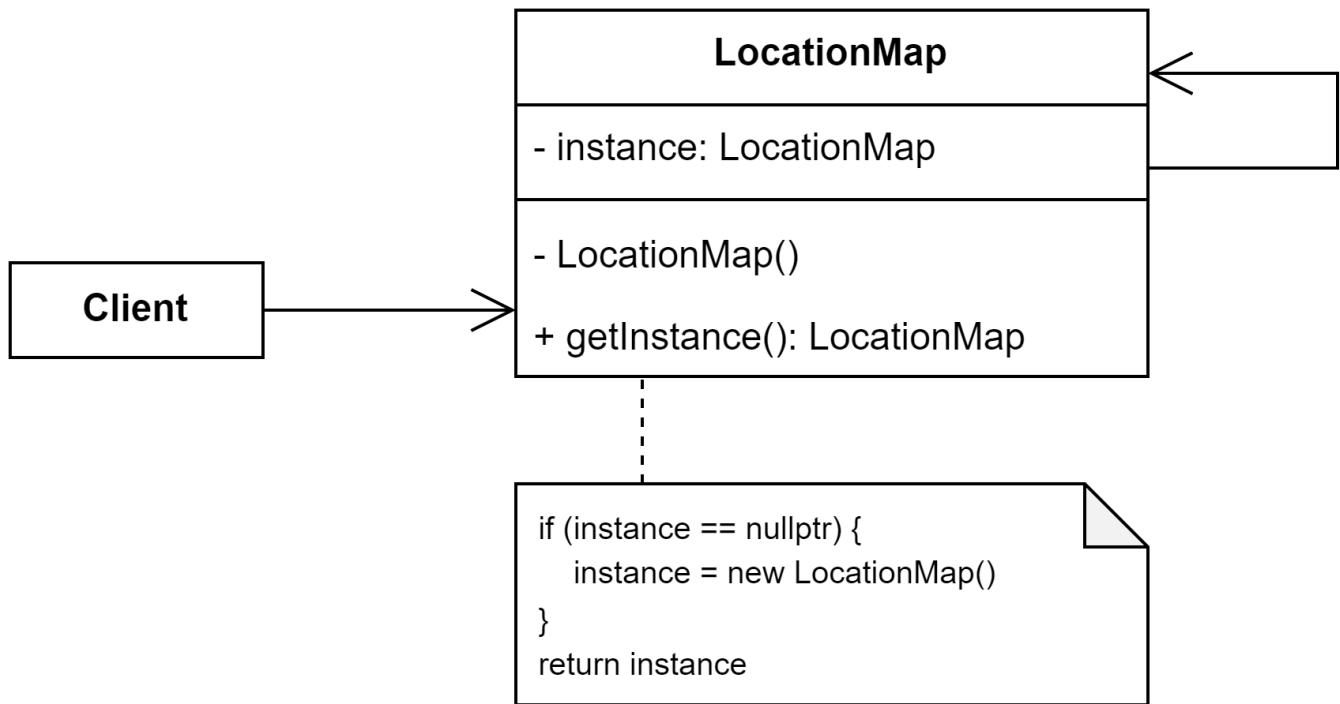
每次创建 `LocationMap` 对象时，都会调用 `initializeLocationMap` 方法进行初始化，这会导致重复的初始化操作，增加了不必要的性能开销。

```
1 void LocationMap::initializeLocationMap() {  
2     locationMap.clear(); // 清空现有的Map  
3  
4     // 初始化等待区域的位置  
5     for (int i = 0; i < WAITING_MAP_COUNT; i++) {  
6         const Location location = { WaitingArea, i };  
7         locationMap[location] = cocos2d::Vec2(WAITING_AREA_START_X + i *  
8             CHAMPION_HORIZONTAL_INTERVAL, WAITING_AREA_START_Y);  
9     }  
10    // 初始化战斗区域的位置  
11    for (int i = 0; i < BATTLE_MAP_ROWS; i++) {  
12        for (int j = 0; j < BATTLE_MAP_COLUMNS; j++) {  
13            const Location location = { BattleArea, i * BATTLE_MAP_COLUMNS + j  
};  
14            locationMap[location] = cocos2d::Vec2(BATTLE_AREA_START_X + j *  
15                CHAMPION_HORIZONTAL_INTERVAL - ((i % 2 == 0) ? 0 :  
16                CHAMPION_HORIZONTAL_INTERVAL / 2), BATTLE_AREA_START_Y + i *  
17                CHAMPION_VERTICAL_INTERVAL);  
18        }  
19    }  
20}
```

### 重构必要性

在重构前的代码中，`LocationMap` 类的设计存在多实例问题、全局访问问题和初始化问题。具体来说，`LocationMap` 类可以被多次实例化，导致系统中存在多个对象，浪费内存资源并可能引发数据不一致；其他模块在使用 `LocationMap` 时需要明确知道使用哪个实例，增加了代码的复杂性和耦合度；每次创建 `LocationMap` 对象时都会重复调用初始化方法，增加了不必要的性能开销。为了解决这些问题，使用单例模式进行重构是非常必要的。单例模式确保一个类只有一个实例，并提供一个全局访问点，从而避免了多实例问题，简化了全局访问，并优化了初始化逻辑。

## UML 类图



单例模式 UML 类图

## 重构步骤

为了解决上述问题，我们使用单例模式对 **LocationMap** 类进行重构。单例模式确保一个类只有一个实例，并提供一个全局访问点。以下是重构的具体步骤：

### 1. 私有化构造函数

将 **LocationMap** 的构造函数设为私有，防止外部代码直接创建 **LocationMap** 对象。

```
1 class LocationMap {  
2     public:  
3         ...  
4  
5     private:  
6         std::map<Location, cocos2d::Vec2> locationMap; // 位置属性与屏幕坐标键值对  
7  
8         // 构造函数  
9         LocationMap();  
10  
11        ...  
12 };
```

### 2. 删除拷贝构造函数和赋值操作符

为了防止通过拷贝构造函数或赋值操作符创建新的实例，我们将这两个操作符设为 `delete`。

```
1 class LocationMap {
2 public:
3     ...
4
5 private:
6     ...
7
8     // 禁止拷贝构造函数和赋值操作符
9     LocationMap(const LocationMap&) = delete;
10    LocationMap& operator=(const LocationMap&) = delete;
11};
```

### 3. 提供静态获取实例的方法

添加一个静态方法 `getInstance`，用于获取 `LocationMap` 的唯一实例。该方法内部使用静态局部变量确保只创建一个实例。

```
1 class LocationMap {
2 public:
3     // 获取单例
4     static LocationMap& getInstance();
5
6     // 获取位置属性与屏幕坐标键值对
7     const std::map<Location, cocos2d::Vec2>& getLocationMap() const;
8
9 private:
10    ...
11};
```

### 4. 简化初始化逻辑

将 `initializeLocationMap` 方法的逻辑直接放入构造函数中，因为单例模式确保了构造函数只会被调用一次。

```
1 LocationMap::LocationMap()
2 {
3     for (int i = 0; i < WAITING_MAP_COUNT; i++) {
4         const Location location = { WaitingArea, i };
5         locationMap[location] = cocos2d::Vec2(WAITING_AREA_START_X + i *
6             CHAMPION_HORIZONTAL_INTERVAL, WAITING_AREA_START_Y);
7     }
8     for (int i = 0; i < BATTLE_MAP_ROWS; i++) {
9         for (int j = 0; j < BATTLE_MAP_COLUMNS; j++) {
```

```

9         const Location location = { BattleArea, i * BATTLE_MAP_COLUMNS + j
10        };
10        locationMap[location] = cocos2d::Vec2(BATTLE_AREA_START_X + j *
11          CHAMPION_HORIZONTAL_INTERVAL - ((i % 2 == 0) ? 0 :
12          CHAMPION_HORIZONTAL_INTERVAL / 2), BATTLE_AREA_START_Y + i *
13          CHAMPION_VERTICAL_INTERVAL);
11      }
12  }
13 }

```

## 5. 移除不必要的成员函数

由于单例模式确保了只有一个实例，因此不再需要 `initializeLocationMap` 方法，可以直接在构造函数中完成初始化。

### 实现的改进

#### 1. 单实例管理

- 通过单例模式确保 `LocationMap` 类只有一个实例，避免了多实例问题，节省了内存资源，并消除了数据不一致的风险。

#### 2. 全局访问简化

- 提供静态方法 `getInstance`，使得其他模块可以方便地获取 `LocationMap` 的唯一实例，降低了代码的复杂性和耦合度。

#### 3. 初始化优化

- 将初始化逻辑直接放入构造函数中，利用单例模式的特性确保初始化只会执行一次，避免了重复的初始化操作，提升了性能。

#### 4. 防止非法实例化

- 将构造函数设为私有，并删除拷贝构造函数和赋值操作符，防止外部代码通过拷贝或赋值创建新的实例，增强了代码的安全性。

#### 5. 代码简洁性提升

- 移除了不必要的 `initializeLocationMap` 方法，简化了类的结构，使代码更加清晰和易于维护。

#### 6. 可测试性增强

- 单例模式使得 `LocationMap` 的行为更加可预测，便于单元测试和调试。

## 使用结构型模式（Structural Patterns）重构

## 使用组合模式（Composite Pattern）重构

## 原有问题

在原有的代码中，以 `ChampionAttributesLayer` 类为例，这个类直接创建并管理多个 `Label` 对象来显示英雄属性。这种方式的问题在于：

### 1. 代码重复

每次创建标签时都需要调用 `createLabel` 方法，代码重复且不易维护。

```
1 void ChampionAttributesLayer::createLabel(const std::string& text, const float
2     x, const float y, const int fontSize, const cocos2d::Color4B color)
3 {
4     auto label = Label::createWithTTF(text,
5         "./Resources/Fonts/DingDingJinBuTi.ttf", fontSize);
6     label->setAnchorPoint(Vec2(0, 0.5));
7     label->setPosition(x, y);
8     label->setTextColor(color);
9     this->addChild(label);
10 }
```

### 2. 缺乏灵活性

如果未来需要添加其他类型的节点（如 `Sprite`、`Button` 等），代码需要大量修改。

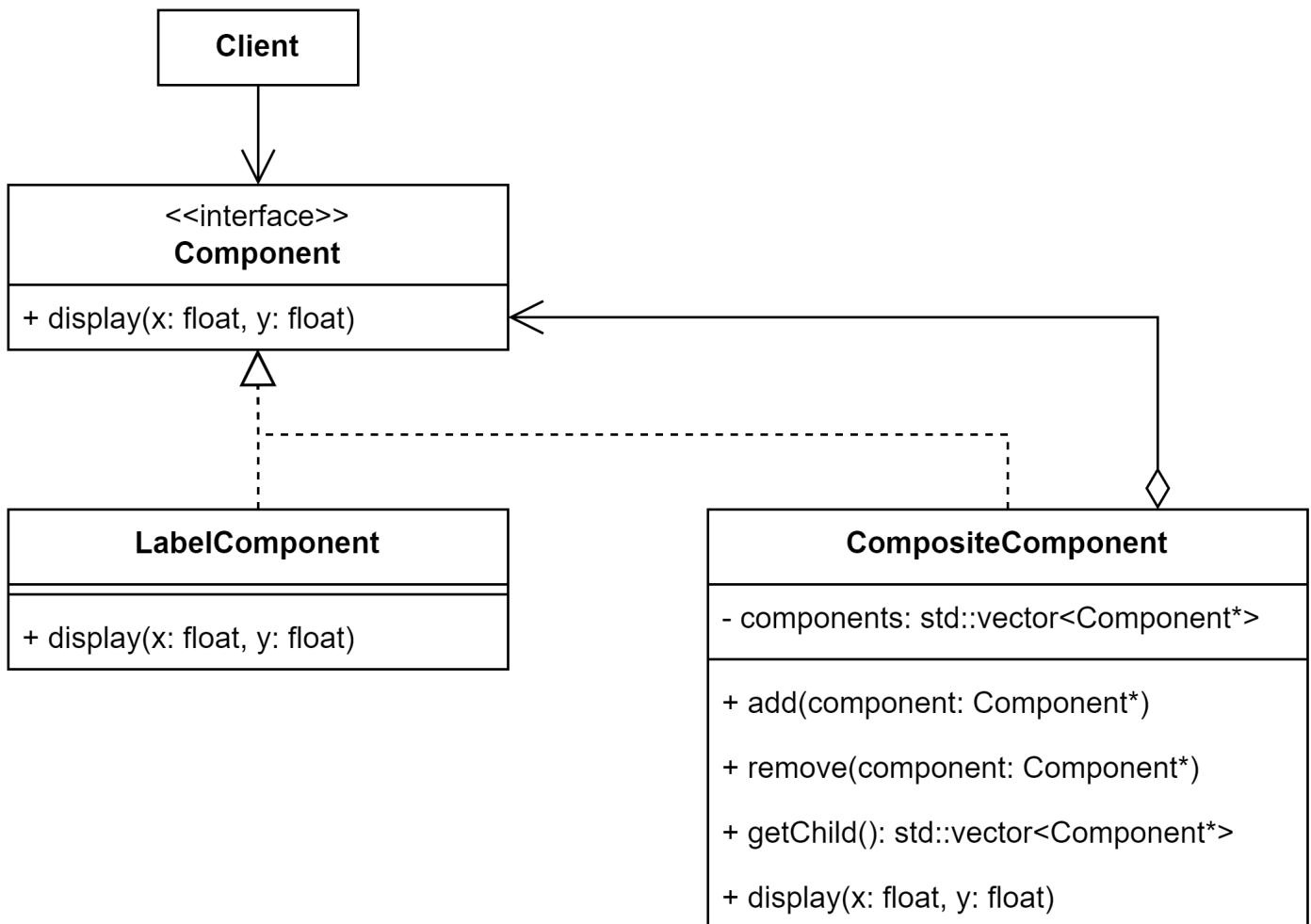
### 3. 紧耦合

`ChampionAttributesLayer` 类与 `Label` 类紧密耦合，不利于扩展和重用。

## 重构必要性

在原有的代码中，`ChampionAttributesLayer` 类直接创建并管理多个 `Label` 对象来显示英雄属性，导致代码重复、缺乏灵活性以及与 `Label` 类的紧耦合。具体来说，每次创建标签时都需要调用 `createLabel` 方法，代码重复且不易维护；未来如果需要添加其他类型的节点（如 `Sprite`、`Button` 等），代码需要大量修改；`ChampionAttributesLayer` 类与 `Label` 类紧密耦合，不利于扩展和重用。为了解决这些问题，使用组合模式进行重构是非常必要的。组合模式将对象组织成树形结构，使得客户端可以统一处理单个对象和组合对象，从而提高了代码的灵活性、可扩展性和可维护性。

## UML 类图



组合模式 UML 类图

## 重构步骤

通过使用组合模式，我们将 `ChampionAttributesLayer` 类与具体的节点类型（如 `Label`）解耦，使得代码更加灵活和可扩展。未来如果需要添加其他类型的节点（如 `Sprite`、`Button` 等），只需创建新的叶子节点类并实现 `Component` 接口即可，无需修改

`ChampionAttributesLayer` 类的代码。下面是使用组合模式重构

`ChampionAttributesLayer` 类的步骤：

### 1. 定义一个组件接口

所有节点（如 `Label`、`Sprite` 等）都实现这个接口。

```

1 class Component {
2 public:
3     virtual ~Component() {}
4     virtual void add(Component* component) {}
5     virtual void remove(Component* component) {}
6     virtual void display(float x, float y) = 0;
7 };
  
```

## 2. 创建叶子节点

叶子节点是没有子节点的节点，如 `Label`。

```
1 class LabelComponent : public Component {  
2 public:  
3     static LabelComponent* create(const std::string& text, int fontSize,  
        cocos2d::Color4B color);  
4     virtual void display(float x, float y) override;  
5  
6 private:  
7     LabelComponent(const std::string& text, int fontSize, cocos2d::Color4B  
        color);  
8     cocos2d::Label* label;  
9 };
```

```
1 LabelComponent::LabelComponent(const std::string& text, int fontSize,  
    cocos2d::Color4B color) {  
2     label = cocos2d::Label::createWithTTF(text,  
    "../Resources/Fonts/DingDingJinBuTi.ttf", fontSize);  
3     label->setTextColor(color);  
4 }  
5  
6 LabelComponent* LabelComponent::create(const std::string& text, int fontSize,  
    cocos2d::Color4B color) {  
7     return new LabelComponent(text, fontSize, color);  
8 }  
9  
10 void LabelComponent::display(float x, float y) {  
11     label->setPosition(x, y);  
12     label->setAnchorPoint(cocos2d::Vec2(0, 0.5));  
13     cocos2d::Director::getInstance()->getRunningScene()->addChild(label);  
14 }
```

## 3. 创建复合节点

复合节点可以包含其他节点（叶子节点或其他复合节点）。

```
1 class CompositeComponent : public Component {  
2 public:  
3     virtual void add(Component* component) override;  
4     virtual void remove(Component* component) override;  
5     virtual void display(float x, float y) override;
```

```
6     std::vector<Component*> getChild() const;
7
8 private:
9     std::vector<Component*> components;
10 }
```

```
1 void CompositeComponent::add(Component* component) {
2     components.push_back(component);
3 }
4
5 void CompositeComponent::remove(Component* component) {
6     components.erase(std::remove(components.begin(), components.end(),
7         component), components.end());
8 }
9
10 void CompositeComponent::display(float x, float y) {
11     for (auto& component : components) {
12         component->display(x, y);
13     }
14 }
15 std::vector<Component*> CompositeComponent::getChild() const {
16     return components;
17 }
```

#### 4. 重构 ChampionAttributesLayer 类

使用组合模式来管理节点。

```
1 class ChampionAttributesLayer : public cocos2d::Layer {
2 public:
3     virtual bool init();
4     void showAttributes(const ChampionCategory championCategory);
5     CREATE_FUNC(ChampionAttributesLayer);
6
7 private:
8     void createLabel(const std::string& text, float x, float y, int fontSize =
9         CHAMPION_ATTRIBUTES_FONT_SIZE, cocos2d::Color4B color =
10        cocos2d::Color4B::WHITE);
11     std::string formatFloat(float value, int precision = 2);
12     CompositeComponent* rootComponent;
13 };
```

```
1 // 初始化战斗英雄属性层
2 bool ChampionAttributesLayer::init() {
3     if (!Layer::init()) {
4         return false;
5     }
6
7     rootComponent = new CompositeComponent();
8
9     auto backgroundImage =
cocos2d::Sprite::create("../Resources/ImageElements/ChampionAttributesLayerBack
ground.png");
10    backgroundImage->setPosition(BACKGROUND_IMAGE_START_X,
BACKGROUND_IMAGE_START_Y);
11    backgroundImage->setOpacity(BACKGROUND_IMAGE_TRANSPARENCY);
12    this->addChild(backgroundImage);
13
14    return true;
15 }
16
17 // 显示战斗英雄属性
18 void ChampionAttributesLayer::showAttributes(const ChampionCategory
championCategory) {
19     auto championAttributes = CHAMPION_ATTR_MAP.at(championCategory);
20
21     int championNumber = championAttributes.championCategory % 2 == 1 ?
championAttributes.championCategory : championAttributes.championCategory - 1;
22     std::string championImagePath = "../Resources/Champions/Champion" +
std::to_string(championNumber) + "&" + std::to_string(championNumber + 1) +
".png";
23     auto championImage = cocos2d::Sprite::create(championImagePath);
24     championImage->setPosition(CHAMPION_IMAGE_START_X, CHAMPION_IMAGE_START_Y);
25     this->addChild(championImage);
26
27     createLabel(championAttributes.championName, CHAMPION_NAME_LABEL_START_X,
CHAMPION_NAME_LABEL_START_Y, CHAMPION_NAME_LABEL_FONT_SIZE,
championAttributes.championCategory % 2 == 1 ? cocos2d::Color4B::WHITE :
cocos2d::Color4B({ GOLDEN_R, GOLDEN_G, GOLDEN_B }));
28     createLabel(std::to_string(championAttributes.level), LEVEL_LABEL_START_X,
LEVEL_LABEL_START_Y);
29     createLabel(std::to_string(championAttributes.healthPoints),
FIRST_COLUMN_START_X, HEALTH_POINTS_LABEL_START_Y);
30     createLabel(std::to_string(championAttributes.attackDamage),
FIRST_COLUMN_START_X, ATTACK_DAMAGE_LABEL_START_Y);
31     createLabel(std::to_string(championAttributes.attackRange),
FIRST_COLUMN_START_X, ATTACK_RANGE_LABEL_START_Y);
32     createLabel(formatFloat(championAttributes.attackSpeed),
FIRST_COLUMN_START_X, ATTACK_SPEED_LABEL_START_Y);
```

```

33     createLabel(std::to_string(championAttributes.price),
34     SECOND_COLUMN_START_X, PRICE_LABEL_START_Y);
35     createLabel(formatFloat(championAttributes.movementSpeed),
36     SECOND_COLUMN_START_X, MOVEMENT_SPEED_START_Y);
37     createLabel(formatFloat(championAttributes.defenseCoefficient),
38     SECOND_COLUMN_START_X, DEFENSE_COEFFICIENT_START_Y);
39     createLabel(std::to_string(championAttributes.skillTriggerThreshold),
40     SECOND_COLUMN_START_X, SKILL_TRIGGER_THRESHOLD_START_Y);
41 }
42 // 创建属性标签
43 void ChampionAttributesLayer::createLabel(const std::string& text, float x,
44 float y, int fontSize, cocos2d::Color4B color) {
45     auto labelComponent = LabelComponent::create(text, fontSize, color);
46     rootComponent->add(labelComponent);
47     labelComponent->display(x, y);
48 }
49 // 浮点数格式化输出
50 std::string ChampionAttributesLayer::formatFloat(const float value, const int
precision)
51 {
52     std::ostringstream oss;
53     oss << std::fixed << std::setprecision(precision) << value;
54     return oss.str();

```

## 实现的改进

### 1. 解耦与抽象

- 通过定义 `Component` 接口，将 `ChampionAttributesLayer` 类与具体的节点类型（如 `Label`、`Sprite` 等）解耦，使得代码更加灵活和可扩展。

### 2. 减少代码重复

- 使用组合模式后，创建和管理节点的逻辑被封装在 `Component` 接口及其实现类中，避免了重複调用 `createLabel` 方法，减少了代码冗余。

### 3. 支持多种节点类型

- 通过组合模式，可以轻松添加新的节点类型（如 `SpriteComponent`、`ButtonComponent` 等），只需实现 `Component` 接口即可，无需修改

### 4. 统一管理节点

- 使用 `CompositeComponent` 类统一管理所有子节点（包括叶子节点和其他复合节点），简化了节点的添加、删除和显示操作。

## 5. 增强扩展性

- 未来如果需要添加新的节点类型或修改现有节点的行为，只需扩展 `Component` 接口及其实现类，而无需修改 `ChampionAttributesLayer` 类的代码。

## 6. 提高代码可读性

- 组合模式将复杂的节点管理逻辑分解为多个简单的类（如 `LabelComponent`、`CompositeComponent` 等），使得代码结构更加清晰，易于理解和维护。

## 7. 支持动态结构

- 组合模式允许在运行时动态地添加或移除节点，使得 `ChampionAttributesLayer` 类可以灵活地适应不同的需求。

## 8. 简化客户端代码

- `ChampionAttributesLayer` 类只需与 `Component` 接口交互，无需关心具体的节点类型和实现细节，简化了客户端代码。

# 使用外观模式（Facade Pattern）重构

## 原有问题

在原始的代码中，音频播放的逻辑直接暴露在全局函数和全局变量中，存在以下问题：

### 1. 高耦合性

外部代码直接依赖 `AudioEngine` 和全局变量，导致代码耦合度高，难以维护和扩展。

### 2. 代码重复

如果多个地方需要播放音频，可能会重复编写类似的代码（如停止当前音乐、设置音量等）。

### 3. 难以扩展

如果需要增加新的功能（如暂停、恢复、音量控制等），需要在多个地方修改代码。

```
1 void audioPlayer(const std::string& audioPath, bool isLoop)
2 {
3     if (isLoop) {
4         if (g_backgroundMusicSign != DEFAULT_MUSIC_SIGN) {
5             cocos2d::experimental::AudioEngine::stop(g_backgroundMusicSign);
6         }
7         g_backgroundMusicSign =
8             cocos2d::experimental::AudioEngine::play2d(audioPath, isLoop);
9         cocos2d::experimental::AudioEngine::setVolume(g_backgroundMusicSign,
10             g_backgroundMusicVolume);
```

```
9     }
10    else {
11        g_soundEffectSign =
12            cocos2d::experimental::AudioEngine::play2d(audioPath, isLoop);
13        cocos2d::experimental::AudioEngine::setVolume(g_soundEffectSign,
14            g_soundEffectVolume);
15    }
16 }
```

## 4. 全局变量风险

使用全局变量存储音频状态（如 `g_backgroundMusicSign`）可能导致意外的修改或冲突。

```
1 int g_backgroundMusicSign = DEFAULT_MUSIC_SIGN;
2 int g_soundEffectSign = DEFAULT_MUSIC_SIGN;
3 float g_backgroundMusicVolume = DEFAULT_MUSIC_VOLUME;
4 float g_soundEffectVolume = DEFAULT_MUSIC_VOLUME;
```

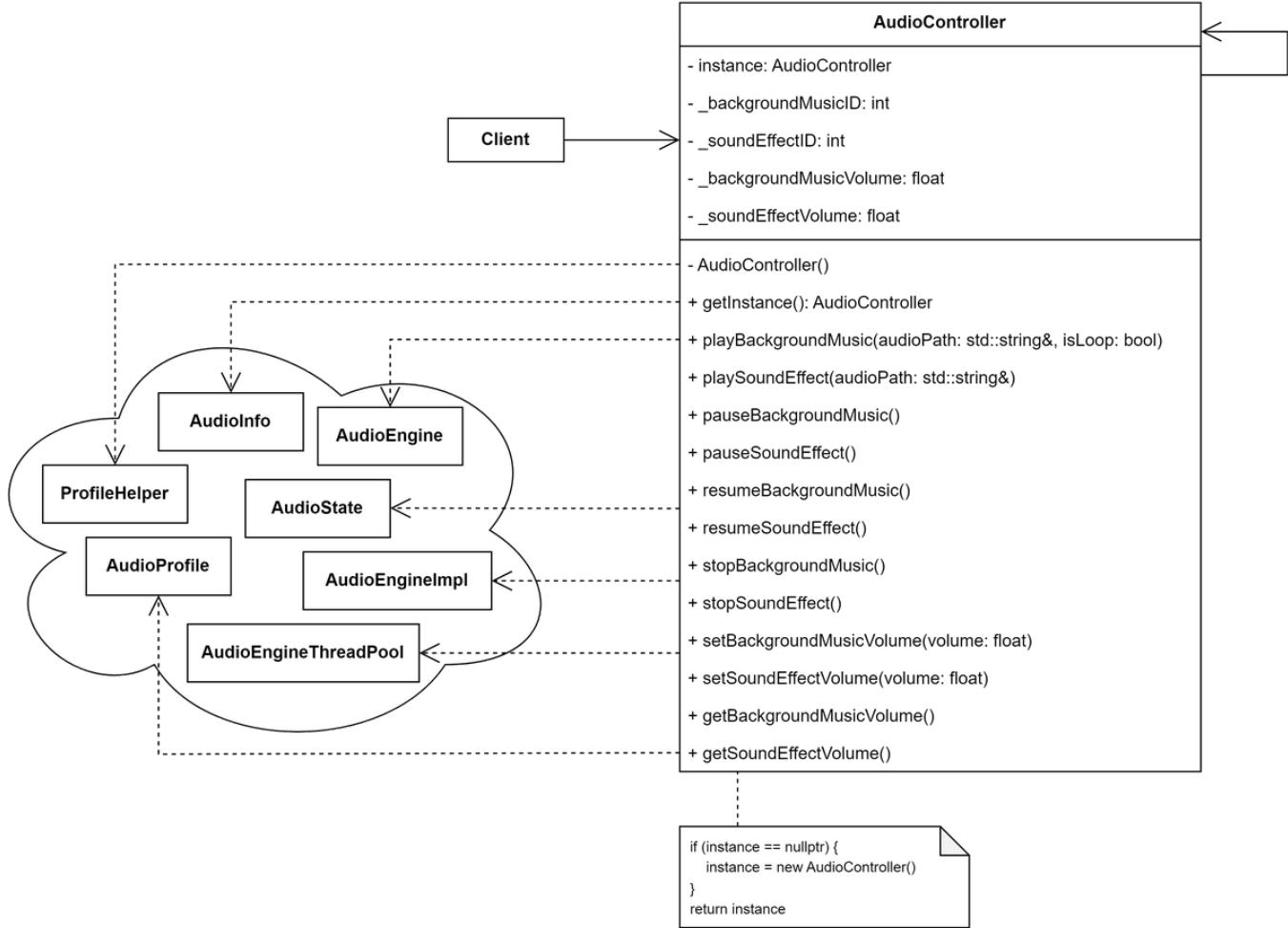
## 5. 缺乏封装

音频播放的逻辑分散在全局函数中，缺乏统一的接口，不利于代码的组织和复用。

### 重构必要性

在原始的代码中，音频播放的逻辑直接暴露在全局函数和全局变量中，导致代码存在高耦合性、重复性、难以扩展以及全局变量风险等问题。具体来说，外部代码直接依赖 `AudioEngine` 和全局变量，导致代码耦合度高，难以维护和扩展；多个地方需要播放音频时可能会重复编写类似的代码；新增功能（如暂停、恢复、音量控制等）需要在多个地方修改代码；全局变量的使用可能导致意外的修改或冲突；音频播放的逻辑分散在全局函数中，缺乏统一的接口，不利于代码的组织和复用。为了解决这些问题，使用外观模式进行重构是非常必要的。外观模式通过提供一个统一的接口封装复杂的子系统，从而降低了代码的耦合度，提高了可维护性和可扩展性。

### UML 类图



外观模式 UML 类图

## 重构步骤

为了解决上述问题，我们可以通过外观模式对音频播放逻辑进行封装，提供一个统一的接口供外部调用。以下是具体的重构步骤：

### 1. 分析原始代码的功能

- 原始代码的功能包括：

- 播放背景音乐（循环）
- 播放音效（非循环）
- 控制音量
- 停止当前播放的音乐

- 需要扩展的功能：

- 暂停、恢复音频
- 获取当前音量
- 更灵活的音量控制

### 2. 设计外观类（**AudioController**）并定义外观类的接口

- 将音频播放的逻辑封装到一个类中。
- 提供统一的接口，隐藏底层 `AudioEngine` 的复杂性。
- 使用单例模式确保全局只有一个 `AudioController` 实例。
- 定义外观类的接口，提供以下方法：
  - 播放背景音乐
  - 播放音效
  - 暂停、恢复、停止音频
  - 设置和获取音量

```
1 class AudioController {  
2     public:  
3         // 单例模式  
4         static AudioController* getInstance();  
5  
6         // 播放音频  
7         void playBackgroundMusic(const std::string& audioPath, bool isLoop = true);  
8         void playSoundEffect(const std::string& audioPath);  
9  
10        // 暂停音频  
11        void pauseBackgroundMusic();  
12        void pauseSoundEffect();  
13  
14        // 恢复音频  
15        void resumeBackgroundMusic();  
16        void resumeSoundEffect();  
17  
18        // 停止音频  
19        void stopBackgroundMusic();  
20        void stopSoundEffect();  
21  
22        // 设置音量  
23        void setBackgroundMusicVolume(float volume);  
24        void setSoundEffectVolume(float volume);  
25  
26        // 获取音量  
27        float getBackgroundMusicVolume() const;  
28        float getSoundEffectVolume() const;  
29  
30    private:  
31        // 私有构造函数（单例模式）  
32        AudioController();  
33        ~AudioController();
```

```

34
35     // 禁用拷贝构造函数和赋值运算符
36     AudioController(const AudioController&) = delete;
37     AudioController& operator=(const AudioController&) = delete;
38
39     // 背景音乐和音效的ID
40     int _backgroundMusicID;
41     int _soundEffectID;
42
43     // 音量
44     float _backgroundMusicVolume;
45     float _soundEffectVolume;
46 };

```

### 3. 实现外观类

- 在 `AudioController` 类中实现上述接口。
- 使用成员变量存储音频状态（如当前播放的音频ID、音量等）。
- 封装底层 `AudioEngine` 的调用。

```

1 // 单例实例
2 AudioController* AudioController::_instance = nullptr;
3
4 AudioController* AudioController::getInstance() {
5     if (!_instance) {
6         _instance = new AudioController();
7     }
8     return _instance;
9 }
10
11 AudioController::AudioController()
12     : _backgroundMusicID(-1), _soundEffectID(-1),
13       _backgroundMusicVolume(1.0f), _soundEffectVolume(1.0f) {}
14
15 AudioController::~AudioController() {
16     // 释放资源
17     stopBackgroundMusic();
18     stopSoundEffect();
19     delete _instance;
20 }
21
22 // 播放背景音乐
23 void AudioController::playBackgroundMusic(const std::string& audioPath, bool
24     isLoop) {
25     if (_backgroundMusicID != -1) {

```

```
25         cocos2d::experimental::AudioEngine::stop(_backgroundMusicID);
26     }
27     _backgroundMusicID = cocos2d::experimental::AudioEngine::play2d(audioPath,
28     isLoop);
29 }
30
31 // 播放音效
32 void AudioController::playSoundEffect(const std::string& audioPath) {
33     _soundEffectID = cocos2d::experimental::AudioEngine::play2d(audioPath,
34     false);
35     cocos2d::experimental::AudioEngine::setVolume(_soundEffectID,
36     _soundEffectVolume);
37 }
38
39 // 暂停背景音乐
40 void AudioController::pauseBackgroundMusic() {
41     if (_backgroundMusicID != -1) {
42         cocos2d::experimental::AudioEngine::pause(_backgroundMusicID);
43     }
44 }
45
46 // 暂停音效
47 void AudioController::pauseSoundEffect() {
48     if (_soundEffectID != -1) {
49         cocos2d::experimental::AudioEngine::pause(_soundEffectID);
50     }
51 }
52
53 // 恢复背景音乐
54 void AudioController::resumeBackgroundMusic() {
55     if (_backgroundMusicID != -1) {
56         cocos2d::experimental::AudioEngine::resume(_backgroundMusicID);
57     }
58 }
59
60 // 恢复音效
61 void AudioController::resumeSoundEffect() {
62     if (_soundEffectID != -1) {
63         cocos2d::experimental::AudioEngine::resume(_soundEffectID);
64     }
65 }
66
67 // 停止背景音乐
68 void AudioController::stopBackgroundMusic() {
69     if (_backgroundMusicID != -1) {
```

```

68         cocos2d::experimental::AudioEngine::stop(_backgroundMusicID);
69         _backgroundMusicID = -1;
70     }
71 }
72
73 // 停止音效
74 void AudioController::stopSoundEffect() {
75     if (_soundEffectID != -1) {
76         cocos2d::experimental::AudioEngine::stop(_soundEffectID);
77         _soundEffectID = -1;
78     }
79 }
80
81 // 设置背景音乐音量
82 void AudioController::setBackgroundMusicVolume(float volume) {
83     _backgroundMusicVolume = volume;
84     if (_backgroundMusicID != -1) {
85         cocos2d::experimental::AudioEngine::setVolume(_backgroundMusicID,
86             volume);
87     }
88
89 // 设置音效音量
90 void AudioController::setSoundEffectVolume(float volume) {
91     _soundEffectVolume = volume;
92     if (_soundEffectID != -1) {
93         cocos2d::experimental::AudioEngine::setVolume(_soundEffectID, volume);
94     }
95 }
96
97 // 获取背景音乐音量
98 float AudioController::getBackgroundMusicVolume() const {
99     return _backgroundMusicVolume;
100 }
101
102 // 获取音效音量
103 float AudioController::getSoundEffectVolume() const {
104     return _soundEffectVolume;
105 }

```

#### 4. 替换原始代码

- 将原始代码中直接调用 `AudioEngine` 的地方替换为调用 `AudioController` 的接口。
- 删除全局变量和全局函数。

```
1 #include "AudioController.h"
2
3 // 播放背景音乐
4 AudioController::getInstance()-
>playBackgroundMusic("../Resources/Music/PreparationScene_RagsToRings.mp3");
5
6 // 播放音效
7 AudioController::getInstance()-
>playSoundEffect("../Resources/Music/SkillSoundEffect.mp3");
8
9 // 设置背景音乐音量
10 AudioController::getInstance()->setBackgroundMusicVolume(0.5f);
11
12 // 暂停背景音乐
13 AudioController::getInstance()->pauseBackgroundMusic();
14
15 // 停止音效
16 AudioController::getInstance()->stopSoundEffect();
```

## 实现的改进

### 1. 降低耦合度

- 通过 `AudioController` 类封装音频播放的逻辑，外部代码只需与 `AudioController` 交互，而无需直接依赖 `AudioEngine`，降低了代码的耦合度。

### 2. 减少代码重复

- 将音频播放的逻辑集中到 `AudioController` 类中，避免了在多个地方重复编写类似的代码（如停止当前音乐、设置音量等）。

### 3. 增强扩展性

- 新增功能（如暂停、恢复、音量控制等）只需在 `AudioController` 类中添加相应的方法，而无需修改外部代码，提高了代码的可扩展性。

### 4. 消除全局变量风险

- 使用 `AudioController` 类的成员变量存储音频状态（如当前播放的音频ID、音量等），避免了全局变量的使用，减少了意外修改或冲突的风险。

### 5. 提供统一接口

- 通过 `AudioController` 类提供统一的接口，隐藏了底层 `AudioEngine` 的复杂性，使得代码更加模块化和易于维护。

### 6. 支持单例模式

- 使用单例模式确保全局只有一个 `AudioController` 实例，避免了多个实例之间的状态不一致问题。

## 7. 简化客户端代码

- 外部代码只需调用 `AudioController` 的接口即可完成音频播放、暂停、恢复、停止等操作，简化了客户端代码。

## 8. 提高代码可读性

- 将复杂的音频播放逻辑封装到 `AudioController` 类中，使得代码结构更加清晰，易于理解和维护。

# 使用行为模式（Behavioral Patterns）重构

## 使用策略模式（Strategy Pattern）重构

### 原有问题

策略模式是一种行为设计模式，它允许定义一系列算法（策略），并将每个算法封装在独立的类中，使得它们可以互换使用。每个技能是一个独立的算法，策略模式可以将算法封装到独立的类中，避免条件分支。`Champion` 类的 `skill` 方法存在问题如下：

### 1. 条件分支过多

`skill` 函数中包含了大量的 `if-else` 条件分支，用于判断不同英雄的技能效果。每个条件分支处理不同的技能逻辑，导致代码冗长且难以维护。

```
1 void Champion::skill()
2 {
3     if (attributes.price == CHAMPION_ATTR_MAP.at(FIFTH_LEVEL[1]).price) {
4         triggerSkill(ATTACK_DAMAGE_MAGNIFICATION_HIGH);
5     }
6     else if (attributes.price == CHAMPION_ATTR_MAP.at(FOURTH_LEVEL[1]).price) {
7         if (attributes.championCategory == Champion25 ||
8             attributes.championCategory == Champion26) {
9             attributes.attackDamage += SKILL_ATTACK_DAMAGE_UP;
10            attributes.defenseCoefficient += SKILL_DEFENSE_COEFFICIENT_UP;
11        }
12    else {
13        triggerSkill(ATTACK_DAMAGE_MAGNIFICATION_MIDDLE, false);
14    }
15    else if (attributes.price == CHAMPION_ATTR_MAP.at(SECOND_LEVEL[1]).price
16 || attributes.price == CHAMPION_ATTR_MAP.at(THIRD_LEVEL[1]).price) {
17        if (attributes.attackRange > ATTACK_RANGE_THRESHOLD) {
18            attributes.attackDamage += SKILL_ATTACK_DAMAGE_UP;
19            attributes.attackSpeed += SKILL_ATTACK_SPEED_UP;
20        }
21    }
22 }
```

```
20     else if (attributes.defenseCoefficient >=
21         DEFENSE_COEFFICIENT_THRESHOLD_HIGH) {
22         attributes.defenseCoefficient += SKILL_DEFENSE_COEFFICIENT_UP;
23     }
24     else {
25         triggerSkill(ATTACK_DAMAGE_MAGNIFICATION_LOW);
26     }
27     else if (attributes.price == CHAMPION_ATTR_MAP.at(FIRST_LEVEL[1]).price) {
28         if (attributes.attackRange > ATTACK_RANGE_THRESHOLD) {
29             attributes.attackDamage += SKILL_ATTACK_DAMAGE_UP;
30             attributes.attackSpeed += SKILL_ATTACK_SPEED_UP;
31         }
32         else if (attributes.defenseCoefficient >=
33             DEFENSE_COEFFICIENT_THRESHOLD_LOW) {
34             attributes.defenseCoefficient += SKILL_DEFENSE_COEFFICIENT_UP;
35         }
36         else {
37             triggerSkill(ATTACK_DAMAGE_MAGNIFICATION_LOW);
38         }
39     }
40     attributes.magicPoints = 0;
41 }
```

## 2. 违反开闭原则

当需要新增一个技能时，必须修改 `skill` 函数的内部逻辑，违反了开闭原则（对扩展开放，对修改封闭）。

## 3. 职责不单一

`skill` 函数不仅负责技能的执行，还负责技能的选择逻辑，违反了单一职责原则。

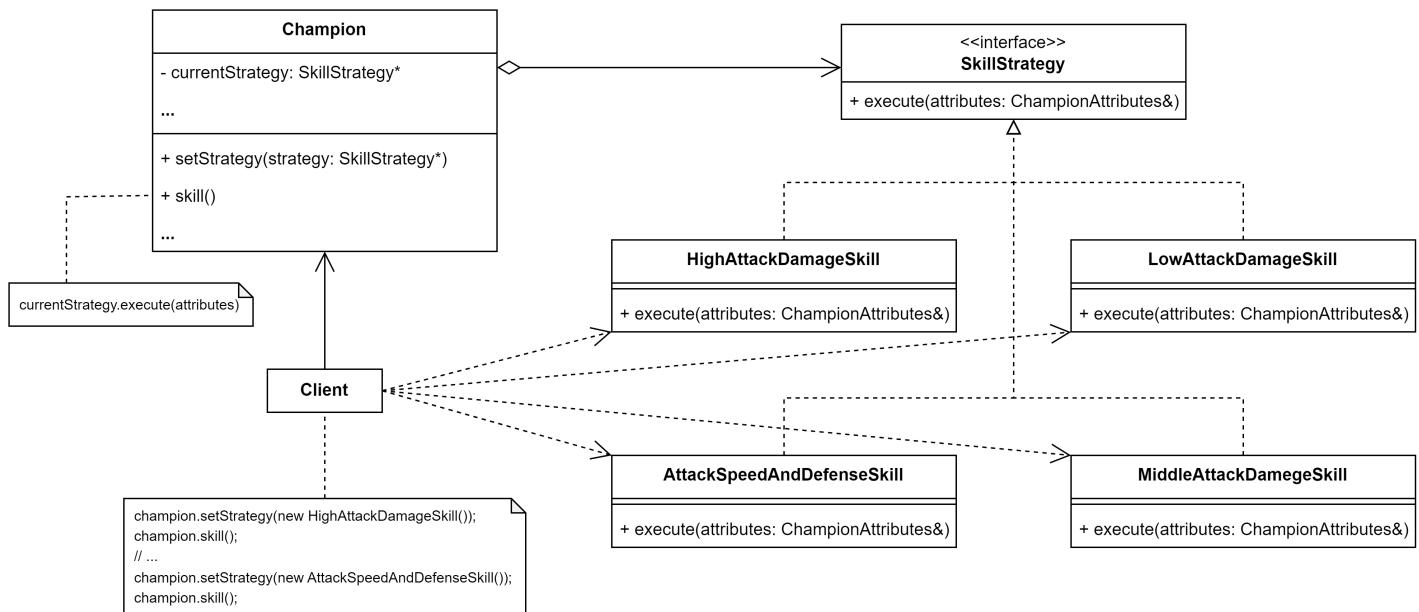
## 4. 扩展性差

由于所有技能逻辑都集中在 `skill` 函数中，新增或修改技能时需要深入理解整个函数的逻辑，增加了出错的风险。

## 重构必要性

在原有的代码中，`Champion` 类的 `skill` 方法存在条件分支过多、违反开闭原则、职责不单一以及扩展性差等问题。具体来说，`skill` 函数中包含了大量的 `if-else` 条件分支，用于判断不同英雄的技能效果，导致代码冗长且难以维护；新增技能时需要修改 `skill` 函数的内部逻辑，违反了开闭原则；`skill` 函数不仅负责技能的执行，还负责技能的选择逻辑，违反了单一职责原则；由于所有技能逻辑都集中在 `skill` 函数中，新增或修改技能时需要深入理解整个函数的逻辑，增加了出错的风险。为了解决这些问题，使用策略模式进行重构是非常必要的。策略模式将每个技能的逻辑封装到独立的类中，使得它们可以互换使用，从而提高了代码的可维护性、可扩展性和可读性。

## UML 类图



策略模式 UML 类图

## 重构步骤

策略模式允许我们定义一系列算法，并将每个算法封装在独立的类中，使得它们可以互换使用。我们可以将每个技能的逻辑封装到一个独立的策略类中，然后在 `skill` 函数中根据不同的条件选择合适的策略。

### 1. 定义策略接口

定义一个 `SkillStrategy` 接口，包含一个 `execute` 方法。

```
1 class SkillStrategy {
2 public:
3     virtual void execute(ChampionAttributes& attributes) = 0;
4     virtual ~SkillStrategy() = default;
5 };
```

### 2. 实现具体策略类

为每个技能实现一个具体的策略类，实现 `execute` 方法。

```
1 class AttackSpeedAndDefenseSkill : public SkillStrategy {
2 public:
3     void execute(ChampionAttributes& attributes) override {
4         attributes.attackSpeed += SKILL_ATTACK_SPEED_UP;
5         attributes.defenseCoefficient += SKILL_DEFENSE_COEFFICIENT_UP;
6     }
7 };
```

```

8
9 class HighAttackDamageSkill : public SkillStrategy {
10 public:
11     void execute(ChampionAttributes& attributes) override {
12         attributes.attackDamage *= ATTACK_DAMAGE_MAGNIFICATION_HIGH;
13     }
14 };
15
16
17 class LowAttackDamageSkill : public SkillStrategy {
18 public:
19     void execute(ChampionAttributes& attributes) override {
20         attributes.attackDamage *= ATTACK_DAMAGE_MAGNIFICATION_LOW;
21     }
22 };
23
24 class MiddleAttackDamageSkill : public SkillStrategy {
25 public:
26     void execute(ChampionAttributes& attributes) override {
27         attributes.attackDamage *= ATTACK_DAMAGE_MAGNIFICATION_MIDDLE;
28     }
29 };

```

### 3. 添加 `setStrategy` 函数并重构 `skill` 函数

在 `Champion` 类中添加私有变量 `currentStrategy`，之后在 `skill` 函数中根据条件选择合适的策略并执行。

```

1 void Champion::setStrategy(SkillStrategy* strategy) {
2     if (currentStrategy) {
3         delete currentStrategy;
4     }
5     currentStrategy = strategy;
6 }
7
8 void Champion::skill() {
9     if (currentStrategy) {
10         currentStrategy->execute(attributes);
11     }
12
13     attributes.magicPoints = 0;
14 }

```

### 实现的改进

## 1. 消除条件分支

- 通过将每个技能的逻辑封装到独立的策略类中，消除了 `skill` 函数中的大量 `if-else` 条件分支，使得代码更加简洁和易于维护。

## 2. 遵循开闭原则

- 新增技能时只需添加一个新的策略类，而无需修改 `skill` 函数的内部逻辑，符合开闭原则（对扩展开放，对修改封闭）。

## 3. 单一职责原则

- `skill` 函数只负责选择和执行策略，而具体的技能逻辑由各个策略类负责，符合单一职责原则。

## 4. 提高扩展性

- 新增或修改技能时只需添加或修改相应的策略类，而无需深入理解 `skill` 函数的逻辑，降低了出错的风险。

## 5. 代码复用

- 将通用的技能逻辑封装到基类或独立的策略类中，提高了代码的复用性。

## 6. 增强可读性

- 每个策略类的职责明确，代码结构更加清晰，易于理解和维护。

## 7. 动态切换策略

- 可以在运行时动态切换技能策略，使得 `Champion` 类的行为更加灵活。

# 使用状态模式（State Pattern）重构

## 原有问题

状态模式是一种行为设计模式，它允许对象在其内部状态改变时改变其行为。每个羁绊效果是一个独立的状态，状态模式可以将状态逻辑封装到独立的类中，避免条件分支。`Champion` 类的 `bond` 方法存在问题如下：

### 1. 条件分支过多

`bond` 函数中使用了 `switch-case` 语句，根据不同的羁绊类型执行不同的逻辑。每个 `case` 分支处理不同的羁绊效果，导致代码冗长且难以维护。

```
1 void Champion::bond()
2 {
3     switch (attributes.bond) {
4         case Brotherhood:
5             attributes.movementSpeed *= BROTHERHOOD_MOVEMENT_SPEED_MULTIPLIER;
6             attributes.attackSpeed *= BROTHERHOOD_ATTACK_SPEED_MULTIPLIER;
7             break;
```

```
8     case Lout:
9         attributes.healthPoints = static_cast<int>(attributes.healthPoints
10        * LOUT_HEALTH_POINTS_MULTIPLIER);
11        attributes.movementSpeed *= LOUT_MOVEMENT_SPEED_MULTIPLIER;
12        attributes.attackDamage *= LOUT_ATTACK_DAMAGE_MULTIPLIER;
13        break;
14    case DarkSide:
15        attributes.skillTriggerThreshold = static_cast<int>
16        (attributes.skillTriggerThreshold * DARKSIDE_SKILL_TRIGGER_MULTIPLIER);
17        attributes.attackDamage *= DARKSIDE_ATTACK_DAMAGE_MULTIPLIER;
18        break;
19    case GoodShooter:
20        attributes.attackSpeed *= GOODSHOOTER_ATTACK_SPEED_MULTIPLIER;
21        break;
22    case PopStar:
23        attributes.attackSpeed *= POPSTAR_ATTACK_SPEED_MULTIPLIER;
24        attributes.movementSpeed *= POPSTAR_MOVEMENT_SPEED_MULTIPLIER;
25        break;
26    default:
27        break;
28    }
29 }
```

## 2. 违反开闭原则

当需要新增一个羁绊效果时，必须修改 `bond` 函数的内部逻辑，违反了开闭原则。

## 3. 职责不单一

`bond` 函数不仅负责羁绊效果的应用，还负责羁绊类型的选择逻辑，违反了单一职责原则。

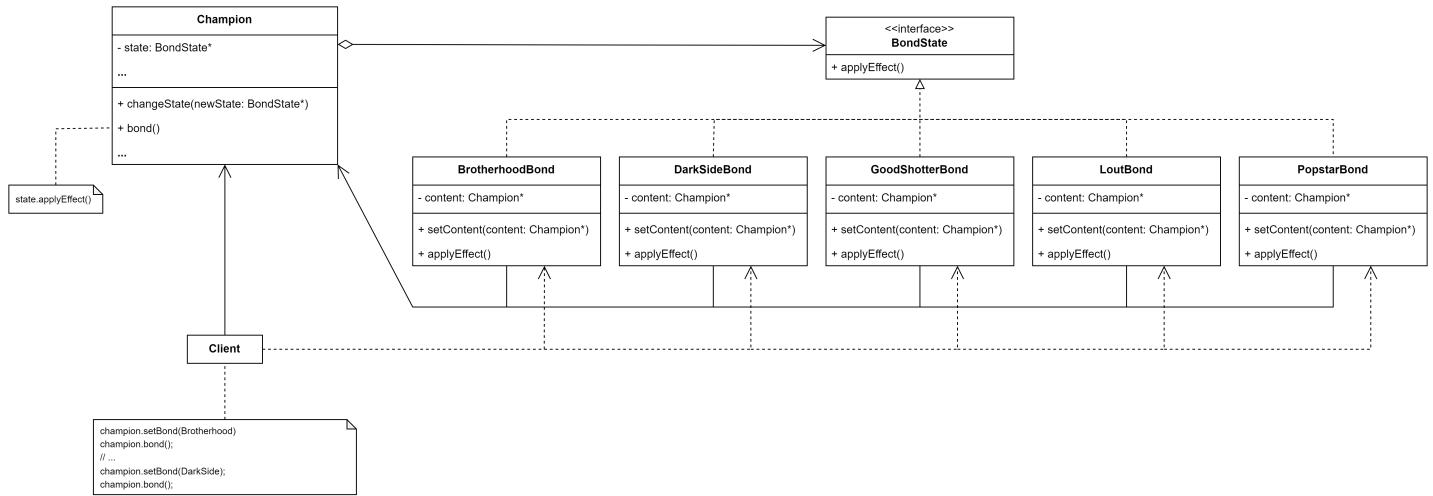
## 4. 扩展性差

由于所有羁绊逻辑都集中在 `bond` 函数中，新增或修改羁绊时需要深入理解整个函数的逻辑，增加了出错的风险。

## 重构必要性

在原有的代码中，`Champion` 类的 `bond` 方法存在条件分支过多、违反开闭原则、职责不单一以及扩展性差等问题。具体来说，`bond` 函数中使用了 `switch-case` 语句，根据不同的羁绊类型执行不同的逻辑，导致代码冗长且难以维护；新增羁绊效果时需要修改 `bond` 函数的内部逻辑，违反了开闭原则；`bond` 函数不仅负责羁绊效果的应用，还负责羁绊类型的选择逻辑，违反了单一职责原则；由于所有羁绊逻辑都集中在 `bond` 函数中，新增或修改羁绊时需要深入理解整个函数的逻辑，增加了出错的风险。为了解决这些问题，使用状态模式进行重构是非常必要的。状态模式将每个羁绊效果封装到独立的状态类中，使得对象在其内部状态改变时能够改变其行为，从而提高了代码的可维护性、可扩展性和可读性。

## UML 类图



状态模式 UML 类图

## 重构步骤

状态模式允许对象在其内部状态改变时改变其行为。我们可以将每个羁绊效果封装到一个独立的状态类中，然后在 `bond` 函数中根据当前的羁绊状态执行相应的逻辑。

### 1. 定义状态接口

定义一个 `BondState` 接口，包含一个 `applyEffect` 方法。

```
1 class BondState {
2 public:
3     virtual void applyEffect() = 0;
4     virtual void setContent(Champion* content) = 0;
5     virtual ~BondState() = default;
6 };
```

### 2. 实现具体状态类

为每个羁绊效果实现一个具体的状态类，实现 `applyEffect` 方法。

```
1 class BrotherhoodBond : public BondState {
2 public:
3     void applyEffect() override {
4         this->getAttributes().movementSpeed *=
5             BROTHERHOOD_MOVEMENT_SPEED_MULTIPLIER;
6         this->getAttributes().attackSpeed *=
7             BROTHERHOOD_ATTACK_SPEED_MULTIPLIER;
8     }
9 }
```

```
8     void setContent(Champion* content) override {
9         this->content = content;
10    }
11
12 private:
13     Champion* content;
14 };
15
16 class DarkSideBond : public BondState {
17 public:
18     void applyEffect() override {
19         this->getAttributes().skillTriggerThreshold = static_cast<int>(this-
>getAttributes().skillTriggerThreshold * DARKSIDE_SKILL_TRIGGER_MULTIPLIER);
20         this->getAttributes().attackDamage *=
DARKSIDE_ATTACK_DAMAGE_MULTIPLIER;
21    }
22
23    void setContent(Champion* content) override {
24        this->content = content;
25    }
26
27 private:
28     Champion* content;
29 };
30
31 class GoodShooterBond : public BondState {
32 public:
33     void applyEffect() override {
34         this->getAttributes().attackSpeed *=
GOODSHOOTER_ATTACK_SPEED_MULTIPLIER;
35    }
36
37    void setContent(Champion* content) override {
38        this->content = content;
39    }
40
41 private:
42     Champion* content;
43 };
44
45 class LoutBond : public BondState {
46 public:
47     void applyEffect() override {
48         this->getAttributes().healthPoints = static_cast<int>(this-
>getAttributes().healthPoints * LOUT_HEALTH_POINTS_MULTIPLIER);
49         this->getAttributes().movementSpeed *= LOUT_MOVEMENT_SPEED_MULTIPLIER;
50         this->getAttributes().attackDamage *= LOUT_ATTACK_DAMAGE_MULTIPLIER;
```

```

51     }
52
53     void setContent(Champion* content) override {
54         this->content = content;
55     }
56
57 private:
58     Champion* content;
59 };
60
61 class PopStarBond : public BondState {
62 public:
63     void applyEffect() override {
64         this->getAttributes().attackSpeed *= POPSTAR_ATTACK_SPEED_MULTIPLIER;
65         this->getAttributes().movementSpeed *=
66             POPSTAR_MOVEMENT_SPEED_MULTIPLIER;
67     }
68
69     void setContent(Champion* content) override {
70         this->content = content;
71     }
72
73 private:
74     Champion* content;
75 };

```

### 3. 添加 `changeState` 函数并重构 `bond` 函数

在 `Champion` 类中添加私有变量 `state`，添加 `changeState` 函数，在 `bond` 函数中根据当前的羁绊状态执行相应的逻辑。

```

1 void Champion::changeState(BondState* newState) {
2     if (state) {
3         delete state;
4     }
5     state = newState;
6     if (state) {
7         state->setContent(this);
8     }
9 }
10
11 void Champion::bond() {
12     if (state) {
13         state->applyEffect(attributes);
14     }

```

## 实现的改进

### 1. 消除条件分支

- 通过将每个羁绊效果封装到独立的状态类中，消除了 `bond` 函数中的 `switch-case` 条件分支，使得代码更加简洁和易于维护。

### 2. 遵循开闭原则

- 新增羁绊效果时只需添加一个新的状态类，而无需修改 `bond` 函数的内部逻辑，符合开闭原则（对扩展开放，对修改封闭）。

### 3. 单一职责原则

- `bond` 函数只负责选择和应用状态，而具体的羁绊效果由各个状态类负责，符合单一职责原则。

### 4. 提高扩展性

- 新增或修改羁绊效果时只需添加或修改相应的状态类，而无需深入理解 `bond` 函数的逻辑，降低了出错的风险。

### 5. 代码复用

- 将通用的羁绊逻辑封装到基类或独立的状态类中，提高了代码的复用性。

### 6. 增强可读性

- 每个状态类的职责明确，代码结构更加清晰，易于理解和维护。

### 7. 动态切换状态

- 可以在运行时动态切换羁绊状态，使得 `Champion` 类的行为更加灵活。

## 使用观察者模式（Observer Pattern）重构

### 原有问题

在当前的代码中，`Server` 类直接处理客户端的连接、消息接收和广播逻辑。这种设计存在以下问题：

### 1. 紧耦合

`Server` 类直接与客户端通信逻辑耦合在一起，导致代码难以维护和扩展。如果未来需要添加新的功能（如日志记录、消息过滤等），需要修改 `Server` 类的代码。

```
1 class Server {
2 public:
3     // 构造函数
```

```
4     Server();
5
6     // 析构函数
7     ~Server();
8
9     // 运行服务器
10    void run();
11
12    // 友元函数声明
13    friend void clientHandler(const SOCKET clientSocket, Server& server);
14
15 private:
16    WSADATA wsaData;                                // Windows Sockets
17
18    API
17    SOCKET serverSocket, clientSocket;              // 服务器和客户端的
18    struct sockaddr_in server, client;             // 服务器和客户端的地
19    char hostname[HOSTNAME_MAX_LENGTH];            // 主机名
20    struct hostent* host;                          // 主机地址
21    int port;                                    // 端口
22    int currentConnections;                      // 服务器当前连接数量
23    std::vector<SOCKET> clients;                // 所有连接到服务器的
24    std::vector<std::map<SOCKET, std::string>> playerNames; // 所有连接到服务器的
25    std::vector<std::map<SOCKET, std::string>> battleMaps; // 所有连接到服务器的
26
27    // 创建和尝试绑定套接字
28    void createAndBindSocket();
29
30    // 监听和接受客户端的连接请求并进行处理
31    void handleConnections();
32
33    // 检查所有连接到服务器的客户端信息均已发送
34    bool areAllReady(const std::vector<std::map<SOCKET, std::string>>& data);
35
36    // 关闭客户端套接字
37    void closeClientSocket(const SOCKET clientSocket);
38
39    // 序列化所有连接到服务器的客户端玩家昵称
40    std::string serializePlayerNames();
41
42    // 获取键值对数据
43    std::string getPairedData(const std::vector<std::map<SOCKET,
44    std::string>>& data, size_t index);
```

```

44
45     // 清空所有连接到服务器的客户端玩家字符串数据
46     void clearStrings(std::vector<std::map<SOCKET, std::string>>& data);
47 }

```

## 2. 单一职责原则

`Server` 类承担了过多的职责，包括连接管理、消息处理、广播等，违反了单一职责原则。

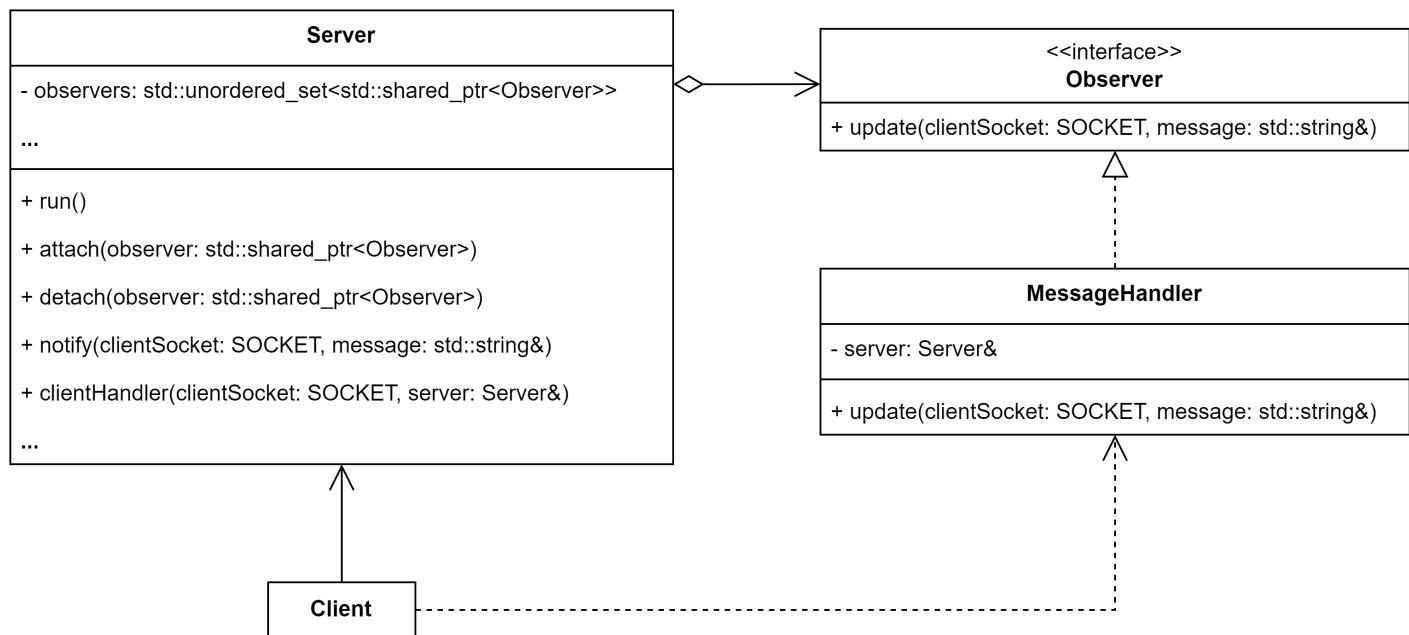
## 3. 可扩展性差

如果未来需要支持不同类型的客户端或消息处理逻辑，现有的设计难以扩展。

### 重构必要性

在当前的代码中，`Server` 类直接处理客户端的连接、消息接收和广播逻辑，导致代码存在紧耦合、违反单一职责原则以及可扩展性差等问题。具体来说，`Server` 类与客户端通信逻辑紧密耦合，难以维护和扩展；`Server` 类承担了过多的职责，包括连接管理、消息处理、广播等，违反了单一职责原则；如果未来需要支持不同类型的客户端或消息处理逻辑，现有的设计难以扩展。为了解决这些问题，使用观察者模式进行重构是非常必要的。观察者模式将消息处理逻辑从`Server`类中解耦出来，使得`Server`类只负责连接管理和通知观察者，从而提高了代码的可维护性、可扩展性和可读性。

## UML 类图



观察者模式 UML 类图

### 重构步骤

为了使用观察者模式重构代码，我们可以将消息处理逻辑从`Server`类中解耦出来，让`Server`类只负责连接管理和通知观察者。具体步骤如下：

#### 1. 定义观察者接口

创建一个 `Observer` 接口，定义观察者需要实现的方法（如 `update`）。

```
1 class Observer {
2 public:
3     virtual ~Observer() = default;
4     virtual void update(const SOCKET clientSocket, const std::string& message)
5         = 0;
6 };
```

## 2. 创建具体观察者

实现具体的观察者类（如 `MessageHandler`），负责处理客户端的消息。

```
1 class MessageHandler : public Observer {
2 public:
3     MessageHandler(Server& server) : server(server) {}
4     void update(const SOCKET clientSocket, const std::string& message)
5         override;
6
7 private:
8     Server& server;
9 };
```

## 3. 修改 `Server` 类

将 `Server` 类改为被观察者（`Subject`），并维护一个观察者列表。当有新的消息到达时，`Server` 类通知所有观察者。

```
1 class Server {
2 public:
3     ...
4
5 private:
6     ...
7
8     std::unordered_set<std::shared_ptr<Observer>> observers;
9
10    ...
11 };
```

## 4. 客户端处理逻辑

将原有的客户端处理逻辑移动到具体的观察者类中。

```
1 void MessageHandler::update(const SOCKET clientSocket, const std::string&
2   message) {
3     char buffer[BUFFER_SIZE];
4     strncpy(buffer, message.c_str(), BUFFER_SIZE);
5
6     std::time_t now =
7       std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
8     std::cout << std::put_time(std::localtime(&now), "[%H:%M:%S]");
9
10    if (!strcmp(buffer, BATTLE_MAP_IDENTIFIER, MESSAGE_IDENTIFIER_LENGTH)) {
11      char battleMap[BUFFER_SIZE];
12      sscanf(buffer, BATTLE_MAP_FORMAT, battleMap);
13      for (auto& map : server.battleMaps) {
14        if (map.find(clientSocket) != map.end()) {
15          map[clientSocket] = static_cast<std::string>(battleMap);
16          break;
17        }
18      }
19    }
20
21    if (!strcmp(buffer, PLAYER_NAME_IDENTIFIER, MESSAGE_IDENTIFIER_LENGTH)) {
22      char playerName[BUFFER_SIZE];
23      sscanf(buffer, PLAYER_NAME_FORMAT, playerName);
24      for (auto& map : server.playerNames) {
25        if (map.find(clientSocket) != map.end()) {
26          map[clientSocket] = static_cast<std::string>(playerName);
27          break;
28        }
29      }
30    }
31
32    if (!strcmp(buffer, HEALTH_POINTS_IDENTIFIER, MESSAGE_IDENTIFIER_LENGTH))
33    {
34      for (const SOCKET& sock : server.clients) {
35        send(sock, buffer, strlen(buffer), 0);
36      }
37
38      if (server.currentConnections % 2 == 0) {
39        if (server.areAllReady(server.playerNames)) {
40          strcpy(buffer, server.serializePlayerNames().c_str());
```

```
41         std::cout << "Broadcast: " << buffer << std::endl;
42         for (const SOCKET& sock : server.clients) {
43             send(sock, buffer, strlen(buffer), 0);
44         }
45     }
46 }
47
48 if (server.areAllReady(server.battleMaps)) {
49     for (size_t i = 0; i < server.clients.size(); i++) {
50         sprintf(buffer, BATTLE_MAP_FORMAT,
51             server.getPairedData(server.battleMaps, i).c_str());
52         std::cout << "Send Client " << server.clients[i ^ 1] << "[" << (i
53             ^ 1) << "]'s BattleMap to Client " << server.clients[i] << "[" << i << "]: "
54             << buffer << std::endl;
55         send(server.clients[i], buffer, strlen(buffer), 0);
56     }
57     server.clearStrings(server.battleMaps);
58 }
```

## 实现的改进

### 1. 解耦与抽象

- 通过定义 `Observer` 接口，将消息处理逻辑从 `Server` 类中解耦出来，使得 `Server` 类只负责连接管理和通知观察者。

### 2. 单一职责原则

- `Server` 类不再承担消息处理的职责，而是专注于连接管理，符合单一职责原则。

### 3. 提高扩展性

- 新增消息处理逻辑时只需添加新的观察者类，而无需修改 `Server` 类的代码，提高了代码的可扩展性。

### 4. 动态添加观察者

- 可以在运行时动态添加或移除观察者，使得系统更加灵活。

### 5. 代码复用

- 将通用的消息处理逻辑封装到基类或独立的观察者类中，提高了代码的复用性。

### 6. 增强可读性

- 每个观察者类的职责明确，代码结构更加清晰，易于理解和维护。

### 7. 支持多种消息类型

- 可以为不同类型的消息创建不同的观察者类，使得消息处理逻辑更加模块化。

# 使用课程未涉及的设计模式重构

## 使用反应器模式（Reactor Pattern）重构

### 反应器模式（Reactor Pattern）

Reactor Pattern（反应器模式）是一种事件处理策略，用于并发处理多个服务请求。该模式的核心组件是一个事件循环（Event Loop），通常运行在单线程或单进程中，负责多路复用（Demultiplex）传入的请求，并将它们分发给正确的请求处理器（Request Handler）。

#### 核心思想

- **事件驱动：**Reactor 模式通过事件通知机制来处理 I/O 操作，而不是使用阻塞式 I/O 或多线程。这使得它能够在单线程中高效处理大量并发 I/O 请求。
- **非阻塞 I/O：**Reactor 模式依赖于非阻塞 I/O，只有在 I/O 操作完成后才会触发事件通知，从而避免了线程因等待 I/O 而阻塞的问题。
- **回调机制：**请求处理器以回调函数的形式注册到事件循环中，事件循环在接收到事件后调用相应的回调函数进行处理。这种设计实现了事件处理逻辑与事件分发逻辑的分离，提高了代码的灵活性和可维护性。

#### 工作流程

- **注册事件处理器：**在事件循环启动前，应用程序将句柄和对应的事件处理器注册到分发器中。
- **事件循环：**事件循环通过多路复用器监控所有注册的句柄，等待事件发生。
- **事件触发：**当某个句柄的 I/O 操作完成时，多路复用器通知分发器。
- **调用处理器：**分发器根据事件类型调用相应的事件处理器进行处理。

#### 优点

- **高并发性：**通过非阻塞 I/O 和事件驱动机制，Reactor 模式能够在单线程中高效处理大量并发请求。
- **低资源消耗：**避免了多线程带来的上下文切换和内存开销。
- **灵活性：**事件处理器以回调形式注册，便于扩展和修改。

#### 缺点

- **调试困难：**由于使用了回调机制，程序的执行流程变得复杂，增加了调试和分析的难度。
- **单线程瓶颈：**对于计算密集型任务，单线程可能成为性能瓶颈。
- **复杂性：**相比简单的“线程 per 连接”模型，Reactor 模式的实现更为复杂。

#### 应用场景

Reactor 模式广泛应用于需要高并发处理的场景，尤其是 I/O 密集型应用，例如：

- **Web 服务器**: 如 Nginx、Node.js
- **网络框架**: 如 Netty、Twisted
- **应用服务器**: 如 Spring Framework

## 原有问题

当前代码的主要问题在于：

### 1. 阻塞式 I/O

`listenForServerMessages` 函数使用阻塞式 I/O 操作，这会导致线程在等待数据时无法执行其他任务，降低了程序的响应性。

```
1 void OnlineModeControl::listenForServerMessages()
2 {
3     while (keepListening) {
4         char buffer[BUFFER_SIZE];
5         recv(s, buffer, BUFFER_SIZE, 0);
6         if (!strcmp(buffer, HEALTH_POINTS_IDENTIFIER,
7             MESSAGE_IDENTIFIER_LENGTH)) {
8             int healthPoints;
9             SOCKET socket;
10            sscanf(buffer, HEALTH_POINTS_FORMAT, &healthPoints, &socket);
11            updatePlayerHealthPoints(healthPoints, socket);
12        }
13    }
14 }
```

### 2. 线程管理复杂

代码中使用了显式的线程管理 (`std::thread`)，这增加了代码的复杂性，并且容易引发线程安全问题。

```
1 OnlineModeControl::OnlineModeControl(std::string ipv4, std::string portStr) :
2     port(std::stoi(portStr)),
3     recvSize(0),
4     currentConnections(0),
5     keepListening(true),
6     Control(MAX_CONNECTIONS)
7 {
8     strcpy(this->message, "");
```

```
9     strcpy(this->ipv4, ipv4.c_str());
10    try {
11        humanPlayer = new HumanPlayer(g_PlayerName);
12        enemyPlayer = new HumanPlayer("");
13    }
14    catch (const std::bad_alloc& e) {
15        std::cerr << "Memory allocation failed: " << e.what() << std::endl;
16        if (humanPlayer) {
17            delete humanPlayer;
18        }
19        if (enemyPlayer) {
20            delete enemyPlayer;
21        }
22        throw;
23    }
24    listeningThread = std::thread(&OnlineModeControl::listenForServerMessages,
25        this);
26 }
```

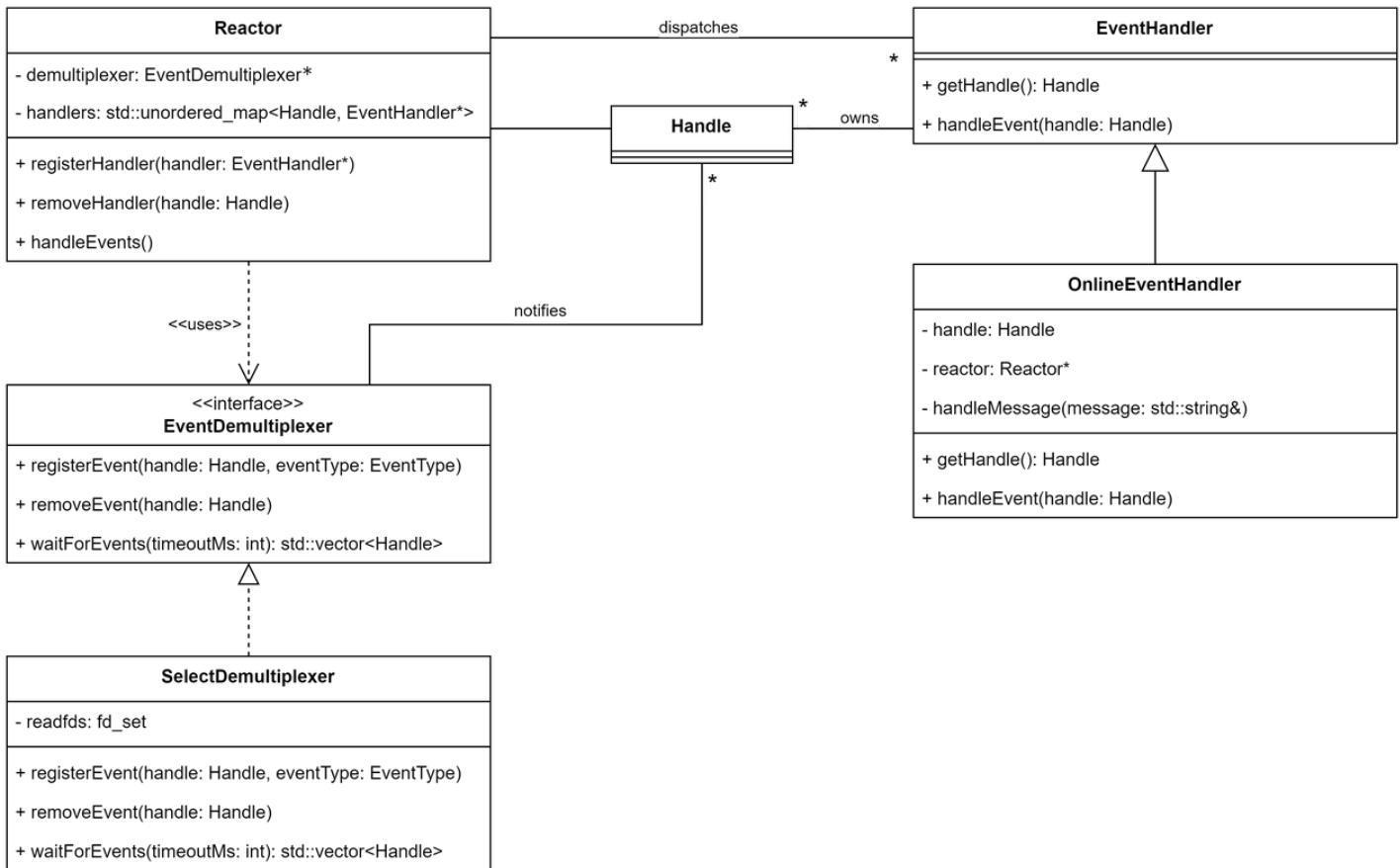
### 3. 缺乏事件驱动机制

当前的代码没有充分利用事件驱动机制，导致代码的可扩展性和可维护性较差。

#### 重构必要性

在当前的代码中，`listenForServerMessages` 函数使用阻塞式 I/O 操作，导致线程在等待数据时无法执行其他任务，降低了程序的响应性。同时，代码中使用了显式的线程管理（`std::thread`），增加了代码的复杂性，并且容易引发线程安全问题。此外，当前的代码没有充分利用事件驱动机制，导致代码的可扩展性和可维护性较差。为了解决这些问题，使用 Reactor Pattern 进行重构是非常必要的。Reactor 模式通过事件驱动和非阻塞 I/O 机制，能够在单线程中高效处理大量并发请求，从而提高了代码的响应性、可扩展性和可维护性。

#### UML 类图



反应器模式 UML 类图

## 重构步骤

为了使用 Reactor Pattern 重构代码，我们需要：

### 1. 定义事件类型和句柄

- 在 `EventDemultiplexer.h` 中定义 `EventType` 枚举和 `Handle` 类型。
- `EventType` 用于区分不同类型的事件（如读事件、写事件），`Handle` 是文件描述符（如套接字）的类型别名。

```

1 enum class EventType {
2     READ_EVENT,
3     WRITE_EVENT
4 };
5
6 using Handle = SOCKET;
  
```

### 2. 实现事件多路分发器

- 在 `EventDemultiplexer.h` 中实现 `EventDemultiplexer` 接口和 `SelectDemultiplexer` 类。

- `EventDemultiplexer` 负责监听多个文件描述符的事件，`SelectDemultiplexer` 是具体的实现，使用 `select` 系统调用。

```
1 class EventDemultiplexer {
2 public:
3     virtual ~EventDemultiplexer() = default;
4     virtual void registerEvent(Handle handle, EventType eventType) = 0;
5     virtual void removeEvent(Handle handle) = 0;
6     virtual std::vector<Handle> waitForEvents(int timeoutMs) = 0;
7 };
8
9 class SelectDemultiplexer : public EventDemultiplexer {
10 public:
11     SelectDemultiplexer() {
12         FD_ZERO(&readfds);
13     }
14
15     void registerEvent(Handle handle, EventType eventType) override {
16         FD_SET(handle, &readfds);
17     }
18
19     void removeEvent(Handle handle) override {
20         FD_CLR(handle, &readfds);
21     }
22
23     std::vector<Handle> waitForEvents(int timeoutMs) override {
24         fd_set tmpfds = readfds;
25         struct timeval timeout = {0, timeoutMs * 1000};
26
27         int selectResult = select(0, &tmpfds, nullptr, nullptr, &timeout);
28         if (selectResult > 0) {
29             std::vector<Handle> activeHandles;
30             for (int i = 0; i < FD_SETSIZE; ++i) {
31                 if (FD_ISSET(i, &tmpfds)) {
32                     activeHandles.push_back(i);
33                 }
34             }
35             return activeHandles;
36         }
37         return {};
38     }
39
40 private:
41     fd_set readfds;
42 };
```

### 3. 定义事件处理器接口

- 在 `EventHandler.h` 中定义 `EventHandler` 抽象类。
- `EventHandler` 是所有具体事件处理器的基类，定义了处理事件的接口。

```
1 class EventHandler {
2 public:
3     virtual ~EventHandler() = default;
4     virtual void handleEvent(Handle handle) = 0;
5     virtual Handle getHandle() const = 0;
6 };
```

### 4. 实现具体的事件处理器

- 在 `OnlineEventHandler.h` 中实现 `OnlineEventHandler` 类。
- `OnlineEventHandler` 是具体的事件处理器，负责处理网络消息并更新玩家状态。

```
1 class OnlineEventHandler : public EventHandler {
2 public:
3     OnlineEventHandler(Handle handle, Reactor* reactor)
4         : handle(handle), reactor(reactor) {}
5
6     void handleEvent(Handle handle) override {
7         char buffer[BUFFER_SIZE];
8         int recvSize = recv(handle, buffer, BUFFER_SIZE, 0);
9         if (recvSize > 0) {
10             buffer[recvSize] = '\0';
11             handleMessage(buffer);
12         }
13     }
14
15     Handle getHandle() const override {
16         return handle;
17     }
18
19 private:
20     void handleMessage(const std::string& message) {
21         if (!strcmp(message.c_str(), HEALTH_POINTS_IDENTIFIER,
22                     MESSAGE_IDENTIFIER_LENGTH)) {
23             int healthPoints;
24             SOCKET socket;
25             sscanf(message.c_str(), HEALTH_POINTS_FORMAT, &healthPoints,
26                   &socket);
27             updatePlayerHealthPoints(healthPoints, socket);
28         }
29     }
30 }
```

```
26     }
27 }
28
29 Handle handle;
30 Reactor* reactor;
31 };
```

## 5. 实现 `Reactor` 类

- 在 `Reactor.h` 中实现 `Reactor` 类。
- `Reactor` 是事件循环的核心，负责注册事件处理器、监听事件并调用相应的回调。

```
1 class Reactor {
2 public:
3     Reactor() : demultiplexer(new SelectDemultiplexer()) {}
4
5     ~Reactor() {
6         delete demultiplexer;
7     }
8
9     void registerHandler(EventHandler* handler) {
10        handlers[handler->getHandle()] = handler;
11        demultiplexer->registerEvent(handler->getHandle(),
12            EventType::READ_EVENT);
13    }
14
15    void removeHandler(Handle handle) {
16        demultiplexer->removeEvent(handle);
17        handlers.erase(handle);
18    }
19
20    void handleEvents() {
21        auto activeHandles = demultiplexer->waitForEvents(100);
22        for (auto handle : activeHandles) {
23            if (handlers.find(handle) != handlers.end()) {
24                handlers[handle]->handleEvent(handle);
25            }
26        }
27    }
28 private:
29     EventDemultiplexer* demultiplexer;
30     std::unordered_map<Handle, EventHandler*> handlers;
31 };
```

## 6. 重构主逻辑

- 在主逻辑中初始化 `Reactor` 和 `OnlineEventHandler`，并启动事件循环。
- 通过 `Reactor` 的事件循环，主逻辑可以高效地处理多个并发事件，而无需显式管理线程。

```
1 class OnlineModeControl {
2 public:
3     OnlineModeControl(std::string ipv4, std::string portStr)
4         : port(std::stoi(portStr)), keepListening(true) {
5     ...
6     reactor = new Reactor();
7     eventHandler = new ConcreteEventHandler(s, reactor);
8     reactor->registerHandler(eventHandler);
9 }
10
11 ~OnlineModeControl() {
12     ...
13     delete reactor;
14     delete eventHandler;
15 }
16
17 void start() {
18     while (keepListening) {
19         reactor->handleEvents();
20
21         std::this_thread::sleep_for(std::chrono::milliseconds(THREAD_SLEEP_DURATION_MILLISECONDS));
22     }
23
24     ...
25
26 private:
27     ...
28     Reactor* reactor;
29     EventHandler* eventHandler;
30     SOCKET s;
31     int port;
32     bool keepListening;
33 };
```

## 实现的改进

### 1. 非阻塞 I/O 提升响应性

- **改进点：**原有的 `listenForServerMessages` 函数使用了阻塞式 I/O 操作（`recv`），导致线程在等待数据时无法执行其他任务，降低了程序的响应性。
- **重构后：**通过 Reactor 模式，I/O 操作变为非阻塞式，事件循环（Event Loop）可以在等待 I/O 操作完成的同时处理其他事件，显著提升了程序的响应性。

## 2. 简化线程管理

- **改进点：**原有的代码使用了显式的线程管理（`std::thread`），增加了代码的复杂性，并且容易引发线程安全问题。
- **重构后：**Reactor 模式通过单线程事件循环处理所有并发请求，避免了显式线程管理的复杂性，减少了线程上下文切换的开销，同时降低了线程安全问题的风险。

## 3. 事件驱动机制提升可扩展性

- **改进点：**原有的代码缺乏事件驱动机制，导致代码的可扩展性和可维护性较差。
- **重构后：**Reactor 模式通过事件驱动机制，将事件处理逻辑与事件分发逻辑分离，使得代码更易于扩展和维护。新增事件类型或处理器时，只需注册相应的事件处理器即可，无需修改核心逻辑。

## 4. 资源消耗降低

- **改进点：**原有的多线程模型会带来较高的上下文切换和内存开销。
- **重构后：**Reactor 模式在单线程中处理所有并发请求，避免了多线程带来的上下文切换和内存开销，降低了资源消耗。

## 5. 代码结构更清晰

- **改进点：**原有的代码结构较为混乱，事件处理逻辑与线程管理逻辑耦合在一起。
- **重构后：**Reactor 模式将事件处理逻辑与事件分发逻辑分离，代码结构更加清晰，便于理解和维护。

## 6. 支持高并发处理

- **改进点：**原有的阻塞式 I/O 和多线程模型在处理大量并发请求时性能较差。
- **重构后：**Reactor 模式通过非阻塞 I/O 和事件驱动机制，能够在单线程中高效处理大量并发请求，特别适合 I/O 密集型应用。

# 项目展示

# 金铲铲之战

## 健康游戏忠告

抵制不良游戏，拒绝盗版游戏。  
注意自我保护，谨防受骗上当。  
适度游戏益脑，沉迷游戏伤身。  
合理安排时间，享受健康生活。

21%

本游戏基于 Cocos2d-x 3.17.2 开发  
Copyright (c) 2023 Jishen Lin, Shuyi Liu, Zhaozhen Yang, Yukun Yang





# 设置

Settings

背景音乐

音 效

练习模式

简单    正常    困难

游戏难度



返回菜单

## 游戏部分玩法介绍

1. 羁绊系统：本游戏共设置五种羁绊，满足条件会对局面有较大增益。



死神 (DarkSide)  
伊芙琳、阿木木、卡尔萨斯



神射手 (GoodShooter)  
库奇、金克斯



莽夫 (Lout)  
奥拉夫、潘森



星之守护 (PopStar)  
卡莎



兄弟 (Brotherhood)  
永恩、亚索

2. 天赋系统：本游戏共设置三种天赋，对局面有较大增益，可以在游戏开始前自选。



海盗：增加每回合金币数量

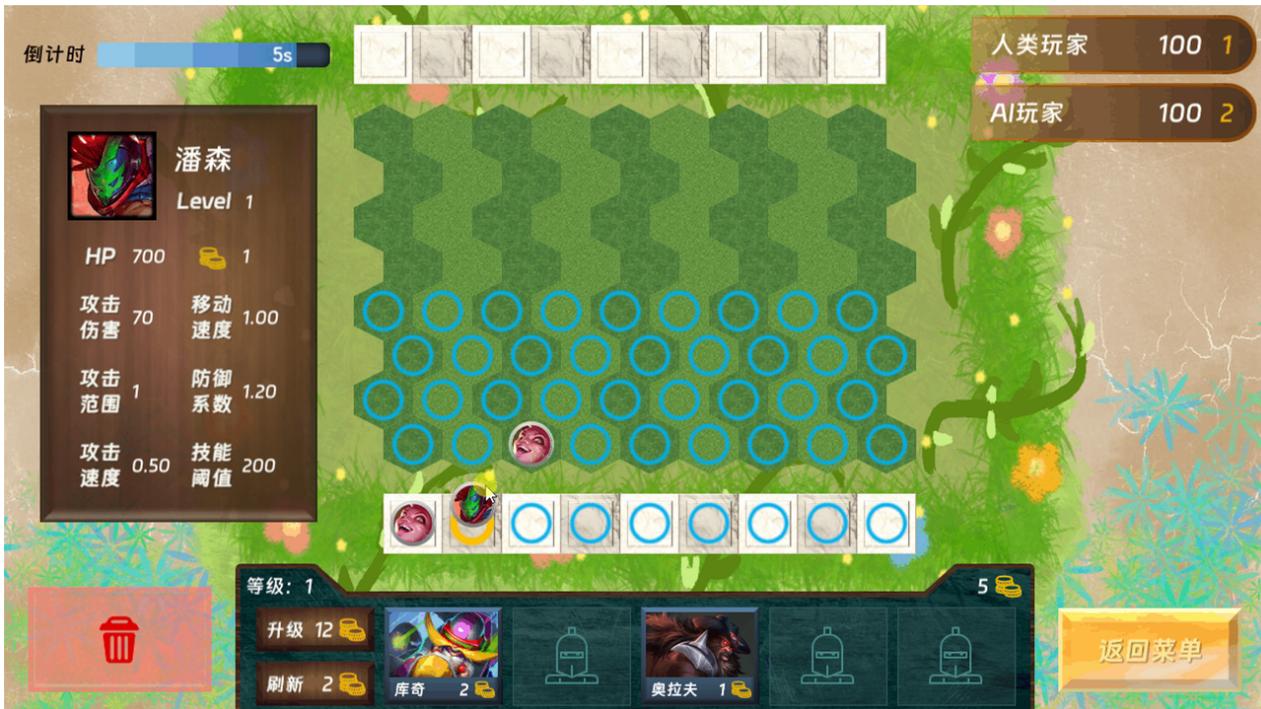


将军：增加英雄位



斗士：增加初始生命值

返回菜单



## 总结

### 使用设计模式重构的意义

设计模式是软件开发中经过验证的最佳实践，能够帮助开发者解决常见的代码设计问题。重构的意义在于：

- 提升代码质量：**重构通过优化代码结构，使其更易理解和维护。清晰的代码不仅减少了开发人员的认知负担，还能降低出错的可能性，从而提高整体代码质量。
- 提高可维护性：**重构使代码更易于修改和扩展。随着项目的发展，清晰的代码结构能够显著降低维护成本，避免技术债务的积累，确保系统长期稳定运行。

- **优化性能**：重构可以消除代码中的性能瓶颈，提升系统运行效率。通过优化算法和资源使用，重构能够减少不必要的资源消耗，从而提高整体性能。
- **支持新功能**：重构使代码更具扩展性，便于添加新功能。清晰的代码结构减少了新功能与现有代码之间的冲突风险，使开发过程更加顺畅。
- **促进团队协作**：重构有助于统一代码风格和标准，使团队成员更容易理解和协作。清晰的代码还能帮助新成员快速上手，促进知识共享和团队合作。
- **降低风险**：重构过程中可以发现并修复潜在问题，从而减少缺陷。优化后的代码更加稳定，降低了系统崩溃和错误发生的可能性。
- **适应变化**：重构使代码更具灵活性，能够更好地应对业务需求的变化。同时，重构也便于引入新技术和工具，确保系统与时俱进。
- **提高开发效率**：重构减少了调试时间，使开发过程更加高效。清晰的代码结构还便于进行自动化测试，进一步提升开发效率和代码可靠性。

设计模式重构不仅解决了当前代码中的问题，还为未来的开发奠定了良好的基础。

## 本项目的重构方法

我们对本项目中的多个模块进行了重构，主要应用了以下几种设计模式：

- **生成器模式（Builder Pattern）**：重构了 Champion 类的初始化逻辑，解决了硬编码和初始化复杂的问题，提升了代码的可维护性和扩展性。
- **单例模式（Singleton Pattern）**：应用于 LocationMap 类，确保全局只有一个实例，避免了多实例带来的内存浪费和数据不一致问题。
- **组合模式（Composite Pattern）**：重构了 ChampionAttributesLayer 类，解耦了具体的节点类型，减少了代码重复，增强了灵活性。
- **外观模式（Facade Pattern）**：将音频播放逻辑封装在 AudioController 类中，简化了外部调用，降低了耦合度。
- **策略模式（Strategy Pattern）**：重构了 Champion 类的技能逻辑，消除了大量的条件分支，提升了代码的可扩展性和可维护性。
- **状态模式（State Pattern）**：应用于 Champion 类的羁绊逻辑，将不同的羁绊效果封装到独立的状态类中，提升了代码的可扩展性。
- **观察者模式（Observer Pattern）**：解耦了 Server 类的消息处理逻辑，使得系统更加灵活，易于扩展。
- **反应器模式（Reactor Pattern）**：引入了事件驱动和非阻塞I/O机制，替代了原有的阻塞式I/O，提升了系统的并发处理能力。

通过这些重构，项目的代码质量得到了显著提升，系统的可维护性、扩展性和性能都得到了优化。

## 总结

通过本文的重构案例，可以学到以下知识：

- **设计模式的应用场景**：不同的设计模式适用于不同的场景。例如，生成器模式适用于复杂对象的创建，单例模式适用于全局唯一实例的管理，策略模式适用于算法的动态切换等。
- **代码解耦的重要性**：通过设计模式（如外观模式、观察者模式），可以将系统中的各个模块解耦，降低代码的复杂性，提升系统的灵活性和可维护性。
- **重构的步骤**：重构不仅仅是修改代码，还需要分析现有问题、设计解决方案、逐步实施并验证效果。本文展示了如何通过UML类图和代码示例来逐步实施重构。
- **性能优化**：通过反应器模式，可以提升系统的并发处理能力，减少资源消耗。这对于高并发的应用场景（如网络游戏）尤为重要。
- **代码的可读性和可维护性**：设计模式不仅解决了技术问题，还使得代码结构更加清晰，易于理解和维护。这对于团队协作和长期项目维护非常重要。

## 参考资料

1. Refactoring.Guru (<https://refactoring.guru>)
2. 设计模式 ([https://en.wikipedia.org/wiki/Design\\_Patterns](https://en.wikipedia.org/wiki/Design_Patterns))
3. 生成器模式 ([https://en.wikipedia.org/wiki/Reactor\\_pattern](https://en.wikipedia.org/wiki/Reactor_pattern))